Apprenez à programmer en Java

Par cysboy

www.siteduzero.com

Licence Creative Commons BY-NC-SA 2.0
Dernière mise à jour le 7/12/2011

Sommaire

Sommaire 1
Apprenez à programmer en Java 5
Partie 1 : Bien commencer en Java 6
Parlons d'Eclipse 7
Préambule 7
Téléchargement 9
Installation 9
Présentation rapide de l'interface 12
D'autres outils à la hauteur 18
1. JBuilder 18
2. NetBeans 18
Votre premier programme 19
Mieux connaitre son environnement Java 19
Avant de commencer 21
Votre premier programme 25
Compilation en ligne de commande (Windows) 29
Les variables et les opérateurs 31
Les différents types de variables 32
Les opérateurs arithmétiques 35
Les opérateurs arithmétiques 36
Les conversions, ou "cast" 38
Astuce d'Eclipse 41
Lire les entrées clavier 42
Utilisation de l'objet Scanner 42
Récupérez ce que vous tapez 44
Les conditions 48
Les opérateurs logiques 48
La structure if....else 50
Les conditions multiples 52
La structure switch 55
La condition ternaire 56
Les boucles 58
La boucle while 58
La boucle do....while 64
La boucle for 65
TP n°1 : un tableau de conversion Celsius - Fahrenheit ! 68
Élaboration 68
Conception 71
Correction 73
Les tableaux 75
Déclarer et initialiser un tableau 76
Les tableaux multi-dimensionnels 77
Utiliser et rechercher dans un tableau ! 79
Un tableau multi-dimensionnel 82
Les méthodes de classe 87
Quelques méthodes bien utiles ! 88
Des méthodes concernant les mathématiques 90
Créer et utiliser sa propre méthode ! 91
La surcharge de méthode 95
Partie 2 : Java Orienté Objet 97
Les premiers pas en "Orienté Objet" 98
Les classes 99
Les constructeurs 100
Votre objet sait parler : accesseurs et mutateurs 108
Travaillez avec votre objet : les méthodes 111
Les variables de classes 117
Astuce Eclipse 121
Ce qu'il faut retenir 124
L'héritage ! 125
La notion d'héritage 126
Construction d'un objet hérité 130
Le polymorphisme 133
Ce qu'il faut retenir 139
Apprendre à modéliser 140
UML, mais qu'est-ce donc ? 141
Modéliser un objet 142
Modéliser les interactions entre objets 144
Les classes abstraites 147
Qu'est-ce que c'est ? 147
Une classe Animal très abstraite 149
Étoffons notre exemple 152
Astuce d'Eclipse 158
Ce qu'il faut retenir 161

Les interfaces ... 161
Une petite devinette ... 161
Votre première interface ... 162
Implémentation de l'interface Rintintin ... 164
Astuce d'Eclipse ... 167
Ce qu'il faut retenir ... 171
Les exceptions ... 172
Premier exemple d'exception et le bloc try{....} catch{...} ... 173
Les exceptions personnalisées ... 176
La gestion de plusieurs exceptions ... 183
Astuce d'Eclipse ... 186
Ce qu'il faut retenir ... 188
Les collections d'objets ... 189
L'objet LinkedList ... 189
L'objet ArrayList ... 191
L'objet Hashtable ... 193
L'objet HashSet ... 193
La généricité en Java ... 195
Notion de base ... 196
Plus loin dans la généricité ! ... 199
Généricité et collection ... 202
Héritage et généricité ... 205
Ce qu'il faut retenir ... 208
Java et la réflexivité ... 209
Commençons par le commencement ... 209
Interroger un objet Class ... 211
Instanciation dynamique ... 216
Ce qu'il faut retenir ... 219
Partie 3 : Java et la programmation événementielle ... 220
Votre première fenêtre ... 221
L'objet JFrame ... 222
Votre fenêtre héritée ... 225
Des méthodes et encore des méthodes ... 225
Ce que vous cache votre fenêtre ... 226
Ce qu'il faut retenir ... 228
Une histoire de conteneur ... 228
Créez un conteneur pour votre fenêtre ... 228
L'objet Graphics ... 231
Plus loin dans le Graphics ... 232
Le cousin caché : l'objet Graphics2D ... 240
Ce qu'il faut retenir ... 246
Faire une animation simple ... 247
Les déplacements : principe ... 248
Continue, ne t'arrêtes pas si vite ! ... 252
Attention aux bords, ne va pas te faire mal... ... 254
Ce qu'il faut retenir ... 257
Votre premier bouton ... 257
Utiliser la classe JButton ... 257
Les layout managers ... 261
Continuons dans notre lancée ... 266
Une classe Bouton personnalisée ... 269
Interaction avec la souris : l'interface MouseListener ... 272
Ce qu'il faut retenir ... 281
Interaction bouton(s) - application ... 281
Déclencher une action : l'interface ActionListener ... 281
Parlez avec votre classe intérieure ... 290
Contrôler votre animation : lancement et arrêt ... 295
Cadeau : votre bouton personnalisé optimisé ! ... 301
Ce qu'il faut retenir ... 304
TP : une calculatrice ... 305
Élaboration ... 305
Conception ... 307
Correction ... 307
Astuce Eclipse : faire un jar exécutable ... 313
Les threads ... 322
Principes et bases ... 322
Une classe héritée de Thread ... 323
Utiliser l'interface Runnable ... 328
Synchronisez vos threads ... 333
Contrôlez votre animation ... 336
Ce qu'il faut retenir ... 339
Les listes : l'objet JComboBox ... 339
Première utilisation ... 339
L'interface ItemListener ... 343
Changer la forme de notre animation ... 346
Ce qu'il faut retenir ... 352
Les cases à cocher : l'objet JCheckBox ... 353
Premier exemple ... 354
Un pseudo-morphing pour notre animation ... 357
Les boutons radio : l'objet JRadioButton ... 364

Ce qu'il faut retenir ... 369
Les champs de texte : l'objet JTextField ... 369
Utilisation ... 369
Un objet plus restrictif : le JFormattedTextField ... 372
Contrôlez vos données post-saisie ... 378
Première approche ... 379
Utiliser des expressions régulières ... 382
Utiliser des regex ... 383
Les regex et l'objet String ... 384
Les regex et l'API regex ... 387
Contrôle du clavier : l'interface KeyListener ... 390
Utiliser les classes anonymes ... 395
Ce qu'il faut retenir ... 399
Les applets ... 400
Les applets : quésaco ? ... 401
Votre première applet ... 403
Codage de l'applet ... 403
Insertion dans une page HTML ... 405
Nota Bene ... 408
Interaction page web - applet ... 409
Interaction applet - page web ... 412
Applets et sécurité ... 415
Ce que vous devez retenir ... 416
Les boîtes de dialogue ... 417
Les boîtes d'information ... 418
Les types de boîtes ... 421
Les boîtes de confirmation ... 422
Les boîtes de saisie ... 430
Des boîtes de dialogue personnalisées ... 433
Voici des screenshots obtenus ... 441
Ce qu'il faut retenir ... 442
Les menus ... 443
La barre de menus et les éléments de menu ... 444
Un menu simple ... 444
Les raccourcis clavier ... 456
Faire un menu contextuel ... 460
Les points importants pour notre menu contextuel ... 461
Les barres d'outils ... 479
Utiliser les actions abstraites ... 489
Ce qu'il faut retenir ... 500
TP : l'ardoise maZique ... 500
Cahier des charges ... 500
Prérequis ... 503
Correction ... 503
Point.java ... 504
DrawPanel.java ... 505
Fenetre.java ... 507
Améliorations possibles ... 510
Les flux d'entrées / sorties (1/2) ... 511
Les flux : qu'est-ce donc ? ... 512
L'objet File ... 512
Les objets FileInputStream et FileOutputStream ... 515
Les flux filtrés : les classes FilterInputStream et FilterOutputStream ... 518
La sérialisation : les objets ObjectInputStream et ObjectOutputStream ... 523
Cas pratique ... 529
Ce qu'il faut retenir ... 538
Les flux d'entrées / sorties (2/2) ... 539
Les objets CharArray(Writer/Reader) et String(Writer/Reader) ... 539
les classes File(Writer/Reader) et Print(Writer/Reader) ... 541
Du renouveau chez les flux : le package java.nio ... 544
TP : Le penduZ ... 546
Cahier des charges ... 547
Prérequis ... 548
Correction ... 550
Gérez vos conteneurs ... 551
Rendre vos conteneurs fractionnables ... 551
Ajouter des scrolls ... 556
Avoir plusieurs contenus ... 560
Ce qu'il faut retenir ... 565
Les arbres ... 566
La composition des arbres ... 567
Des arbres qui vous parlent ... 571
Décorez vos arbres ... 579
Jouons avec nos arbres ... 585
Ce que vous devez retenir ... 600
Les tableaux, les vrais ... 601
Premiers pas ... 602
Les cellules, c'est la vie ... 603
Contrôlez l'affichage de vos cellules ... 611
Des tableaux très actifs ! ... 617
Ajouter des lignes et des colonnes ... 625

Ce qu'il faut retenir ... 637
Ce que vous pouvez voir en plus ... 638
D'autres conteneurs graphiques ... 638
D'autres objets graphiques ... 641
Enjoliver vos IHM ... 646
Partie 4 : Les Design patterns ... 650
Les limites de l'héritage : le pattern strategy ... 651
Posons le problème ... 652
Voici le tableau ... 652
Le code source de ces classes ... 653
Un problème supplémentaire ... 660
Une solution simple et robuste : le pattern strategy ... 664
Ce qu'il faut retenir ... 676
Ajouter des fonctionnalités dynamiquement à vos objets : le pattern decorator ... 677
Posons le problème ... 678
Le pattern decorator ... 682
Les mystères de java.io ... 690
Ce qu'il faut retenir ... 693
Soyez à l'écoute de vos objets : le pattern observer ... 694
Posons le problème ... 694
Des objets qui parlent et qui écoutent : le pattern observer ... 697
le pattern observer : le retour ... 705
Ce qu'il faut retenir ... 705
Un pattern puissant : le pattern MVC ... 706
Premiers pas ... 707
Le modèle ... 708
Le contrôleur ... 713
La vue ... 716
MVC pour le web : le pattern M2VC ... 722
Ce qu'il faut retenir ... 723
Un véritable assemblage : le pattern composite ... 724
Creusez-vous les méninges ... 725
Comment feriez-vous pour gérer ce genre de hiérarchie d'objets ? ... 725
La solution : le pattern composite ... 726
Le composite et Java ... 735
Ce qu'il faut retenir ... 737
Partie 5 : Annexes ... 738
Annexe A : liste des mots clés ... 739
Les mots clés ... 740
Annexe B : Les objets travaillant avec des flux ... 742
Les objets traitant des flux d'entrée (in) ... 743
Sous-classes de InputStream ... 743
Sous-classes de Reader ... 743
Les objets traitant les flux de sortie (out) ... 743
Sous-classes de OutputStream ... 744
Sous-classes de Writer ... 744
Annexe C : Eclipse ... 744
Installation ... 744
Installation Windows ... 745
Installation Linux ... 745
Raccourcis utiles ... 746
Partager ... 748

Apprenez à programmer en Java

Par cysboy

Mise à jour : 12/02/2010
Difficulté : Intermédiaire

BY-NC-SA

114 785 visites depuis 7 jours, classé 4/794

Bonjour à tous, amis Zéros !

Bienvenue dans mon cours de programmation en Java. C'est un langage très utilisé, notamment par un grand nombre de programmeurs professionnels, ce qui en fait un langage incontournable actuellement.

Voici les caractéristiques de Java en quelques mots :

- Java est un langage de programmation moderne développé par **Sun Microsystems** (aujourd'hui racheté par **Oracle**). Il ne faut surtout pas le confondre avec JavaScript (langage de scripts utilisé principalement sur les sites web), car Java n'a rien à voir.
- Une de ses plus grandes forces est son excellente portabilité : une fois votre programme créé, il fonctionnera automatiquement sous Windows, Mac, Linux, etc.
- On peut faire de nombreuses sortes de programmes avec Java :
  - des applications, sous forme de fenêtre ou de console ;
  - des applets, qui sont des programmes Java incorporés à des pages web ;
  - des applications pour appareils mobiles, avec J2ME ;
  - et bien d'autres ! J2EE, JMF, J3D pour la 3D...

Comme vous le voyez, Java permet de réaliser une très grande quantité d'applications différentes ! Mais... comment apprendre un langage si vaste qui offre autant de possibilités ?

Heureusement, ce cours est là pour tout vous apprendre de Java à partir de zéro .

Exemples de programmes réalisés en Java

Ce cours vous plaît ?
Si vous avez aimé ce cours, vous pouvez retrouver le livre "*Apprenez à programmer en Java*" du même auteur, en vente sur le Site du Zéro, en librairie et dans les boutiques en ligne. Vous y trouverez ce cours adapté au format papier avec une série de chapitres inédits.

Plus d'informations

N.B. : je tiens à faire une dédicace spéciale à ptipilou, zCorrecteur émérite, sans qui ce tuto n'aurait pas vu le jour !

Un grand merci pour ton travail et ton soutien !

Retour en haut
Retour en haut

Partie 1 : Bien commencer en Java

Bon, vous ne connaissez rien à Java ? Eh bien c'est par ici que ça se passe ! Java est normalement un langage fait pour développer des applications graphiques, mais pour arriver à cela, nous devons tout de même passer par la programmation Java en mode console.

Donc, dans cette première partie, nous allons voir les bases du langage, ainsi que leur fidèle compagnon Eclipse.
Retour en haut

Parlons d'Eclipse

Pour ceux qui l'avaient deviné, Eclipse est le petit logiciel qui va nous permettre de développer nos applications, ou nos applets, et aussi celui qui va compiler tout ça.
Eh oui ! Java est un langage compilé. Je ne l'avais pas dit, je crois...

Donc, notre logiciel va permettre de traduire nos futurs programmes Java en langage compilé. Mais celui-ci ne peut pas être compris par l'ordinateur. Ce code compilé s'appelle du **Byte Code**. Il n'est compréhensible que par un environnement Java, vulgairement appelé JRE (*Java Runtime Environment*) disponible sur le site de Sun MicroSystems.

Préambule

Avant toute chose, quelques mots sur le projet Eclipse.

Eclipse IDE est un environnement de développement libre permettant potentiellement de créer des projets de développement mettant en œuvre n'importe quel langage de programmation (C++, PHP...). Eclipse IDE est principalement écrit en Java.

La spécificité d'Eclipse IDE vient du fait que son architecture est totalement développée autour de la notion de **plug-in**. Cela signifie que toutes les fonctionnalités de celui-ci sont développées en tant que plug-in. Pour faire court, si vous voulez ajouter des fonctionnalités à Éclipse, vous devez :

- télécharger le plug-in correspondant,
- copier les fichiers spécifiés dans les répertoires spécifiés,
- démarrer Eclipse, et ça y est !

Lorsque vous téléchargez un nouveau plug-in pour Éclipse, celui-ci se présente souvent comme un dossier contenant généralement deux sous-dossiers. Un dossier « *plugins* » et un dossier « *features* ». Ces dossiers existent aussi dans le répertoire d'Éclipse. Il vous faut donc copier le contenu des dossiers de votre plug-in dans le dossier correspondant dans Éclipse (*plugins* dans plugins, et *features* dans features).

C'est pour toutes ces raisons que j'ai choisi Eclipse comme outil de développement ; de plus, vous verrez qu'il est relativement simple d'utilisation.

Maintenant que ces quelques mots ont été dits, je vous invite donc à passer à l'étape suivante.
Retour en haut

Téléchargement

Avant de vous lancer dans le téléchargement d'**Éclipse**, vous devez avant tout vous assurer d'avoir **un environnement Java, ou JRE sur votre machine**.

Un JRE (ou **J**ava **R**untime **E**nvironment) va vous servir à lire les programmes qui ont été codés en Java. Comme je vous l'ai dit plus tôt, Eclipse est codé en Java : donc, pour utiliser Eclipse, il vous faut un **JRE**.

Rendez-vous donc sur la page de téléchargement des JRE sur le site de SUN Microsystem (fondateur du langage).
Choisissez la dernière version du JRE.

Après avoir cliqué sur "**Download**", vous arrivez sur une nouvelle page, où vous devez choisir votre système d'exploitation et cocher le bouton "**Accept License Agreement**".

Sélectionnez votre système d'exploitation (ici, j'ai mis *Windows*) et n'oubliez pas de cocher la case : "**I agree to the Java SE Development Kit 6 License Agreement**" afin d'accepter la licence.

Euh... ça veut dire quoi, ***JSE*** ?

Alors on va faire simple. Je vous ai dit plus tôt que Java permet de développer différents types d' applications. Eh bien il faut des outils différents pour les différents types d'applications.

- ***J2SE ou Java 2 Standard Édition*** : permet de développer des applications dites "client", par exemple... euh... Éclipse est une application "client". C'est ce que nous allons faire dans ce tutoriel.
- ***J2EE ou Java 2 Enterprise Édition*** : permet de développer des applications web en Java.
- ***J2ME ou Java 2 Micro Édition*** : permet de développer des applications pour appareil portable, comme des téléphones portables, des PDA...

Voilà, fin de l'aparté...

Vous pouvez maintenant télécharger et installer votre JRE. Ceci fait, je vous invite donc à télécharger Éclipse IDE sur cette page en choisissant **"Éclipse IDE for java developers"**, en choisissant la version d'Eclipse correspondant à votre OS.

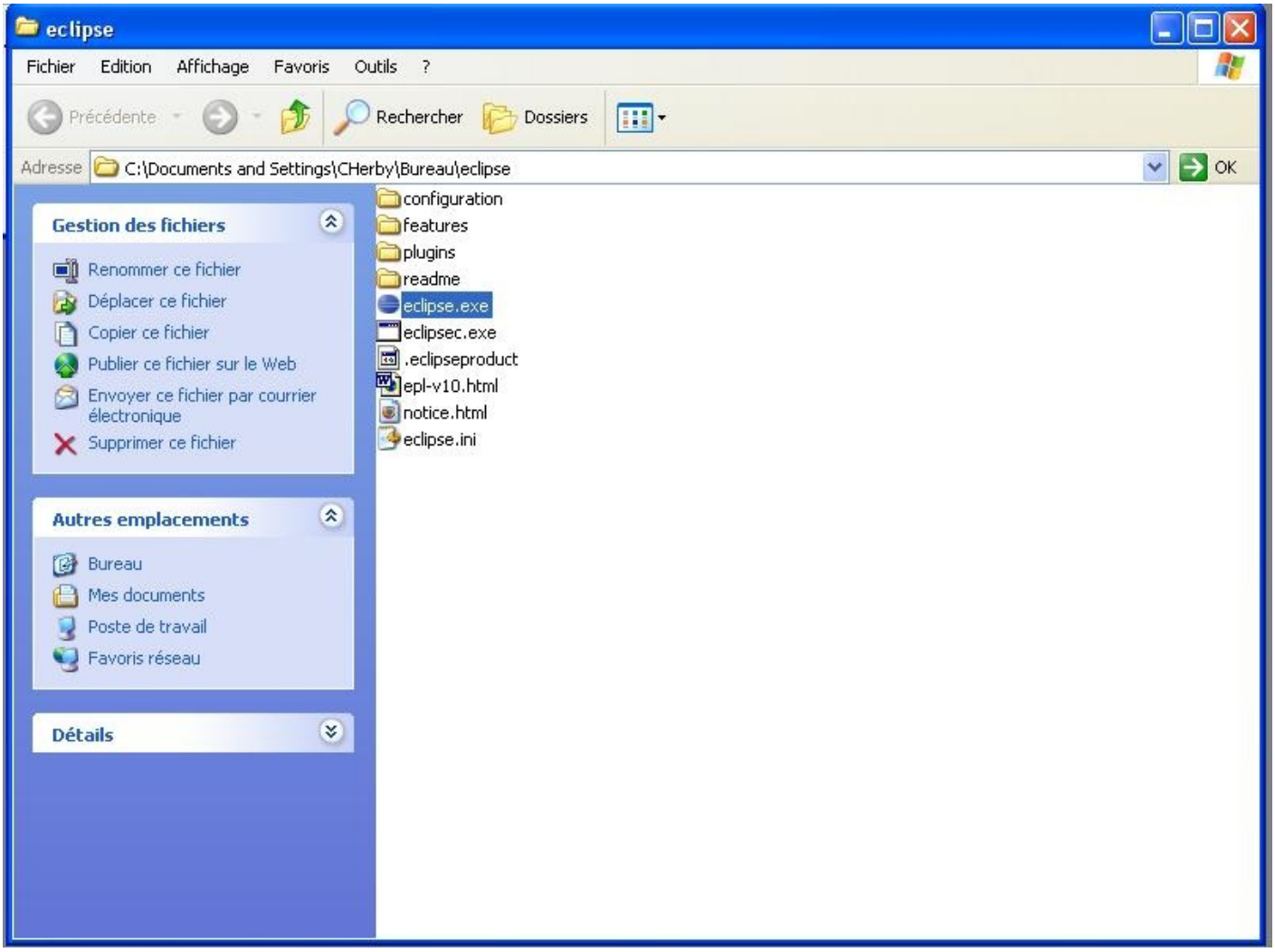
Sélectionnez maintenant le miroir que vous souhaitez utiliser pour obtenir Éclipse. Voilà, vous n'avez plus qu'à attendre la fin du téléchargement.

Retour en haut

Installation

Maintenant que vous avez un JRE et que vous avez téléchargé Éclipse, nous allons voir comment nous servir de celui-ci.

Vous devez maintenant avoir une archive contenant Éclipse. Décompressez-la où vous voulez, puis, entrez dans ce dossier. Si comme moi vous êtes sous Windows, vous devriez avoir ceci :



Ne travaillant pas sous Mac OS ni sous Linux, je ne peux malheureusement pas vous proposer de *screenshot* pour ces OS, mais je pense que ça doit fortement y ressembler...

La suite sera donc pour Windows, mais cela ne doit pas être très différent sur les autres OS.

Ensuite, si vous le souhaitez, vous pouvez créer un raccourci de l'exécutable "**eclipse.exe**" pour pouvoir y accéder plus facilement. Ceci fait, lancez Eclipse.

Peu de temps après, vous devriez avoir une fenêtre comme celle-ci :

Ici, Eclipse vous demande dans quel dossier vous souhaitez enregistrer vos projets. Libre à vous de choisir celui-ci. J'ai coché la *checkbox* pour qu'Éclipse se souvienne de ma décision. Mais lorsque vous créerez des projets, rien ne vous empêche de spécifier un autre dossier que celui de votre **"*workspace*"**.

Une fois cette étape effectuée, vous arrivez sur la page d'accueil d'Éclipse. Je n'ai jamais trop regardé ce que celle-ci propose ; donc, si vous avez envie de jeter un coup d'oeil, allez-y.

Retour en haut

Présentation rapide de l'interface

Je vais maintenant vous faire faire un tour rapide de l'interface que vous propose Eclipse, en gros, des éléments dont nous allons nous servir dans ce tutoriel.

Je ne connais malheureusement pas toutes les fonctionnalités d'Eclipse...
Par conséquent, je peux très bien omettre certains points qui peuvent être importants pour d'autres développeurs.
Merci de me tenir au courant le cas échéant.

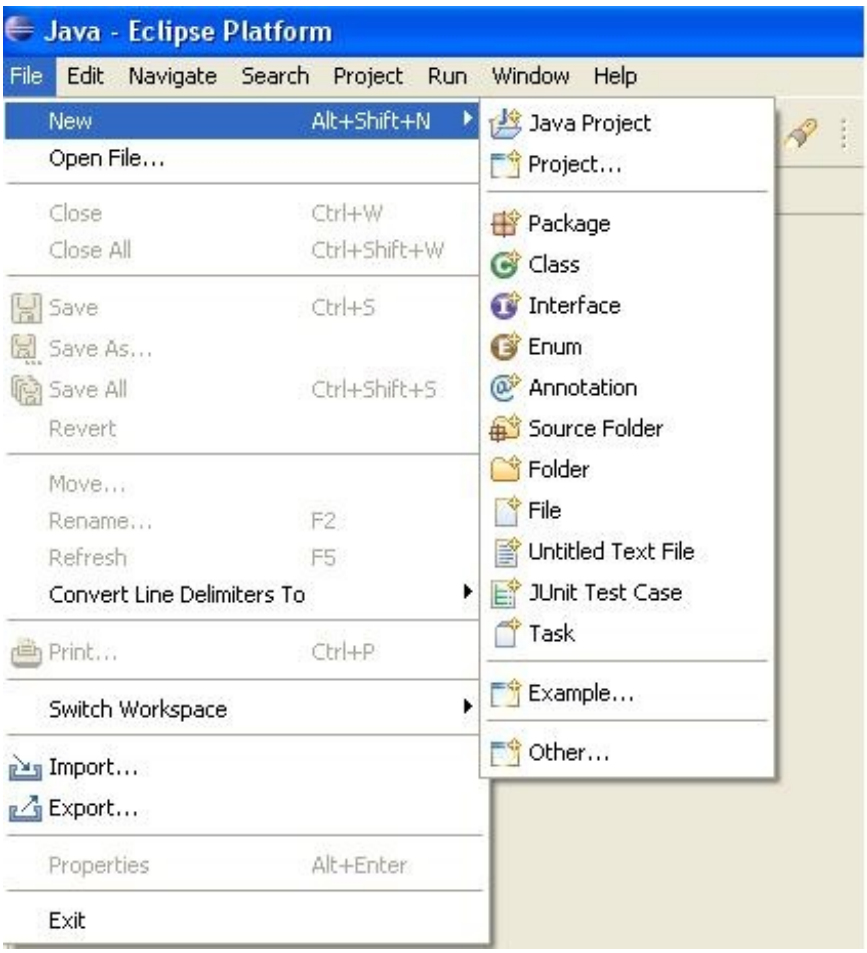
Avant de commencer, regardez bien les raccourcis clavier présents dans les menus... Ils sont très utiles et peuvent vous faire gagner beaucoup de temps !

On est obligés ?

Bien sûr que non, mais croyez-moi, quand on y a goûté, on ne peut plus s'en passer...

Allez, trêve de bavardages, on est partis.

Menu "File"

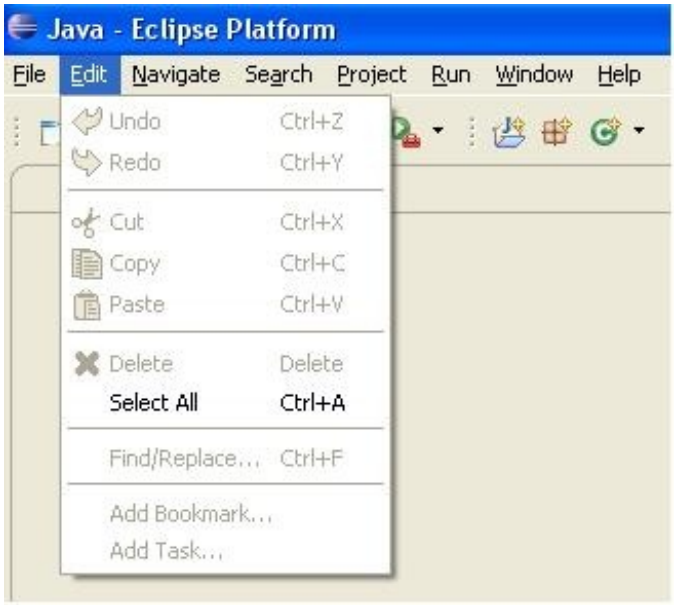


C'est ici que nous pourrons créer de nouveaux projets Java, enregistrer ceux-ci, les exporter le cas échéant...
Ici, les raccourcis à retenir sont :

- **ALT + SHIFT + N** : Nouveau projet
- **CTRL + S** : enregistrer la ***classe*** * Java où on est positionnés
- **CTRL + SHIFT + S** : tout sauvegarder
- **CTRL + W** : fermer la classe Java où on est positionnés
- **CTRL + SHIFT + W** : fermer toutes les classes Java ouvertes.

**** classe : on y reviendra.***

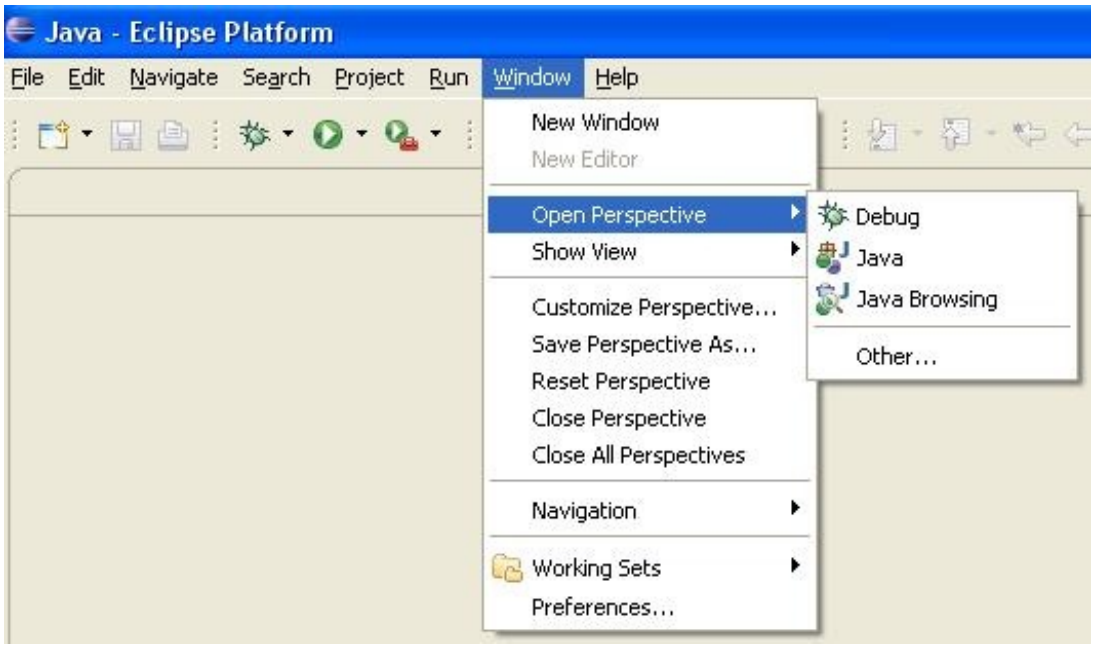
Menu "Edit"



Dans ce menu, nous pourrons utiliser les commandes "copier", "coller"...
Ici, les raccourcis à retenir sont :

- **CTRL + C** : copier la sélection
- **CTRL + X** : couper la sélection
- **CTRL + V** : coller la sélection
- **CTRL + A** : tout sélectionner
- **CTRL + F** : chercher / remplacer.

Menu "Window"



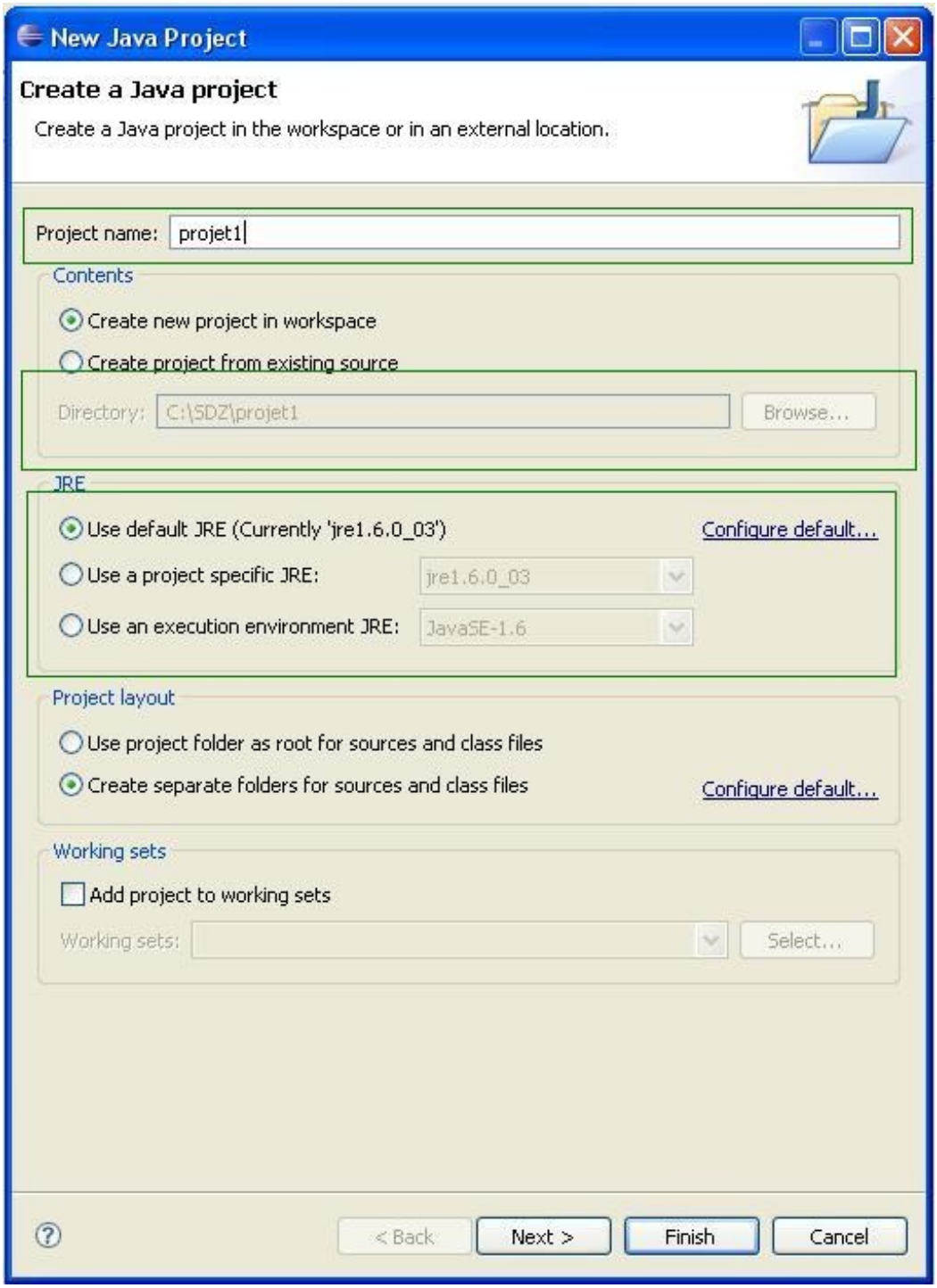
Dans celui-ci, nous pourrons configurer Eclipse selon nos besoins.

La barre de navigation

Nous avons dans l'ordre :

- 1 : "nouveau" général. Cliquer sur ce bouton revient à faire "Fichier - Nouveau"
- 2 : enregistrer. Revient à faire CTRL + S.
- 3 : imprimer.
- 4 : exécuter la classe ou le projet spécifié. Nous verrons ceci plus en détail.
- 5 : créer un nouveau projet Java. Revient à faire "Fichier - Nouveau - Java project".
- 6 : créer une nouvelle classe dans un projet. Revient à faire "Fichier - Nouveau - Classe".

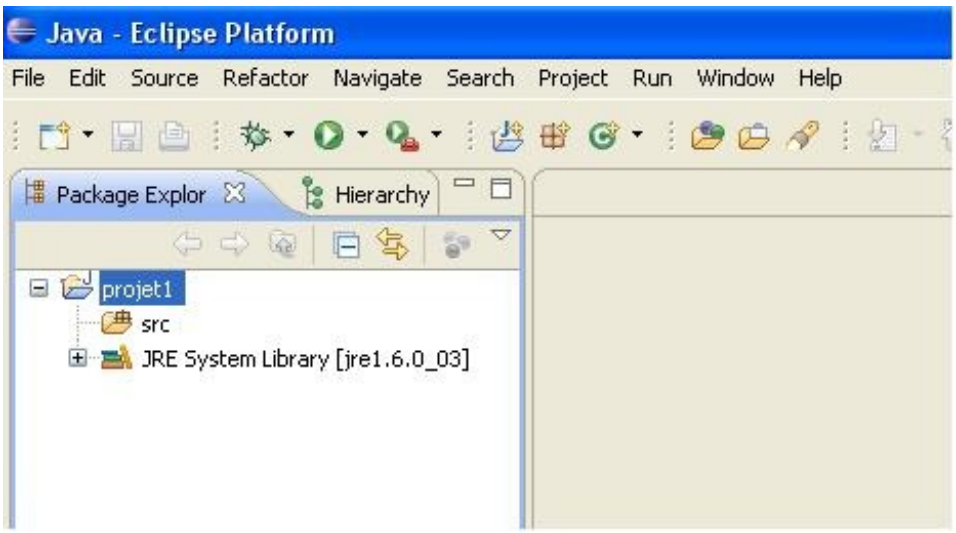
Je vous demande maintenant de créer un nouveau projet Java. Vous devriez arriver à cette fenêtre sans trop de difficultés :



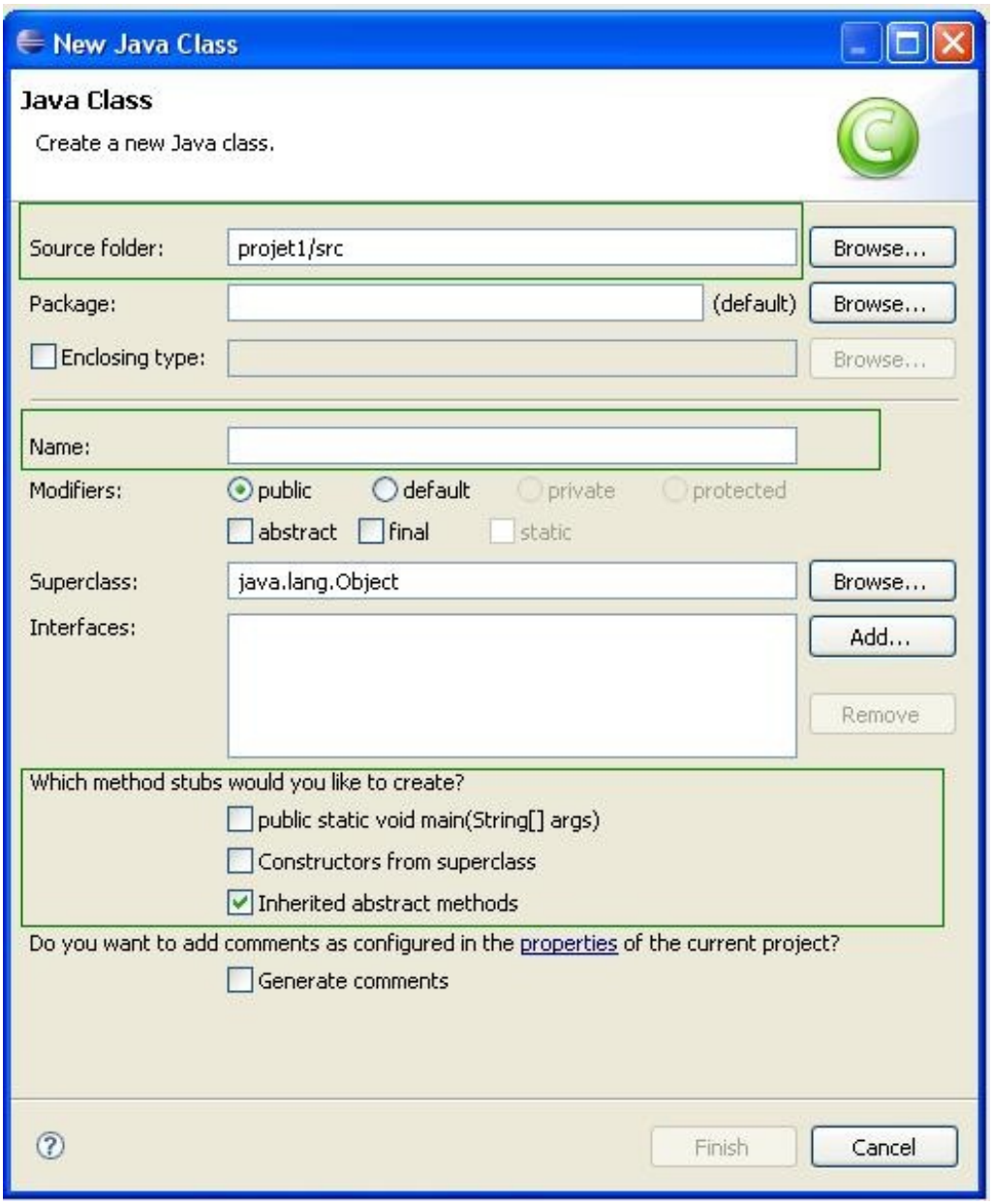
Renseignez le nom de votre projet comme je l'ai fait plus haut (encadré 1). Vous pouvez aussi voir où sera enregistré ce projet (encadré 2).
Un peu plus compliqué maintenant, vous avez donc un environnement Java sur votre machine, mais dans le cas où vous en auriez plusieurs, vous pouvez aussi spécifier à Eclipse quel JRE utiliser pour ce projet.

En fait, vous pourrez changer ceci à tout moment dans Eclipse en allant dans Window / Préférences, en dépliant l'arbre "Java" dans la fenêtre et en choisissant "*Installed JRE*".
Par contre, n'utilisez ça que si vous êtes sûrs de vous !

Vous devriez avoir un nouveau projet dans la fenêtre de gauche, comme ceci :

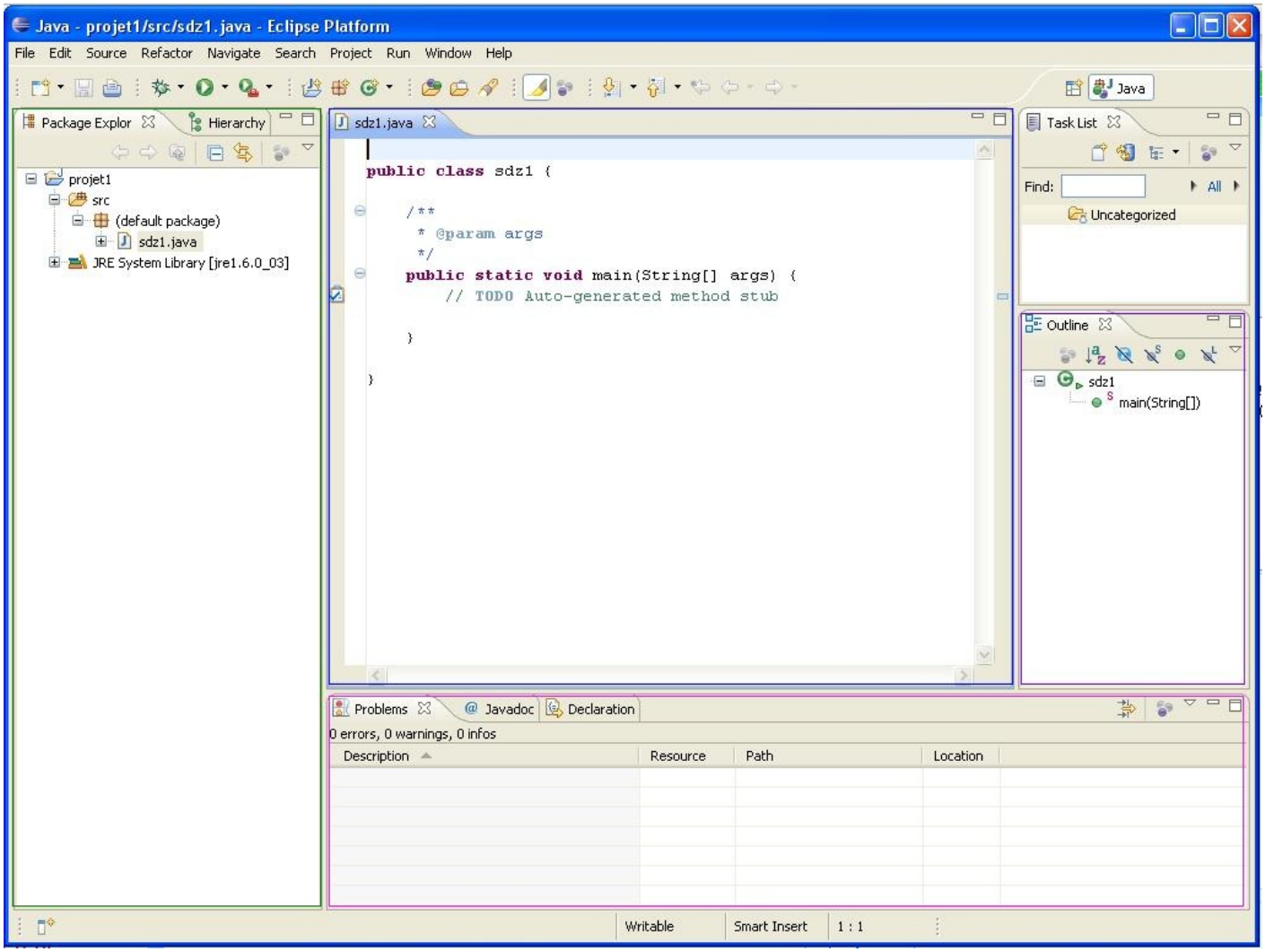


Pour boucler la boucle, ajoutons dès maintenant une nouvelle classe dans ce projet. Je suis sûr que vous allez y arriver sans moi...
Vous êtes donc devant cette fenêtre :



Dans l'encadré 1, nous pouvons voir où seront enregistrés nos fichiers Java.
Dans l'encadré 2, nommez votre classe Java, moi j'ai **sdz1**.
Dans l'encadré 3, Eclipse vous demande si cette classe a un truc particulier. Eh bien oui ! Cochez "***public static void main(String[] args)***", puis cliquez sur "***finish***" (nous allons revenir sur ce dernier point dans la partie suivante).

Une fois ces opérations terminées, vous êtes devant ceci :



Alors avant de commencer à coder, nous allons un peu explorer notre espace de travail.

Dans l'encadré vert (à gauche), vous trouverez le dossier de votre projet ainsi que son contenu. Ici, vous pourrez gérer votre projet comme bon vous semble (ajout, suppression...).

Dans l'encadré bleu (au centre), je pense que vous avez deviné... C'est ici que nous allons écrire nos codes sources.

Dans l'encadré rose (en bas), c'est là que vous verrez apparaître le contenu de vos programmes ainsi que les erreurs éventuelles !

Et enfin pour finir, **dans l'encadré violet (à droite)**, dès lors que nous aurons appris à coder nos propres fonctions ainsi que des objets, c'est ici que la liste des méthodes et des variables sera affiché.

Maintenant que vous avez Eclipse bien en main, nous allons faire un tour rapide des autres IDE de développement pour Java.
Retour en haut

D'autres outils à la hauteur

Surtout gardez en mémoire qu'un IDE est un outil de développement.
Comment ça, je ne vous l'avais pas dit...

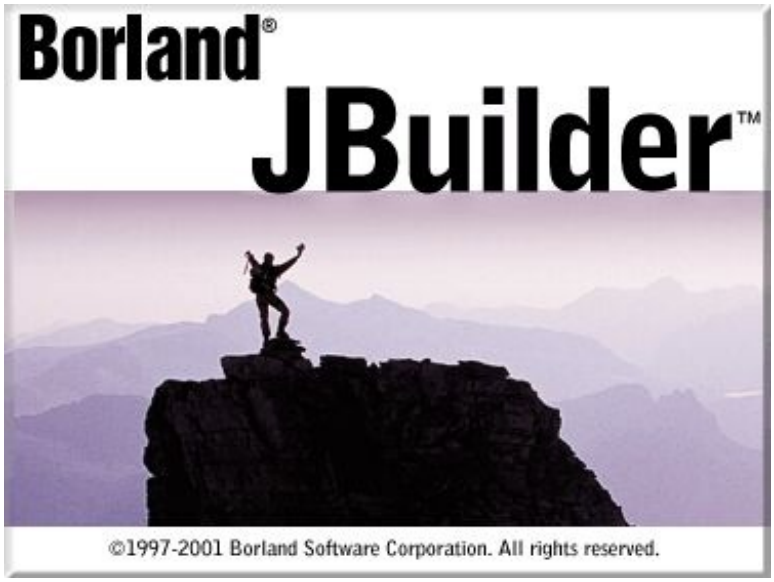
Ce qu'il faut garder en tête, c'est qu'un IDE, comme n'importe quel outil, est :

- fonctionnel,
- adapté,
- évolutif.

Ce que j'entends par là, c'est que comme un menuisier, vous utiliserez des outils pour travailler et comme lui, vous choisirez celui qui vous convient le mieux. Partez du principe que les IDE ont grosso-modo les mêmes fonctionnalités, compiler en ***byte code***, exécuter, debugger...

Les IDE de la liste suivante sont tout aussi adaptés qu'Eclipse pour développer en Java. A vous de vous faire un avis. Testez, comparez et choisissez.

1. JBuilder



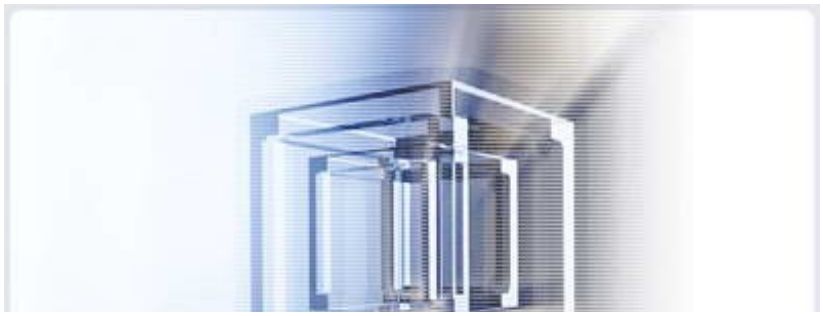
Celui-ci est idéal pour le développement à grande échelle, pour les projets d'entreprise, etc.
Il intègre tout un tas de technologies comme **XML**, **jsp/servlet**, concept d'ingénierie, outils **UML**...

C'est simple : je crois même qu'il fait du café...
Tout ceci fait de lui un outil puissant, mais très lourd pour les configurations moyennes.

En gros, vous serez peut-être amenés à l'utiliser, mais en entreprise...

Bon : pour les curieux de nature, vous pouvez faire un tour ici.
Ne cherchez pas de version gratuite, JBuilder est payant. Mais je crois que vous pouvez avoir une version d'évaluation... A vérifier...

2. NetBeans



NetBeans est l'IDE créé par Sun Microsystems, il contient donc toutes les fonctionnalités d'un bon IDE :

- un debugger,
- un outil de modélisation UML,
- tout le nécessaire pour faire des applications J2EE,
- il est **GRATUIT !**

- ...

De plus, il semblerait qu'on puisse *customiser* l'environnement de celui-ci.

Si vous voulez vous faire une opinion, ce que je vous conseille vivement, vous pouvez le télécharger ici.

Il existe bien d'autres IDE pour développer en Java, mais ceux-ci (Eclipse compris) représentent la majorité des IDE utilisés. En voici d'autres, pour information :

- JCreator
- IntelliJ
- Sun ONE Studio
- JDeveloper
- JEdit
- Le Bloc-notes de Windows (***vi*** sous Linux), ou encore Notepad++.

Qu... Quoi ? On peut faire des programmes Java avec le Bloc-notes de Windows ?

Tout à fait...
En fait, lorsque vous créerez un programme Java dans votre IDE préféré et que vous l'exécuterez, celui-ci va pré-compiler votre code Java en ***byte code*** pour qu'ensuite votre **JVM** (***J**ava **V**irtual **M**achine, cf chapitre suivant*) l'interprète et l'exécute.

Mais toutes ces étapes peuvent être faites à la main, en ligne de commande, mais nous n'en parlerons pas maintenant.

Bon, je crois qu'il est temps pour un petit QCM...

Retour en haut

Voilà ! Une partie pénible de terminée !
Par contre, Eclipse regorge de fonctionnalités. Je n'en ai pas parlé pour le moment car je pense que vous comprendrez mieux le moment venu, avec un exemple concret d'utilisation.

J'espère que vous avez encore de l'énergie, car, dans le chapitre suivant, nous aborderons quelque chose de bien plus passionnant...
Bon ! Tenez-vous prêts pour faire vos premiers codes Java ! !

Retour en haut

Votre premier programme

Dans ce chapitre, nous allons faire nos premiers programmes en Java.
Mais tout d'abord, quelques explications sur le fonctionnement du JRE s'imposent.
Alors... go !

Mieux connaitre son environnement Java

Bon : cette partie s'annonce riche en informations et pour certains, elle sera même "pompeuse"... Mais afin de bien coder en Java, il vaut mieux comprendre comment tout cela fonctionne.

Dans le JRE, que je vous ai fait télécharger dans la partie précédente, se trouve le coeur même de Java. Celui-ci s'appelle la **JVM (pour Java Virtual Machine)** que j'appellerai dorénavant **machine virtuelle**. Celle-ci a pour rôle, comme je vous l'ai dit dans le chapitre précédent, d'exécuter les programmes Java sur votre machine, c'est ce qui fait que les programmes Java sont dit **"portables"**.

Qu'est-ce tu entends par là ?

J'allais y venir...

Comme je vous l'ai maintes fois répété, les programmes Java sont, avant d'être utilisés par la machine virtuelle, pré-compilés en byte code (par votre IDE ou encore à la main). Ce byte code n'est compréhensible que par une JVM, et c'est celle-ci qui va faire le lien entre ce code et votre machine.

Vous aviez sûrement remarqué que sur la page de téléchargement du JRE, plusieurs liens étaient disponibles :

- un lien pour Windows,
- un lien pour Mac,
- un lien pour Linux.

Ceci car la machine virtuelle Java se présente différemment selon qu'on se trouve sous Mac, sous Linux ou encore sous Windows. Par contre le byte code, lui, reste le même quelque soit l'environnement où a été développé et pré-compilé votre programme Java.

Conséquence directe :

quel que soit l'OS sous lequel a été codé un programme Java, n'importe quelle machine pourra l'exécuter si elle dispose d'une JVM !

Tu n'arrêtes pas de nous rabâcher byte code par-ci... byte code par-là...
Mais c'est quoi, au juste ?

Eh bien un byte code - et je dis bien UN byte code - n'est autre qu'un code intermédiaire entre votre code Java et le code machine.
Ce code particulier se trouve dans les fichiers pré-compilés de vos programmes ; en Java, un fichier source a l'extension **.java** et un fichier pré-compilé a l'extension **.class** ; c'est dans ce dernier que vous pourrez voir du byte code.

Par contre, vos fichiers **.java** sont de bêtes fichiers texte, seule l'extension est changée... Vous pouvez donc les ouvrir, les créer ou encore les mettre à jour avec... le Bloc-notes de Windows par exemple...

Pour en revenir à notre byte code, je vous invite à regarder un fichier **.class** à la fin de cette partie (vous en aurez au moins un), mais je vous préviens, c'est illisible ! !

Inutile de vous dire que votre machine est incapable d'interpréter du byte code !

Alors pourquoi je vous ai dit UN byte code ? Tout simplement parce que la machine virtuelle de Java se moque que le byte code soit à la base du Java ou autre chose.
Son rôle est d'interpréter ce code et de le faire fonctionner. Donc, si un jour vous arrivez à faire du byte code avec comme fichier source un **.cpp** (pour C++), ça **devrait** ***(et j'insiste sur le devrait)*** **fonctionner**.

Vous pouvez avoir un aperçu de ce qui devrait fonctionner sur cette page.

Ce qu'il faut retenir :

- la JVM est le coeur de Java,

- celle-ci fait fonctionner vos programmes Java, pré-compilés en byte code,
- les fichiers contenant le code source de vos programmes Java ont l'extension **.java**,
- les fichiers pré-compilés correspondant à vos codes sources Java ont l'extension **.class**,
- le byte code est un code intermédiaire entre votre programme et votre machine,
- un programme Java, codé sous Windows peut être pré-compilé sous Mac et enfin exécuté sous Linux,
- votre machine NE PEUT PAS interpréter du byte code.

Je pense qu' après cette partie purement théorique, vous avez hâte de commencer à taper des lignes de code... Mais avant, il me reste quelques choses à vous dire !

Retour en haut

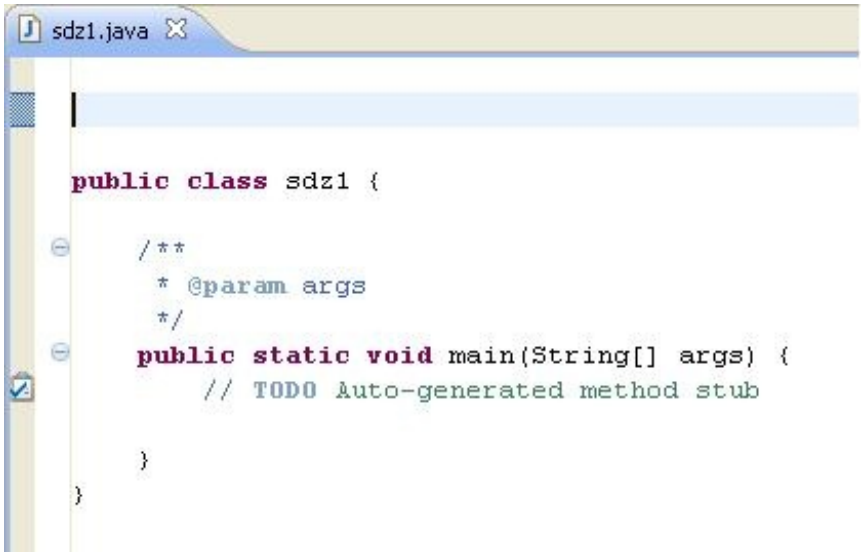
Avant de commencer

Avant de commencer, vous devez savoir que tous les programmes Java sont composés d'au moins une classe.
Cette classe doit contenir (applet mise à part) une méthode main. Celle-ci est la méthode principale du programme, c'est elle que la JVM va chercher afin de lancer votre programme. Pour faire court, c'est le point de départ du programme.
Vous voyez donc son importance !
Et pour qu'on se comprenne bien, une méthode est une suite d'instructions à exécuter. Une méthode contient :

- ***une entête :*** celle-ci va être un peu comme la carte d'identité de la méthode. Nous y reviendrons.
- ***un corps :*** le contenu de la méthode. Ce qui va être fait ! Délimité par des accolades {}.
- ***une valeur de retour :*** le résultat que la méthode va retourner. **SAUF** pour les méthodes de type **`void`** qui ne renvoient rien. Nous y reviendrons aussi.

Vous verrez, un peu plus tard, qu'un programme n'est qu'une multitude de classes qui s'utilisent l'une l'autre.
Mais pour le moment, nous n'allons travailler qu'avec une seule classe.

Dans la partie précédente, je vous avais demandé de créer un projet Java ; ouvrez-le si ce n'est pas déjà fait. Donc pour rappel, voici ce que vous aviez sous votre nez :



Vous voyez la fameuse classe dont je vous parlais ? Ici, elle s'appelle **`sdz1`**.
Vous pouvez voir que le mot **`class`** est précédé du mot **`public`**, nous verrons la signification de celui-ci lorsque nous programmerons des objets.

Pour le moment, ce que vous devez retenir c'est que votre classe est définie par un mot clé, **`class`**, qu'elle a un nom ; ici, **`sdz1`** et que le contenu de celle-ci est délimité par des accolades, {}.

Comme je vous l'ai dit plus haut, notre unique classe contient la méthode **main**. La syntaxe de cette méthode est toujours la même :

Code : Java

```
public static void main(String[] args){
// Contenu de votre classe
}
```

Ce sera entre les accolades de la méthode main que nous écrirons nos codes sources.

Excuse-nous, mais... Pourquoi as-tu écrit "//Contenu de votre classe" et non "Contenu de votre classe" ?

Bonne question !
Je vous ai dit plus haut que votre programme Java, avant de pouvoir être exécuté, doit être pré-compilé en byte code. Eh bien la possibilité de forcer le compilateur à ignorer certaines instructions existe !
On appelle ça des **commentaires** et il existe deux syntaxes :

- ***les commentaires unilignes :*** introduits par les symboles //, ils mettent tous ce qui les suit en commentaires, du moment que le texte se trouve sur la même ligne que les //.

Code : Java

```
public static void main(String[] args){
//Un commentaire
//un autre
//Encore un autre
Ceci n'est pas un commentaire ! ! ! !
}
```

- les commentaires multilignes : ils sont introduits par les symboles /* et se terminent par les symboles */.

Code : Java

```
public static void main(String[] args){

/*
Un commentaire
Un autre
Encore un autre
*/
Ceci n'est pas un commentaire ! !
}
```

D'accord ! Mais ça sert à quoi ?

C'est simple : au début, vous ne ferez que de très petits programmes.
Mais dès que vous aurez pris de la bouteille, la taille de ceux-ci, ainsi que le nombre de classes qui les composera, augmentera. Vous serez contents de trouver quelques lignes de commentaires au début de votre classe pour vous dire à quoi elle sert, ou encore des commentaires dans une méthode qui fait des choses compliquées afin de savoir ou vous en êtes dans vos traitements...

Il existe en fait *une troisième syntaxe*, mais celle-ci a un but particulier. Elle permettra de générer une documentation de votre programme ! Une Javadoc (Java Documentation).
Je n'en parlerai que très peu, et pas dans ce chapitre. Nous verrons cela lorsque nous programmerons des objets mais, pour les curieux, je vous conseille le très bon tutoriel de dworkin.

Je profite de cet aparté sur les commentaires pour vous donner une astuce bien pratique !
Il vous arrivera forcément à un moment donné de vouloir mettre une partie de code en commentaire. Dans ce cas, il y a 99,99999999 % de chance que vous choisissiez les commentaires multilignes.
L'inconvénient, c'est que lorsque vous voudrez remettre votre morceau de code en activité, il vous faudra retirer les `/* */`.
Mais si 5 minutes plus tard, vous voulez remettre la même portion de code en commentaire .

Regardez ce morceau de code (il ne fait rien du tout, c'est un exemple) :

Code : Java

```
public static void main(String[] args){
```

```
        String str = "toto";
        int entier = 0;

        if(entier != 0){

       /*
       for(int i = 0; i < 20; i++){
       System.out.println("oui ! ! ! ! !");
       }
       //*/

        }
       }
```

Vous pouvez voir que j'ai utilisé les commentaires multilignes ! Mais avec une variante...
La ligne qui ferme les commentaires est précédée... de //. Mais c'est ce qui met une ligne en commentaires...

C'est idiot de mettre ça là... Ils sont en commentaire !

Je le sais bien, et c'est là qu'est l'astuce. Pour le moment, les deux // sont en commentaires comme les lignes qui se trouvent entre /* et */.
Mais que se passe-t-il si j'ajoute un / devant ma première instruction de commentaire ?

Code : Java

```
      public static void main(String[] args){

        String str = "toto";
        int entier = 0;

        if(entier != 0){

       //*
         for(int i = 0; i < 20; i++){
           System.out.println("oui ! ! ! ! !");
         }
       //*/

        }
       }
```

Eh bien là, ce sont mes commentaires multilignes qui sont devenus des commentaires, et mes lignes sont de nouveau actives dans mon code !

Explique-nous ça !

C'est très simple. Le fait d'ajouter un / devant /* met l'étoile en commentaire... Dans ce cas, il ne s'agit plus d'un commentaire multilignes, mais uniligne ! Et là je crois que vous avez deviné l'utilité de cette ligne //*/... Sur celle-ci, c'est l'instruction de commentaire multiligne fermante qui est en commentaire !
Donc, plus besoin d'ajouter de commentaire, d'en effacer, de les remettre, de les ré-effacer...
Vous encadrez la portion de code que vous souhaitez enlever de /* .... //*/ et lorsque vous la voulez à nouveau, vous ajoutez un / devant l'instruction ouvrante des commentaires ! Et si vous voulez remettre la même portion de code en commentaire, enlevez le / ajouté devant /* !

A partir de maintenant et jusqu'à ce que nous programmions des interfaces graphiques, nous allons faire ce qu'on appelle des **programmes procéduraux**. Cela signifie que le programme se déroulera de façon procédurale.

Euh... késako ?

En fait, un programme procédural est un programme qui s'effectue de **haut en bas**, **une ligne après l'autre.**
Bien sûr, il y a des instructions qui permettent de répéter des morceaux de code, mais le programme en lui même se terminera une fois arrivé à la fin du code.
Ceci vient en opposition à la programmation événementielle (ou graphique) qui elle, est basée sur des événements (clic de souris, choix dans un menu...).

Je pense que j'ai fait le tour de ce que je voulais vous dire...
Bon : pour votre premier programme, je pense que le traditionnel "Hello World ! ! " est de mise...
Donc, allons-y !
Retour en haut

Votre premier programme

Nous entrons enfin dans le vif du sujet !
Mais ne vous attendez pas à faire un vrai programme tout de suite...

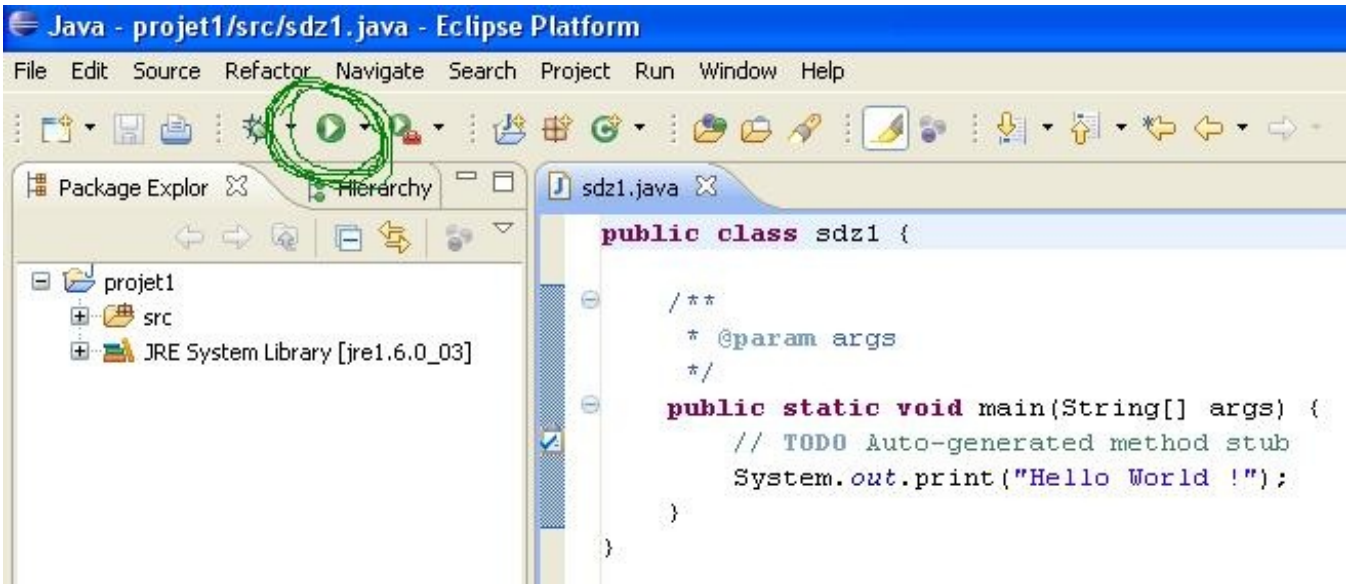
Maintenant, vous pouvez taper les lignes suivantes entre les accolades de votre méthode main :

Code : Java

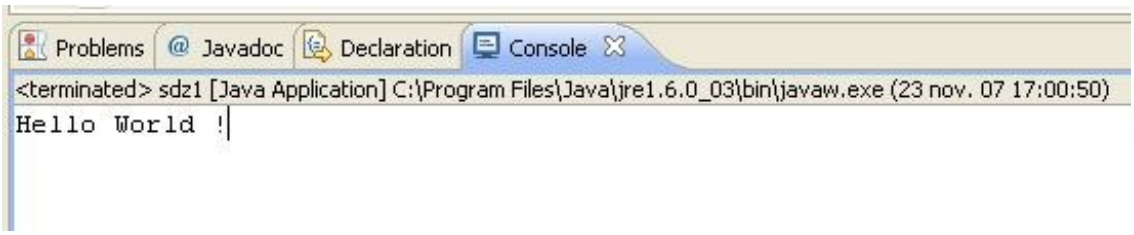
```
System.out.print("Hello World !");
```

N'oubliez surtout pas le **;** à la fin de la ligne !
Toutes les instructions en Java sont suivies d'un point virgule.

Une fois que vous avez saisi cette ligne de code dans votre méthode main, vous devez lancer le programme.
Si vous vous souvenez bien de la présentation du chapitre précédent, vous avez dû cliquer sur la flèche blanche dans un rond vert :



Si vous regardez dans votre console, fenêtre en bas sous Eclipse, vous devez avoir :



Expliquons un peu cette ligne de code.
Littéralement, elle signifie "la méthode **`print()`** va écrire *Hello World !* en utilisant l'objet **`out`** de la classe **`System`**".

- **`System`** : ceci correspond à l'appel d'une classe qui s'appelle "`System`". C'est une classe utilitaire qui permet surtout d'utiliser l'entrée et la sortie standard.
- **`out`** : objet de la classe **System** qui gère la sortie standard.
- **`print`** : méthode qui écrit dans la console la chaîne passée en paramètre.

Si vous mettez plusieurs `System.out.print`, voici ce qui ce passe.
Prenons ce code :

Code : Java

```
System.out.print("Hello World ! !");
  System.out.print("My name is");
  System.out.print("Cysboy");
```

Lorsque vous l'exécutez, vous devriez avoir une chaîne de caractères qui se suivent, sans saut à la ligne. En gros, vous devez avoir ceci dans votre console :
Hello World ! !My name is Cysboy

Je me doute que vous souhaiteriez faire un retour à la ligne pour que votre texte soit plus lisible... Pour réussir cela, vous avez plusieurs solutions.

- Soit vous utilisez un caractère d'échappement, ici, **`\n`**.
- Soit vous utilisez la méthode **`println()`** à la place de la méthode **`print()`**.

Donc, si nous reprenons notre précédent code et que nous appliquons ceci, voilà ce que ça donnerait :
(notre code modifié) :

Code : Java

```
System.out.print("Hello World ! ! \n");
  System.out.println("My name is");
  System.out.println("\nCysboy");
```

Le résultat :
Hello World ! !
My name is

Cysboy

Vous pouvez voir que :

- lorsque vous utilisez le caractère d'échappement **`\n`**, quelle que soit la méthode appelée, celle-ci intègre immédiatement un retour à la ligne à l'emplacement de ce dernier.
- lorsque vous utilisez la méthode **`println()`**, celle-ci ajoute automatiquement un retour à la ligne à la fin de la chaîne passée en paramètre.
- rien ne vous empêche de mettre un caractère d'échappement dans la méthode `println()`.

J'en profite au passage pour vous donner deux autres caractères d'échappement :

- **`\r`** : va mettre un retour chariot.
- **`\t`** : va faire une tabulation.

Vous avez sûrement remarqué que la chaîne de caractères que l'on affiche est entourée de " <chaine> ".
En Java, les double quotes sont des délimiteurs de chaînes de caractères ! Si vous voulez afficher un double quote avec la sortie standard, vous devrez "*l'échapper*" avec un \, ce qui donnerait
`System.out.println("coucou mon \"choux\" !");`

Maintenant que vous savez faire un "Hello World", je vous propose de voir la compilation de vos programmes en ligne de commande.

Cette sous-partie n'est ici que pour les plus curieux d'entre vous.

Vous pouvez passer outre cette sous-partie si vous le voulez, et aller directement au QCM mais, partez du principe que ça pourrait vous servir un jour...

Retour en haut

Compilation en ligne de commande (Windows)

Donc bienvenue aux plus curieux !

Avant de vous apprendre à compiler et à exécuter un programme en ligne de commandes, il va vous falloir le JDK de SUN (**J**ava SE **D**evelopment **K**it). C'est dans celui-ci que nous aurons de quoi compiler nos programmes. Le nécessaire à l'exécution des programmes est aussi dans le JRE... Mais vous l'aurez en plus dans le JDK.

Je vous invite donc à retourner sur le site de SUN et à télécharger celui-ci. Une fois cette opération effectuée, il est conseillé de mettre à jour votre variable d'environnement %PATH%.

Euh... Quoi ?

Votre **variable d'environnement**. C'est grâce à celle-ci que Windows arrive à trouver des exécutables sans lui spécifier le chemin d'accès complet. Vous, enfin Windows, en a plusieurs, mais nous nous intéresserons qu'à une seule.

En gros, cette variable contient le chemin d'accès à certains programmes.

Par exemple, si vous spécifiez le chemin d'accès d'un programme X dans votre variable d'environnement et que, comme par un malheureux hasard, vous n'avez plus aucun raccourci vers X, c'est simple : vous l'avez définitivement perdu dans les méandres de votre PC.
Eh bien vous pourrez le lancer en faisant "**démarrer > Exécuter**" et en tapant la commande "**`X.exe`**" (en partant du principe que le nom de l'exécutable est `X.exe`).

D'accord, mais comment on fait ? Et pourquoi on doit faire ça pour le JDK ?

J'y arrive...
Une fois votre JDK installé, ouvrez le répertoire **bin** de celui-ci, **mais également** celui-ci de votre JRE. Nous allons nous attarder sur deux fichiers.

Dans le répertoire **bin** de votre JRE, vous devez avoir un fichier appelé **java.exe**. Fichier que vous retrouvez aussi dans le répertoire **bin** de votre JDK. C'est grâce à ce fichier que votre ordinateur peut lancer vos programmes par le biais de la JVM.

Lorsque vous lancez une application sous Eclipse, ce fichier est lancé de façon implicite !

Le deuxième ne se trouve que dans le répertoire **bin** de votre JDK, il s'agit de **javac.exe** (**java c**ompiler). C'est celui-ci qui va pré-compiler vos programmes Java en byte code.

Alors pourquoi le faire pour le JDK ?
Eh bien, compiler-exécuter en ligne de commande revient à utiliser ces deux fichiers en leur précisant où se trouvent les fichiers à traiter. Cela veut dire que si on ne met pas à jour la variable d'environnement de Windows, nous devons :

- ouvrir l'invite de commande,
- se positionner dans le répertoire bin de notre JDK,
- appeler la commande souhaitée,
- préciser le chemin du fichier **.java**,
- renseigner le nom du fichier.

Avec notre variable d'environnement mise à jour, nous n'aurons plus qu'à :

- nous positionner dans le dossier de notre programme,
- appeler la commande,
- renseigner le nom du fichier Java.

Comme un exemple vaut mieux que de grands discours, mettons notre variable d'environnement à jour...

Allez dans le "**panneau de configuration**" de votre PC, de là, cliquez sur l'icône "**Système**" ; choisissez l'onglet "**avancé**" et vous devriez avoir un bouton, en bas, intitulé "**Variables d'environnement**" ; cliquez dessus.
Une nouvelle fenêtre s'ouvre. Dans la partie inférieure intitulée "**Variables système**" chercher la variable **Path**. Une fois sélectionnée, cliquez sur **Modifier**.

Encore une fois, une fenêtre, plus petite celle-là, s'ouvre devant vous. Elle contient le **nom** de la variable et sa **valeur**.

Ne changez pas son nom et n'effacez pas le contenu de valeur !
Nous allons juste **ajouter un chemin d'accès**.

Pour ce faire, allez jusqu'au bout de la valeur de la variable, ajoutez-y un point-virgule (**;**) s'il n'y en a pas, et ajoutez maintenant le chemin d'accès au répertoire **bin** de votre JDK, en terminant celui-ci par un **point-virgule** !.
Chez moi, ça donne ceci : "**C:\Sun\SDK\jdk\bin**".

Alors, ma variable d'environnement contenait avant ajout :

% SystemRoot% \system32;% SystemRoot% ;% SystemRoot% \System32\Wbem;

Et maintenant :

% SystemRoot% \system32;% SystemRoot% ;% SystemRoot% \System32\Wbem;C:\Sun\SDK\jdk\bin;

Validez les changements, et c'est tout !
Vous êtes maintenant prêts à compiler en ligne de commande.

Pour bien faire, allez dans le répertoire de votre premier programme et effacez le **.class**.
Ensuite, faites "**démarrer > Exécuter**" (ou encore touche Windows + r), et tapez "**cmd**".

Pour rappel, dans l'invite de commande, on se déplace de dossier en dossier grâce à l'instruction **cd**.
cd <nom du dossier enfant> : pour aller dans un dossier contenu dans celui dans lequel nous sommes.
cd .. : pour remonter d'un dossier dans la hiérarchie.

Par exemple : lorsque j'ouvre la console, je me trouve dans le dossier C:\toto\titi et mon application se trouve dans le dossier C:\sdz, je fais donc :

Code : Bash

```
cd ..
cd ..
cd sdz
```

Lors de la première instruction, je me retrouve dans le dossier C:\toto.
Lors de la deuxième instruction, je me retrouve à la racine de mon disque.
Lors de la troisième instruction, je me retrouve dans le dossier C:\sdz.
Nous sommes maintenant dans le dossier contenant notre fichier Java !

Mais nous pouvons condenser ceci en :

Code : Bash

```
cd ../../sdz
```

Maintenant, vous pouvez créer votre fichier **.class** en utilisant la commande **javac <nomDeFichier.java>**.
Si, dans mon dossier, j'ai un fichier test.java, je le compile en faisant :
javac test.java
Et si vous n'avez aucun message d'erreur, vous pouvez vérifier que le fichier test.class est présent en utilisant l'instruction **dir** qui liste le contenu d'un répertoire.

Cette étape réussie, vous pouvez lancer votre programme Java en faisant **java <nomFichierClassSansExtension>**
Ce qui nous donne :
java test
Et normalement, vous avez le résultat de votre programme Java qui s'affiche sous vos yeux ébahis !

Attention : il ne faut pas mettre l'extension du fichier pour le lancer, mais la mettre pour le compiler.

Donc voilà : vous avez compilé et exécuté un programme Java en ligne de commande... Vous en aurez peut être besoin un jour...
En tout cas, j'espère que vous êtes d'attaque pour un petit QCM...

Retour en haut

J'espère que maintenant vous arrivez mieux à cerner comment fonctionne Java, et à écrire ce que vous voulez à l'écran.
À présent, nous allons voir comment stocker des données en mémoire afin de les afficher, de les calculer...

En avant pour **les variables et les opérateurs**.

Retour en haut

Les variables et les opérateurs

Nous y voilà. Encore un chapitre barbant... Mais celui-là aussi est nécessaire, et je dirais même vital !
En Java, comme dans beaucoup de langages de programmation, avant d'utiliser une variable, nous devons d'abord définir ce qu'elle va contenir.

T'es bien gentil, mais c'est quoi, une *variable* ?

Oups ! Désolé ! Je parle, je parle et j'oublie l'essentiel... Une variable, c'est ce qui va nous permettre de stocker des informations de toute sorte (chiffres, résultats de calcul, des tableaux, des renseignements fournis par l'utilisateur...). Bref, vous ne pourrez pas faire de programme sans variables. Et comme je viens de vous le dire, et j'insiste là dessus, il faut à tout prix définir ce que nos variables vont contenir avant de les utiliser.

Une déclaration de variable se fait comme suit :

<Type de la variable> <Nom de la variable> ;

Cette opération se termine toujours par un ";" (comme toutes les instructions de ce langage) ;
ensuite, on l'initialise en rentrant une valeur.

Bon. Assez de bla-bla, on y va. (Décidément, moi, les rimes en ce moment...)

Les différents types de variables

Tout d'abord, je vous conseille vivement de lire le chapitre qu'a écrit M@teo21 sur les variables dans son tutoriel sur le C. Je pense que vous y verrez plus clair.

Bon. En Java, nous avons deux type de variables :

- des variables de type simple ou "primitif",
- des variables de type complexe ou encore des objets.

Ce qu'on appelle des types simples, ou types primitifs, en Java ce sont tout bonnement des nombres entiers, des nombres réels, des booléens ou encore des caractères. Mais vous allez voir qu'il y a plusieurs façons de déclarer certains de ces types.

Commençons par les variables de type numérique

- Le type **byte** (1 octet) peut contenir les entiers entre -128 et +127.
Ex. :

Code : Java

```
byte temperature;
temperature = 64;
```

- Le type **short** (2 octets) contient les entiers compris entre -32768 et +32767.
Ex. :

Code : Java

```
short vitesseMax;
vitesseMax = 32000;
```

- Le type **int** (4 octets) va de $-2*10^9$ à $2*10^9$ (2 et 9 zéros derrière... ce qui fait déjà un joli nombre).
Ex. :

Code : Java

```
int temperatureSoleil;
temperatureSoleil = 15600000;
```

C'est en kelvins...
- Le type **long**(8 octets) de $-9*10^{18}$ à $9*10^{18}$ (encore plus gros...).
Ex. :

Code : Java

```
long anneeLumiere;
anneeLumiere = 9460700000000000;
```

- Le type **float** (4 octets) correspond à des nombres avec virgule flottante.
Ex. :

Code : Java

```
float pi;
pi = 3.141592653f;
```

ou encore

Code : Java

```
float nombre;
```

```
nombre = 2.0f;
```

Vous remarquerez que nous ne mettons pas de virgule mais un **point !** Et vous remarquerez aussi que même si le nombre en question est rond, on met tout de même .0 derrière celui-ci !

- Le type **double** (8 octets) est identique à **float**, si ce n'est qu'il contient un nombre plus grand derrière la virgule.
Ex. :

Code : Java

```
double division;
division = 0.33333333333333333333333333333333333333333333334;
```

Nous avons aussi des variables stockant du caractère

- Le type **char** contient UN caractère stocké entre de simples quotes '' comme ceci...
Ex. :

Code : Java

```
char caractere;
caractere = 'A';
```

Nous avons aussi le type booléen

- Le type **boolean** qui lui contient `true` (vrai) ou `false` (faux).
Ex. :

Code : Java

```
boolean question;
question = true;
```

Et aussi le type String

- Celle-ci correspond à de la chaîne de caractères.
Ici, il ne s'agit pas d'une variable mais d'un objet qui instancie une classe qui existe dans Java ; nous pouvons l'initialiser en utilisant l'opérateur unaire **new()** dont on se sert pour réserver un emplacement mémoire à un objet (mais nous reparlerons de tout ceci dans la partie deux, lorsque nous verrons les classes), ou alors lui affecter directement la chaîne de caractères.
Vous verrez que celle-ci s'utilise très facilement et se déclare comme ceci :
Ex. :

Code : Java

```
String phrase;
phrase = "Titi et gros minet";
//Deuxième méthode de déclaration de type String
String str = new String();
str = "Une autre chaîne de caractères";
//La troisième
String string = "Une autre chaîne";
//Et une quatrième pour la route
String chaine = new String("Et une de plus ! ");
```

Attention : `String` commence par une majuscule ! Et lors de l'initialisation, on utilise ici des doubles quotes " " .

En fait, String n'est pas un type de variable mais un **objet**.
Notre variable est **"un objet"**, on dit aussi **"une instance"**, ici, une **instance de la classe String**. Nous y reviendrons lorsque nous aborderons les objets.

On te croit sur parole, mais pourquoi String commence par une majuscule et pas les autres ?

C'est simple. Il s'agit d'une convention de nommage.
En fait, c'est une façon d'appeler nos classes, nos variables, etc. Il faut que vous essayiez de respecter cette convention au maximum. Cette convention, la voici :

- **tous vos noms de classes doivent commencer par une majuscule !**
- **tous vos noms de variables doivent commencer par une minuscule.**
- Si un nom de variable est un nom composé, celui-ci commence par une minuscule et son **composé par une majuscule. Et ce, sans séparation**.
- Tout ceci sans accent ! !

Je sais que la première classe que je vous ai demandé de créer ne respecte pas cette convention, mais je ne voulais pas vous dire ça à ce moment-là... Donc, à présent, je vous demanderai de ne pas oublier ces règles !
Sinon ! Châtiment corporel.
Voici quelques exemples de noms de classes et de variables :

Code : Java

```
public class Toto{}
 public class Nombre{}
 public class TotoEtTiti{}
 String chaine;
 String chaineDeCaracteres;
 int nombre;
 int nombrePlusGrand;
 //...
```

Donc, pour en revenir au pourquoi du comment, je vous ai dit que les variables de type String sont des objets. Les objets sont définis par une ossature (un squelette) qui est en fait une classe. Ici, nous utilisons un objet String qui est défini dans une classe qui s'appelle "**`String`**" ; c'est pourquoi String à une majuscule et pas int, float... qui eux ne sont pas définis par une classe.

Pfiou ! Ça en fait des choses à retenir d'un coup ! Mais je ne vous cache pas que les deux premiers types de variables ne sont pas trop utilisés...

Chose importante : veillez à bien respecter la casse (majuscules et minuscules) car une déclaration de CHAR à la place de char ou autre chose provoquera une erreur, tout comme une variable de type string à la place de String !

Alors faites bien attention lors de vos déclarations de variables... Sinon une petite astuce quand même, enfin deux plutôt ! On peut très bien compacter la phase de déclaration et d'initialisation en une seule phase ! Comme ceci :

Code : Java

```
int entier = 32;
float pi = 3.1416f;
char carac = 'z';
String mot = new String("Coucou");
```

Et lorsque nous avons plusieurs variables d'un même type, nous pouvons compacter tout ceci en une déclaration comme ceci :

Code : Java

```
int nbre1 = 2, nbre2 = 3, nbre3 = 0;
```

Ici, toutes les variables sont des entiers et toutes initialisées.

Bon, t'es bien mignon, mais on code quand, nous ?

Minute papillon ! On va y arriver !

Mais avant, nous avons encore quelque chose de très important à voir pour pouvoir travailler sur et avec nos variables :

Les opérateurs arithmétiques

Retour en haut

Les opérateurs arithmétiques

Ceci est notre avant dernière ligne droite avant de commencer à coder ! Et après, vous regretterez le temps où vous n'aviez pas à vous creuser la tête !!

Bon, allez ! Je vois bien que vous brûlez d'impatience, alors on va y aller tout de suite.

Les opérateurs arithmétiques

Ce sont ceux que l'on apprend à l'école primaire...

\+ permet d'ajouter deux variables numériques (mais aussi de concaténer des chaînes de caractères ! Ne vous inquiétez pas, on aura l'occasion d'y revenir).

\- permet de soustraire deux variables numériques.

* permet de multiplier deux variables numériques.

/ Permet de diviser deux variables numériques.(mais je crois que vous aviez deviné).

% permet de renvoyer le reste de la division de deux variables de type numériques, cet opérateur s'appelle le **modulo**.

Quelques exemples de calcul

Code : Java

```
int nbre1, nbre2, nbre3;//déclaration des variables
nbre1 = nbre2 = nbre3 = 0;//initialisation

nbre1 = 1 + 3;          // ici nbre1 vaut 4
nbre2 = 2 * 6;          // ici nbre2 vaut 12
nbre3 = nbre2 / nbre1;      //ici nbre3 vaut 3
nbre1 = 5 % 2;          //ici nbre1 vaut 1 car 5 = 2 * 2 + 1
nbre2 = 99 % 8;         //ici nbre2 vaut 3 car 99 = 8 * 12 + 3
nbre3 = 6 % 3;          //et là, nbre3 vaut 0 car la division n'a
aucun reste
```

Ici, nous voyons bien que nous pouvons affecter des opérations sur des nombres à nos variables mais aussi affecter des opérations sur des variables de même type !!

Je me doute bien que le modulo est assez difficile à assimiler. Voici une utilisation assez simple : Pour déterminer si un entier est pair, il suffit de voir si cet entier modulo 2 renvoie 0 .

Maintenant, quelque chose que les personnes qui n'ont jamais programmé ont du mal à assimiler. Je garde la même déclaration de variable que ci-dessus.

Code : Java

```
int nbre1, nbre2, nbre3;        //déclaration des variables
nbre1 = nbre2 = nbre3 = 0;                  //initialisation

nbre1 = nbre1 + 1;  //nbre1 = lui même donc 0 + 1 => nbre1 = 1
nbre1 = nbre1 + 1;  // nbre1 = 1 (cf ci-dessus) maintenant nbre1 = 1
+ 1 = 2!!!
nbre2 = nbre1;          //nbre2 = nbre1 = 2
nbre2 = nbre2 * 2;  //nbre2 = 2 => nbre2 = 2 * 2 = 4
nbre3 = nbre2;          //nbre3 = nbre2 = 4
nbre3 = nbre3 / nbre3;      //nbre3 = 4 / 4 = 1
nbre1 = nbre3;         //nbre1 = nbre3 = 1
nbre1 = nbre1 - 1;// nbre1 = 1 - 1 = 0
```

Et là aussi, il existe une syntaxe qui raccourcit l'écriture de ce genre d'opération. Regardez bien :

Code : Java

```
nbre1 = nbre1 + 1;
nbre1 += 1;
nbre1++;
++nbre1;
```

Ces trois syntaxes correspondent exactement à la même opération. La troisième syntaxe sera certainement celle que vous utiliserez le plus... mais ne marche que pour augmenter la valeur de 1 ! Si vous devez augmenter la valeur d'une variable de 2, utilisez les deux syntaxes précédentes. On appelle cette syntaxe l'**incrémentation**. La dernière fait la même chose que la troisième, mais avec une subtilité en plus... Nous en reparlerons dans le chapitre sur les boucles.

Sinon, la syntaxe est identique pour la soustraction :

Code : Java

```
nbre1 = nbre1 - 1;
nbre1 -= 1;
nbre1--;
--nbre1;
```

Même commentaire que pour l'addition sauf qu'ici, la troisième syntaxe s'appelle la **décrémentation**.

Les raccourcis pour la multiplication marchent aussi ; regardez plutôt :

Code : Java

```
nbre1 = nbre1 * 2;
nbre1 *= 2;
nbre1 = nbre1 / 2;
nbre1 /= 2;
```

TRES IMPORTANT == On ne peut faire de traitement arithmétique que sur des variables de même type, sous peine de perdre de la précision lors du calcul. On ne s'amuse pas à diviser un **int** par un **float** ! Ou pire, par un **char** !! Et ceci est valable pour tous les opérateurs arithmétiques et pour tous les types de variables numériques. Essayer de garder une certaine rigueur pour vos calculs arithmétiques.

Voici les raisons de ma mise en garde.
Comme je vous l'ai dit plus haut, chaque type de variable a une contenance différente et, pour faire simple, nous allons comparer nos variables à différents récipients.
Une variable de type :

- `byte` correspondrait à un dé à coudre. On ne met pas beaucoup de chose dedans...
- `int`, à un verre. C'est déjà plus grand.
- `double`, à un baril. Pfiou, on en met là-dedans...

A partir de là, ce n'est plus que du bon sens. Vous devez facilement voir que vous pouvez mettre le contenu d'un dé à coudre dans un verre ou un baril. Mais par contre, si vous versez le contenu d'un baril dans un verre... Il y en a plein par terre !
Cela s'appelle : **une perte de précision !**

Ainsi, si nous affectons le résultat d' une opération sur deux variables de type `double` dans une variable de type `int`, le résultat sera de type `int` et **donc ne sera pas un réel mais un entier**.

Pour afficher le contenu d'une variable dans la console, il vous suffit d'appeler l'instruction :

`System.out.println(maVariable);`, ou encore `System.out.print(maDeuxiemeVariable);`.

Je suppose que vous voudriez aussi mettre du texte en même temps que vos variables... Eh bien sachez que l'opérateur + sert aussi comme opérateur de concaténation, c'est-à-dire qu'il permet de mixer du texte brut avec des variables.

Voici un exemple d'affichage avec une perte de précision :

Code : Java

```
double nbre1 = 10, nbre2 = 3;
 int resultat = (int)(nbre1 / nbre2);
 System.out.println("Le résultat est = " + resultat);
```

Sachez aussi que vous pouvez tout à fait mettre des calculs dans un affichage, comme ceci `System.out.print("Résultat = " + nbre1/nbre2);` (le plus joue ici le rôle d'opérateur de concaténation) ; ceci vous permet d'économiser une variable et donc de la mémoire.

Mais pour le bien de ce chapitre, nous n'allons pas utiliser cette méthode...
Vous devez voir que le résultat fait 3, au lieu de 3.33333333333333... Et je pense que vous êtes intrigués par ça :
int resultat = (int)(nbre1 / nbre2);

Avant de vous expliquer, remplacez la ligne citée ci-dessus par celle-ci :
`int resultat = nbre1 / nbre2;`

Vous devez voir qu'Eclipse n'aime pas du tout !
Pour savoir pourquoi, nous allons voir ce qu'on appelle **les conversions** ou **"cast"**.
Retour en haut

Les conversions, ou "cast"

Comme expliqué plus haut, les variables de type `double` contiennent plus d'informations que les variables de type `int`.

Ici, il va falloir écouter comme il faut... heu, pardon ! Lire comme il faut !
Nous allons voir un truc super important en Java. Ne vous en déplaise, vous serez amenés à convertir des variables...

Conversion de type `int` vers un autre type numérique

D'un type **`int`** en type **`float`** :

Code : Java

```
int i = 123;
float j = (float)i;
```

D'un type **`int`** en **`double`** :

Code : Java

```
int i = 123;
double j = (double)i;
```

Et inversement :

Code : Java

```
double i = 1.23;
double j = 2.9999999;
int k = (int)i; // k vaut 1
k = (int)j; // k vaut 2
```

Ce type de conversion s'appelle une **conversion d'ajustement** ou **cast** de variable.

Vous l'avez vu : nous pouvons passer directement d'un type **`int`** à un type **`double`**. Mais ceci ne fonctionnera pas dans le sens inverse sans une perte de précision.
En effet comme vous avez pu le constater, lorsque nous castons un `double` en `int`, la valeur de ce `double` est **tronqué**. Ce qui signifie que l'`int` en question prendra la valeur entière du `double` **quelle que soit la valeur des décimales**.

Pour en revenir à notre ancien problème, il est aussi possible de caster le résultat d'une opération mathématique. En mettant celle-ci entre **()** et précédée du type de cast souhaité.
Donc :

Code : Java

```
double nbre1 = 10, nbre2 = 3;
 int resultat = (int)(nbre1 / nbre2);
 System.out.println("Le résultat est = " + resultat);
```

Fonctionne parfaitement. Mais pour un résultat optimal, vous devez mettre le résultat de l'opération en type `double`.

Et si on faisait l'inverse ? Si nous déclarons deux entiers et que nous mettons le résultat dans un `double` ?

Comme ceci ?

Code : Java

```
int nbre1 = 3, nbre2 = 2;
 double resultat = nbre1 / nbre2;
 System.out.println("Le résultat est = " + resultat);
```

Vous auriez **1** comme résultat.
Je ne cast pas ici, car un `double` peut contenir un `int`.

Et comme ça ?

Code : Java

```
int nbre1 = 3, nbre2 = 2;
 double resultat = (double)(nbre1 / nbre2);
 System.out.println("Le résultat est = " + resultat);
```

Idem...

Comment doit-on faire, alors ?

Avant de vous donner la réponse, vous devez savoir qu'en Java, comme dans d'autres langages d'ailleurs, il y a la notion de **priorité d'opération** et là, nous en avons un très bon exemple !

Sachez que l'affectation, le calcul, le cast, le test, l'incrémentation... toutes ces choses sont des opérations !
Et Java les fait dans un certain ordre, suivant une priorité.

Dans le cas qui nous intéresse, il y a trois opérations :

- un calcul,
- un cast de valeur de résultat de calcul,
- une affectation dans la variable `resultat`.

Eh bien Java exécute cette ligne dans cet ordre !
Il fait le calcul (ici 3/2), il cast le résultat en `double`, puis il l'affecte dans notre variable `resultat`.

D'accord, mais pourquoi on n'a pas 1.5, alors ?

C'est simple : lors de la première opération de Java, la JVM voit un cast à effectuer mais sur un résultat de calcul. La JVM fait ce calcul (division de deux `int` qui, ici, nous donne 1), puis le cast (toujours 1) et affecte la valeur à la variable (encore et toujours 1).
Donc, pour avoir un résultat correct, il faudrait caster chaque nombre avant de faire l'opération, comme ceci :

Code : Java

```
int nbre1 = 3, nbre2 = 2;
 double resultat = (double)(nbre1) / (double)(nbre2);
 System.out.println("Le résultat est = " + resultat); //affiche: Le
résultat est = 1.5
```

Message reçu mais... peut-on changer un type numérique en autre chose ?

Bien sûr, je ne détaillerai pas trop ceci mais maintenant, vous allez transformer l'argument d'un type donné, `int` par exemple, en `String`.

Voici la méthode à appliquer

Code : Java

```
int i = 12;
    String j = new String();
     j = j.valueOf(i);
```

j est donc une variable de type `String` contenant la ***chaîne de caractères 12***. **Ceci fonctionne aussi avec les autres types numériques.** Voyons maintenant comment faire marche arrière, en partant de ce que nous venons de faire.

Code : Java

```
int i = 12;
String j = new String();

        j = j.valueOf(i);

        int k = Integer.valueOf(j).intValue();
```

Maintenant, la variable **k** est de type **int**, et contient le nombre ***12***.

Il y a l'équivalent de `intValue()` pour les autres types numérique : `floatValue()`, `doubleValue()`...

Retour en haut

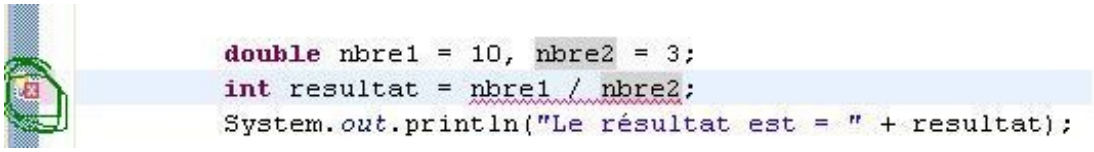
Astuce d'Eclipse

Retapez le code qu'Eclipse n'aimait pas ; pour mémoire, c'était celui-ci :

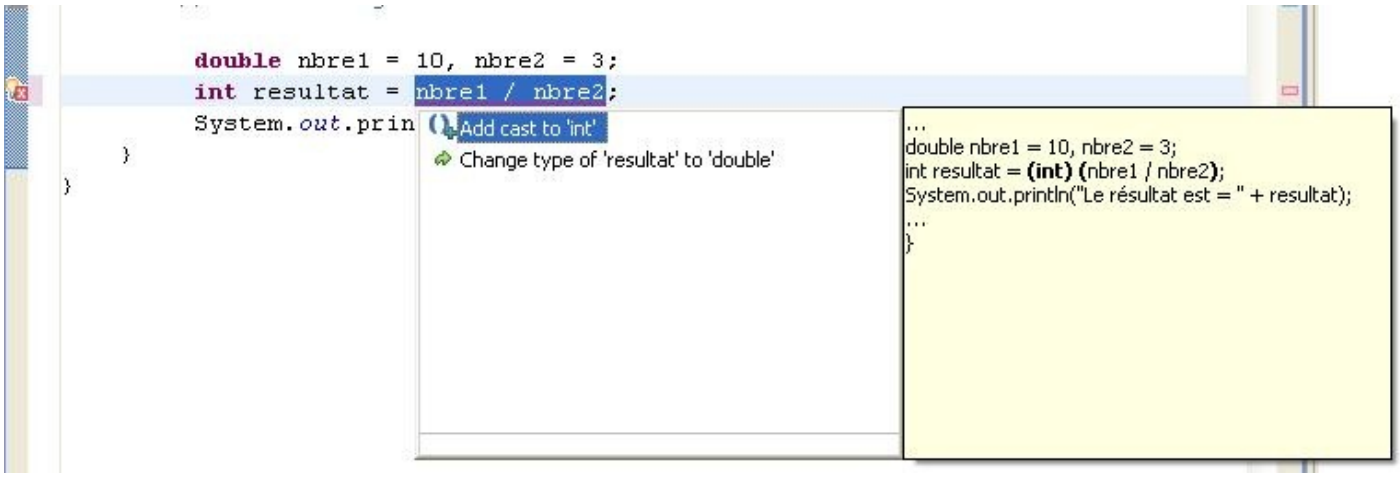
Code : Java

```
double nbre1 = 10, nbre2 = 3;
int resultat = nbre1 / nbre2;
System.out.println("Le résultat est = " + resultat);
```

Eclipse vous souligne `nbre1 / nbre2` et vous met une croix rouge sur la gauche de la zone d'édition, sur cette même ligne.



Si vous cliquez sur cette croix rouge, vous aurez ceci :



Double-cliquez sur **"Add cast to 'int'"** et Eclipse vous met automatiquement le cast de l'opération !

Ensuite pour tester, vous pouvez cliquez sur **"Run"** ou appuyer sur **Control + F11**.
Si vous faites cette manipulation (Ctrl + F11), une fenêtre s'ouvre et vous demande quelle ressource sauver, puis lancer. Pour le moment, le choix est simple puisque vous n'avez qu'une classe. Vous n'avez plus qu'à valider et votre programme se lance !

Au fil de ce tutoriel, vous verrez que cette manipulation va beaucoup vous aider pour des erreurs en tous genres !

Retour en haut

Je suis bien conscient que ces deux chapitres étaient riches en vocabulaire, en nouveautés (pour ceux qui n'auraient pas lu les cours de M@teo), mais bon, voilà : votre calvaire... ne fait que commencer.

Eh oui : tout ceci était un amuse-bouche ! Maintenant, nous rentrons dans le vif du sujet. Dans tous les cas, si vous n' avez pas eu 20/20 à ces deux questionnaires, relisez ces deux chapitres !!

Allez on s' y jette : ***Lire les entrées clavier*** !
TAAAAÏÏÏÏAAAAUUUUTTTT !

Retour en haut

Lire les entrées clavier

Dans ce chapitre, nous verrons comment lire les entrées clavier.
Nous survolerons ceci sans voir les différents cas d'erreurs que cela peut engendrer.
Trêve de bavardage, je suis sûr que vous êtes impatients...

Utilisation de l'objet Scanner

Je me doute qu'il vous tardait de pouvoir communiquer avec votre application...
Le moment est enfin venu ! Mais je vous préviens, la méthode que je vais vous donner a des failles.
Je vous fais confiance pour ne pas rentrer n'importe quoi n'importe quand...

Pour les puristes qui me diront "*ouais, mais si je rentre ça, tu vas te prendre une belle exception dans ta face !*", je le sais, mais je ne trouve pas opportun de vous parler des exceptions et de la manière de les capturer maintenant... Bon. Allons-y !

Je vous ai dit que vos variables de type **String** sont en fait des **objets** de type **String**. Pour que Java puisse lire ce que vous tapez au clavier, vous allez devoir utiliser un objet de type **Scanner** **(merci à Myhtrys)**.

Je vous rappelle que pour **instancier** un objet, c'est-à-dire créer un nouvel objet afin de s'en servir, **vous devez utiliser l'opérateur unaire new()**.

Cet objet peut prendre différents paramètres, mais ici, nous n'en utiliserons qu'un. Celui qui correspond justement à l'entrée standard en Java.
Lorsque vous faites **System.out.println();**, je vous rappelle que vous appliquez la méthode println() sur la sortie standard ; or ici, nous allons utiliser l'entrée standard **System.in**.
Donc, avant de dire à Java de lire ce que nous allons taper au clavier, nous devrons instancier un objet Scanner.

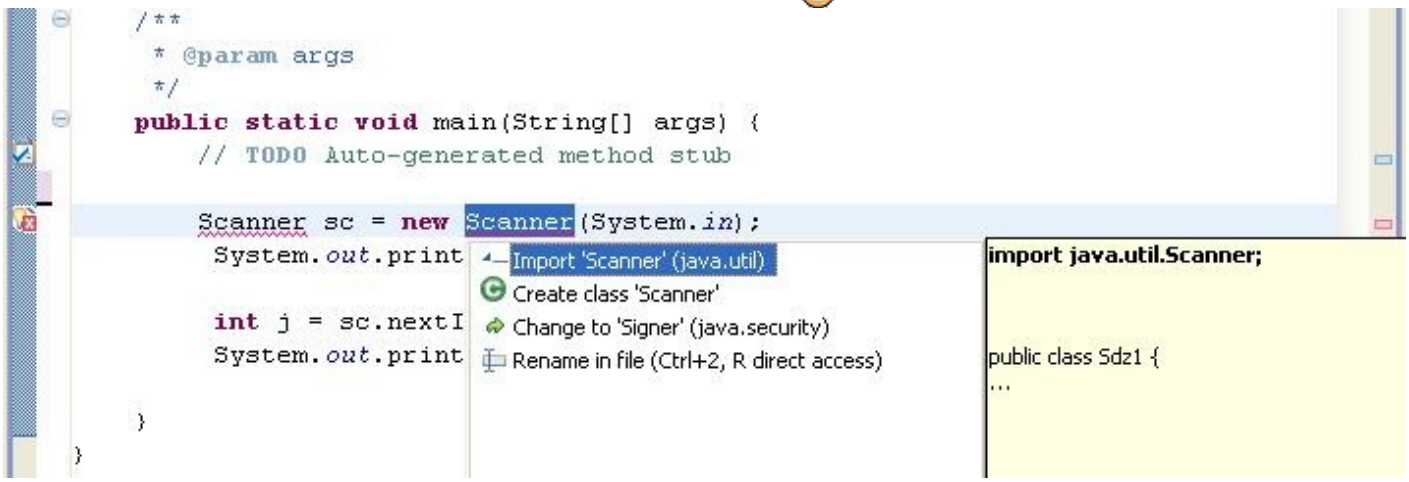
Pour pouvoir utiliser un objet Scanner, nous devons dire à Java où trouver cet objet !

Avant de vous expliquer ceci, créez une nouvelle classe et tapez cette ligne de code dans votre méthode main :

Code : Java

```
Scanner sc = new Scanner(System.in);
```

Vous devez avoir une zolie vague rouge sous le mot **Scanner**.
Mais souvenez-vous de l'astuce que je vous avais donnée pour le cast de variables. Cliquez sur la croix rouge sur la gauche et double-cliquez sur "***Import 'Scanner' java.util***". Et là, l'erreur disparaît !



Maintenant, regardez tout au dessus de votre classe, au dessus de la déclaration de celle-ci, vous devez avoir une ligne :

Code : Java

```
import java.util.Scanner;
```

Voilà ce que nous avons fait. Je vous ai dit qu'il fallait dire à Java où se trouve la classe Scanner.
Pour faire ceci, nous devons **importer la classe Scanner**, grâce à l'instruction **import**.
La classe que nous voulons se trouve dans le **package java.util**.

Tout d'abord, vous devez savoir que le langage Java n'est qu'une multitude de classes ordonnées en **packages**. Par défaut, Java incorpore automatiquement un package contenant les fondements de Java : **java.lang**. C'est dans ce package qu'il y a les variables de bases, la classe System et beaucoup d'autres choses qui vous permettent de faire des programmes. Pour voir le contenu de ce package, vous pouvez allez voir **ici**.

Je pense que vous avez compris qu'un package est en fait un ensemble de classes. En fait, c'est un ensemble de dossiers et de sous-dossiers contenant une ou plusieurs classes.

Par exemple, nous avons importé tout à l'heure la classe Scanner qui se trouve dans le package **java.util**. Remplacez les **.** par des **/**, l'arborescence des dossiers est donc **java/util/** et dans ce dossier se trouve le fichier **Scanner.class** ! Vous verrez ceci plus en détail lorsque nous ferons nos propre packages.

Les classes qui se trouvent dans les autres packages que **java.lang** sont à importer *à la main* dans vos classes Java pour pouvoir vous en servir.
La façon dont nous avons importé la classe java.util.Scanner dans Eclipse est très pratique et très simple. Vous pouvez aussi le faire manuellement en tapant :

Code : Java

```
//Ceci importe la classe Scanner du package java.util
import java.util.Scanner;
//Ceci importe toutes les classes du package java.util
import java.util.*;
```

Si vous faites vos imports manuellement, n'oubliez surtout pas le **;** à la fin !

Maintenant que vous avez assimilé ceci, nous pouvons retourner à nos moutons !

Retour en haut

Récupérez ce que vous tapez

Retournons à notre morceau de code, pour mémoire :

Code : Java

```
Scanner sc = new Scanner(System.in);
```

Afin de vous préparer à saisir des informations, veuillez afficher un message à l'écran ; nous avons donc :

Code : Java

```
Scanner sc = new Scanner(System.in);
System.out.println("Veuillez saisir un mot :");
```

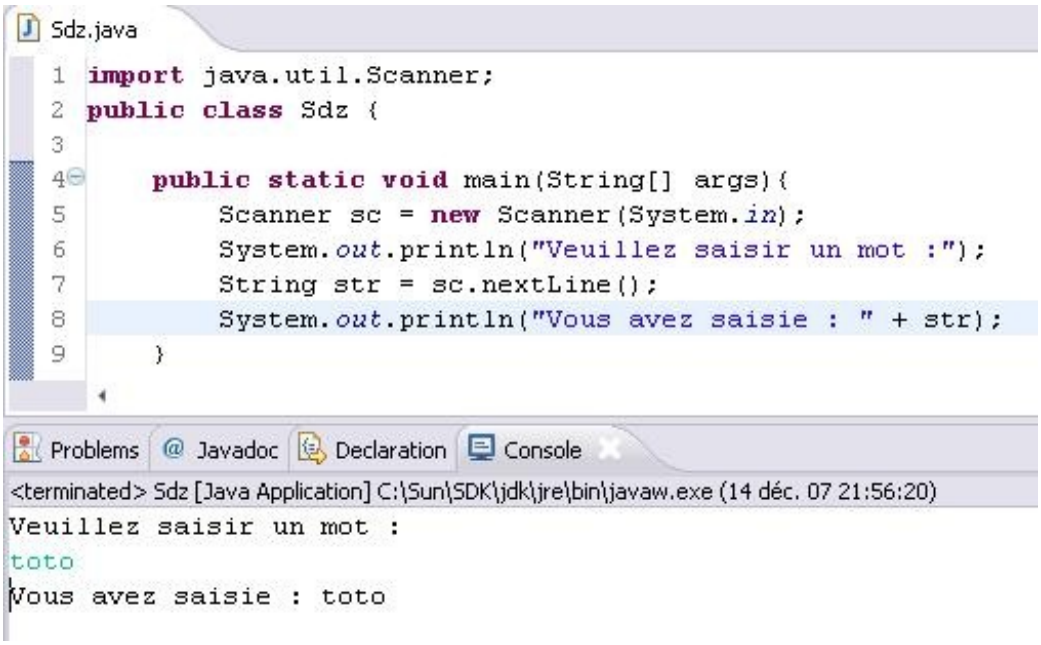
Maintenant, voici l'instruction pour permettre à Java de récupérer ce que vous avez saisi et ensuite de l'afficher :

Code : Java

```
Scanner sc = new Scanner(System.in);
System.out.println("Veuillez saisir un mot :");
String str = sc.nextLine();
System.out.println("Vous avez saisi : " + str);
```

Une fois l'application lancée, le message que vous avez écrit auparavant s'affiche dans la console, en bas dans Eclipse (vous avez l'habitude, maintenant).

Pensez à cliquer dans la console, après votre message, afin que ce que vous saisissez soit écrit dans la console, pour que Java puisse récupérer ce que vous avez inscrit !



Alors ? Heureux ?

Voilà votre première saisie clavier ! Comme vous avez pu le constater, l'instruction `nextLine()` renvoie une chaîne de caractères. Si vous avez essayé de remplacer la variable de type `String` par une variable d'un autre type, vous avez dû voir qu'Eclipse n'apprécie pas du tout... Et si vous avez cliqué sur la croix rouge pour corriger le problème, vous constatez que la seule solution qu'il vous propose est de changer le type de votre variable pour le type String.

Vous aurez remarqué aussi qu'Eclipse vous simplifie la vie au maximum. Lorsque vous tapez "**`sc.`**", Eclipse vous propose une liste de méthodes appartenant à cet objet.

Si vous remplacez la ligne de code qui récupère une chaîne de caractères comme suit :

Code : Java

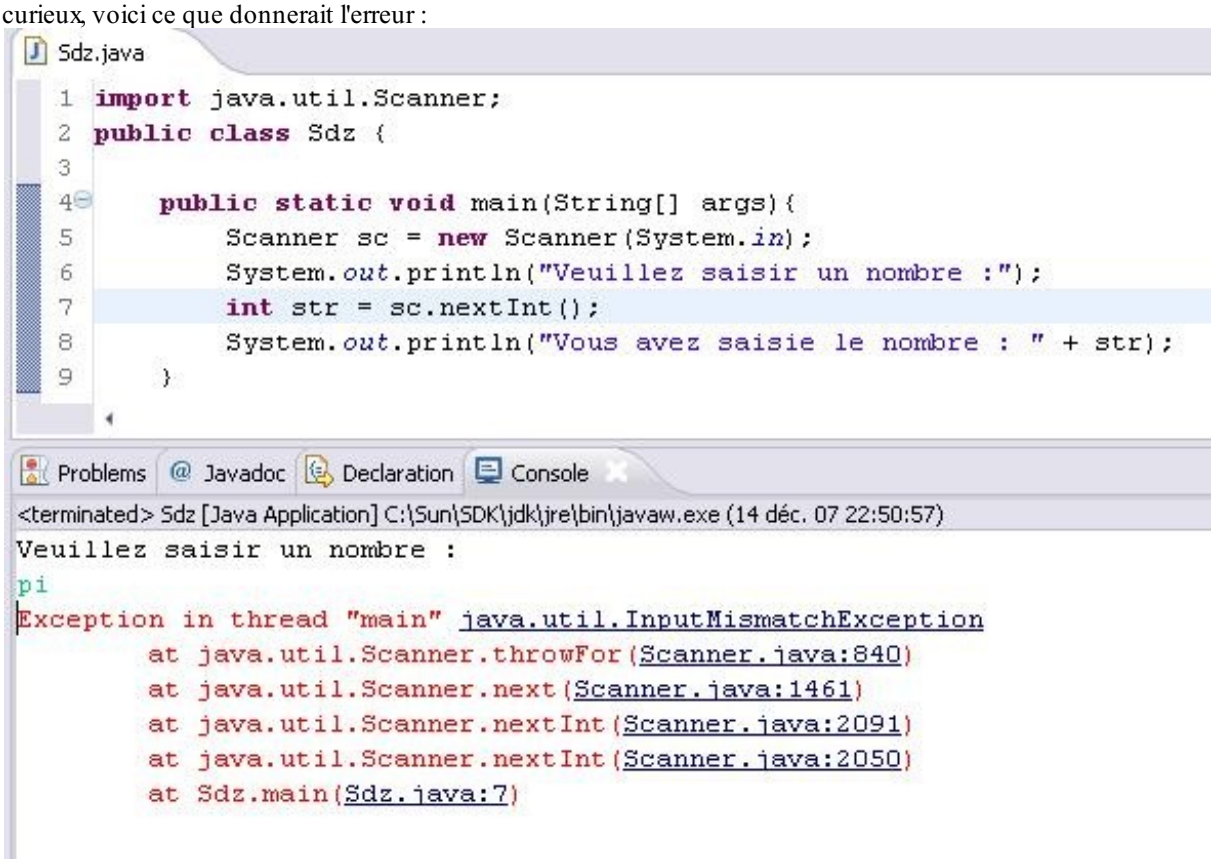
```
Scanner sc = new Scanner(System.in);
System.out.println("Veuillez saisir un nombre :");
int str = sc.nextInt();
System.out.println("Vous avez saisi le nombre : " + str);
```

vous devez voir que lorsque vous utilisez votre variable de type Scanner, et où vous tapez le point permettant d'appeler des méthodes de l'objet, Eclipse vous donne une liste de méthodes associées à cet objet et, lorsque vous commencez à taper le début de la méthode **`nextInt()`**, le choix se restreint jusqu'à ne laisser que cette seule méthode.

Exécutez et testez ce programme et vous verrez qu'il fonctionne à la perfection ! **Sauf... si vous saisissez autre chose qu'un entier !**

C'est ce que je vous disais au départ de ce chapitre. L'objet essaie de récupérer un entier, mais si vous lui donnez autre chose, une exception est levée ! Nous verrons ceci lorsque nous programmerons des objets...

Pour les curieux, voici ce que donnerait l'erreur :



Vous savez maintenant que pour lire un `int`, vous devez utiliser **`nextInt()`**.
De façon générale, dites-vous que pour récupérer un type de variable, il vous suffit d'appeler `next<Type de variable commençant par une majuscule>` (rappelez-vous de la convention de nommage !).
Bon. C'est mon jour de bonté :

Code : Java

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
double d = sc.nextDouble();
long l = sc.nextLong();
byte b = sc.nextByte();
//etc
```

Attention : il y a un type de variable primitive qui n'est pas pris en compte par la classe `Scanner` ; il s'agit des variables de type **`char`**.

Voici comment on pourrait récupérer un caractère :

Code : Java

```
System.out.println("Saisissez une lettre :");
Scanner sc = new Scanner(System.in);
String str = sc.nextLine();
char carac = str.charAt(0);
System.out.println("Vous avez saisi le caractère : " + carac);
```

Qu'est-ce que nous avons fait ici ?

Nous avons récupéré une chaîne de caractères, puis nous avons utilisé une méthode de l'objet `String` (ici, **`charAt(0)`**) afin de récupérer le premier caractère saisi !

Même si vous tapez une longue chaîne de caractères, l'instruction **`charAt(0)`** ne renverra que le premier caractère...

Vous devez vous demander pourquoi `charAt(0)` et non `charAt(1)` ? Ne vous inquiétez pas, nous aborderons ce point lorsque nous verrons les tableaux...

Jusqu'à ce qu'on aborde les exceptions, je vous demande d'être rigoureux et de faire attention à ce que vous attendez comme donnée afin d'utiliser la bonne méthode.

Une précision toutefois. La méthode `nextLine()` récupère le contenu de toute la ligne saisie et repositionne la "*tête de lecture*" au début d'une autre ligne. Par contre, si vous avez invoqué une méthode comme `nextInt()` , `nextDouble()` et si vous invoquez directement derrière la méthode `nextLine()` , celle-ci ne vous invitera pas à saisir une chaîne de caractères mais elle videra la ligne commencée précédemment par les autres instructions car celles-ci ne repositionnent pas la tête de lecture, l'instruction `nextLine()` le fait donc à leurs place. Pour faire simple, ceci :

Code : Java

```
import java.util.Scanner;

public class Main {
 public static void main(String[] args){
  Scanner sc = new Scanner(System.in);
  System.out.println("Saisissez un entier : ");
  int i = sc.nextInt();
  System.out.println("Saisissez une chaîne : ");
  String str = sc.nextLine();
  System.out.println("FIN ! ");
 }
}
```

ne vous demandera pas saisir une chaîne et affichera directement "Fin". Pour pallier ce problème il suffit de vider la ligne après les instructions ne le faisant pas :

Code : Java

```
import java.util.Scanner;

public class Main {
 public static void main(String[] args){
  Scanner sc = new Scanner(System.in);
  System.out.println("Saisissez un entier : ");
  int i = sc.nextInt();
```

```
        System.out.println("Saisissez une chaîne : ");
                    //On vide la ligne avant d'en lire une autre
                    sc.nextLine();
        String str = sc.nextLine();
        System.out.println("FIN ! ");
      }
    }
```

Voilà : je pense que vous êtes fin prêts pour un QCM, maintenant !
Retour en haut
J'espère que cette partie vous a plu et que vous êtes encore en forme...
A partir de maintenant, nous allons aborder les différentes instructions du langage.
Commençons par **les conditions**.
Retour en haut

Les conditions

Nous abordons ici un des chapitres les plus importants et les plus utilisés.
Vous allez voir que tous vos projets ne sont que des enchaînements et des imbrications de **conditions et de boucles (partie suivante).**
Dans une classe, la lecture et l'exécution se font de façon séquentielle. C'est-à-dire en suivant, ligne par ligne. Avec les **conditions**, nous allons pouvoir gérer différents cas de figure, sans pour autant lire tout le code.
Assez de belles paroles ! Rentrons tout de suite dans le vif du sujet.

Les opérateurs logiques

Ceux-ci sont surtout utilisés lors de conditions (SI****** alors fait ceci*****) pour tester des vérités ou des contrevérités ! Mais nous verrons plus en détails leur utilité dans un autre chapitre ! Je vous les colle ici car ils sont faciles à mémoriser ; et puis comme ça, c'est fait, on n'en parlera plus.

== permet de tester l'égalité. Prenons l'exemple complètement inutile suivant (en français, pour le code Java, ce sera plus tard...). SI bleu == bleu alors fait ceci.....

!= pas égal ou encore différent de. Je pense que c'est assez parlant, non ?

< strictement inférieur.

<= inférieur ou égal. Vous l'aviez déjà trouvé, je suis sûr ! ! !

> strictement supérieur.

>= eh oui, voici son grand frère, le majestueux supérieur ou égal !

&& voici l'opérateur ET. Permet de préciser une condition.

|| est le cousin du ET, le OU. Même combat que le précédent.

? : L'opérateur ternaire, pour celui-ci vous comprendrez mieux avec un exemple... vers la fin de ce chapitre.

Comme je vous l'ai dit dans la partie précédente, les opérations en Java sont soumises à un ordre de priorité. Tous ces opérateurs sont soumis à cette règle, de la même manière que les opérateurs arithmétiques...

Il n'y a pas grand-chose à dire sur ces opérateurs sans un exemple concret, donc allons-y.

Retour en haut

La structure if....else

Pour commencer, je vais vous expliquer à quoi servent ces structures conditionnelles. Elles servent tout simplement à pouvoir constituer un programme, en examinant les différents cas de figure que celui-ci propose.
Je m'explique : imaginons un programme qui demande à un utilisateur de rentrer un nombre réel (qui peut être soit négatif, soit nul, soit positif). Les structures conditionnelles vont nous permettre de gérer ces trois cas de figure.

La structure de ces conditions ressemble à ça :

Code : Java

```
if(//condition)
{
.....// exécution des instructions si la condition est remplie
.....
.....
}
else
{
....// exécution des instructions si la condition n'est pas remplie
....
....
}
```

Ceci peut se traduire par "SI... SINON...".

Mettons notre petit exemple du dessus en pratique :

Code : Java

```
int i = 10;

if (i < 0)
        System.out.println("Le nombre est négatif");

else
        System.out.println("Le nombre est positif");
```

Testez ce petit code, et vous verrez comment il fonctionne (par exemple, avec la fonction de pas à pas).
Dans ce cas, notre classe affiche que "***le nombre est positif***".
Expliquons un peu ce qui se passe :

- dans un premier temps, la condition du `if` est testée... (qui dit SI `i` est strictement inférieur à 0).
- Vu que celle-ci est fausse, le programme exécute le `else`.

Attends un peu ! Lorsque tu nous a présenté la structure des conditions, tu as mis des accolades et là, tu n'en mets pas...

Bien observé. En fait, les accolades sont la structure "*normale*" des conditions mais, lorsque le code à l'intérieur d'une d'entre elles n'est composé que **d'une seule ligne de code, les accolades deviennent facultatives**.

Comme nous avons l'esprit perfectionniste, nous voulons que notre programme affiche "**le nombre est nul**", lorsque `i` est égal à 0 ; nous allons donc rajouter une condition.
Comment faire... La condition du `if` est remplie si le nombre est strictement négatif, ce qui n'est pas le cas ici puisque nous allons le mettre à 0, le code contenu dans la clause `else` est donc exécuté si le nombre est égal à 0 et strictement supérieur à 0.
Il nous suffit de rajouter une condition à l'intérieur de la clause `else`. Comme ceci :

Code : Java

```
int i = 0;
        if (i < 0)
                System.out.println("Ce nombre est négatif !");

        else
        {
                if(i == 0)
                        System.out.println("Ce nombre est nul !!");

                else
                        System.out.println("Ce nombre est positif
!!");

        }
```

Ici, la seule petite chose qui devrait vous interpeler, c'est l'**imbrication** d'une structure **`if... else` dans un `else`**. Et encore, parce que je suis tolérant...
Vous voyez aussi que le code à l'intérieur de la première clause **`if`** ne contient qu'une seule ligne de code **[=> accolades facultatives]** et que la clause **`else`** correspondante, elle, a plusieurs ligne en son sein **[=> on entoure donc le code de cette dernière avec des accolades]**. Les clauses à l'intérieur du premier **`else`** n'ont, elles aussi, qu'une seule ligne de code **[=> accolades facultatives].**

Vous verrez vite que vos programmes ne seront que des imbrications de conditions.
Je vous fais voir une autre manière de le faire :

Code : Java

```
int i = 0;
        if (i <= 0)
        {
                if(i == 0)
                  System.out.println("Ce nombre est nul !!");

                else
                  System.out.println("Ce nombre est négatif !");
        }
        else
           System.out.println("Ce nombre est positif !!");
```

Je pense que ce code parle de lui-même...
Ici, la clause du `if` est remplie si `i` est **INFÉRIEUR OU ÉGAL** à 0, `i` vaut 0 la condition est remplie. La suite, vous la connaissez.

Maintenant que vous avez tout compris, je vais vous montrer une autre façon de faire ce code, avec le même résultat (encore heureux !). En ajoutant juste un petit ***SINON SI*** . Regardez bien la magie.

Code : Java

```
int i = 0;
  if (i < 0)
    System.out.println("Ce nombre est négatif !");

  else if(i > 0)
    System.out.println("Ce nombre est positif !!");

  else
    System.out.println("Ce nombre est nul !!");
```

Alors ? Explicite, n'est-ce pas ?

- **SI** `i` est strictement négatif => exécution du code.
- **SINON SI** `i` est positif => exécution du code.
- **SINON** `i` est forcément nul => exécution du code.

Il faut absolument donner au **`else if`** une condition pour qu'il fonctionne. Sans cela, Eclipse vous mettra de zolies vagues rouges sous votre **`else if`**.
Vous en avez sûrement déjà vu à de nombreuses reprises...

Par contre, je vais TRÈS FORTEMENT INSISTER sur un point : regardez l'affichage du code : remarquez le petit décalage entre ma première condition et ma deuxième.

On appelle ceci **l'indentation**, et comme c'est écrit en gros, en gras et en rouge, c'est que c'est hyper important !

En effet, pour vous repérer dans vos futurs programmes, cela sera très utile. Imaginez deux secondes que vous avez un programme de 700 lignes avec 150 conditions, et que tout est écrit le long du bord gauche. Vous allez vous amuser pour retrouver où commence et où se termine une condition. **Je vous le garantis !**
Vous n'êtes pas obligés de le faire, mais je vous assure que vous y viendrez.

À titre d'information, n'essayez pas de faire des comparaisons de **`String`** à ce stade. Je vous expliquerai la marche à suivre lors du chapitre sur ***les fonctions***.

Je vois que vous apprenez vite : nous pouvons donc passer à la vitesse supérieure !
Voyons tout de suite ***les conditions multiples.***
Retour en haut

Les conditions multiples

Avant de commencer, vous devez savoir **qu'on ne peut pas tester l'égalité de chaînes de caractères !**
Du moins pas comme je vous l'ai montré ci-dessus... Nous aborderons ce point plus tard.

Derrière ce nom barbare, se trouve simplement une ou deux (ou X) conditions en plus dans un **`if`**, ou un **`else if`**. Nous allons maintenant utiliser les opérateurs logiques que nous avons vus au début. Pfiou ! C'est vieux !
Alors dans ce cas :

- ***SI*** c'est vieux et ça va aller => alors on continue.
- ***SINON SI*** c'est vieux et je ne me rappelle plus => on va relire le début !
- ***SINON***, allons-y tout de suite !

Voilà un bel exemple de conditions multiples ! Et je n'ai pas été la chercher loin, celle-là. Elle ressemble beaucoup à la condition de notre programme (plus haut).
Je sais par expérience qu'on comprend mieux avec un exemple ; donc, allons-y...

Maintenant, nous allons vérifier si un nombre donné appartient à un intervalle connu ; par exemple, savoir si un nombre est entre 50 et 100. Nous allons essayer de résoudre ce problème avec les outils que nous avons. En gros, ça donnerait quelque chose comme ça :

Code : Java

```
int i = 58;
 if(i < 100)
 {
   if(i > 50)
     System.out.println("Le nombre est bien dans l'intervalle");

   else
    System.out.println("Le nombre n'est pas dans l'intervalle");
 }

 else
   System.out.println("Le nombre n'est pas dans l'intervalle");
```

Rien de bien compliqué : notre objectif dans ce programme est de repérer si un nombre répond à deux conditions, il faut :

- qu'il soit inférieur à 100
- qu'il soit supérieur à 50.

Eh bien les conditions multiples peuvent éliminer deux lignes dans notre précédent code. Regardez plutôt :

Code : Java

```
int i = 58;
 if(i < 100 && i > 50)
   System.out.println("Le nombre est bien dans l'intervalle");

 else
   System.out.println("Le nombre n' est pas dans l'intervalle");
```

Nous avons utilisé l'opérateur **`&&`** qui signifie **ET**. Donc, la condition de notre `if` est devenu :

si `i` est inférieur à 100 **ET** supérieur à 50, alors la condition est remplie.

Avec l'opérateur **`&&`**, la clause est remplie **si et seulement si les conditions formant la clause sont toutes remplies ; si l'une des conditions n'est pas vérifiée, la clause sera considérée comme fausse**.

Ici, nous avons deux conditions liées par l'opérateur **&&** : les deux conditions doivent être vraies pour que la clause soit remplie !

Cet opérateur vous initie à la notion d'*intersection d'ensembles*. Ici, nous avons deux conditions qui définissent chacune un ensemble :

- `i` < 100 définit un ensemble des nombres inférieurs à 100 (59 ou 45 ou 0 ou -1000000)
- `i` > 50 définit les nombres supérieurs à 50 (58 ou 79 ou 101 ou 1000000).

L'opérateur **&&** permet de faire intersection de ces ensembles. La condition regroupe donc les nombres qui appartiennent à ces deux ensembles, ici les nombres de 51 à 99 inclus.
Réfléchissez bien à l'intervalle que vous voulez définir. Regardez ce code :

Code : Java

```
int i = 58;
 if(i < 100 && i > 100)
   System.out.println("Le nombre est bien dans l'intervalle");

 else
   System.out.println("Le nombre n'est pas dans l'intervalle");
```

Ici, la condition ne sera **JAMAIS** remplie car, personnellement, je ne connais aucun nombre qui est à la fois plus petit que 100 et plus grand !

Par contre, si on remplace les inférieur / supérieur stricts par des inférieur / supérieur ou égal, le seul nombre qui puisse valider la clause est 100, car c'est le seul qui appartienne aux deux ensembles.

Reprenez le code précédent, celui-ci où la condition ne sera jamais remplie...
Maintenant, remplacez l'opérateur `&&` par **||** (pour mémoire, c'est un OU).
À l'exécution du programme et après plusieurs tests de valeur pour `i`, vous pouvez vous apercevoir que tous les nombres remplissent cette condition, **sauf** 100.

Nous vérifions ici si le nombre choisi appartient à **L'UN DES DEUX ensembles ou aux DEUX** . On cherche un nombre strictement inférieur à 100 OU un nombre strictement supérieur à 100 : donc tous les nombres remplissent cette condition, **SAUF 100**.
Et là, si nous remplaçons les inégalités strictes par des inégalités larges, tous les nombres remplissent la condition, car 100 fera partie des deux ensembles.

Ici, un seul opérateur large suffit, car si 100 appartient à l'un des deux ensembles, la condition sera remplie car le nombre doit appartenir à l'un ou l'autre, ou aux deux intervalles !

Retour en haut

La structure switch

Cette instruction est un peu particulière... par sa syntaxe, et son utilisation.
Le **`switch`** est surtout utilisé lorsque nous voulons des conditions "à la carte". Le meilleur exemple se trouve sur le site du Zér0 : ce n'est pas tant la note, mais l'appréciation qui suit qui est définie avec un `switch`.
Prenons l'exemple d'un questionnaire de 4 questions, sur 5 points chacune, qui nous donne 5 notes, et donc 5 appréciations possibles, comme ce qui suit :

- **0/20** : tu peux revoir ce chapitre, petit Zér0 !
- **5/20** : concentre-toi un peu plus... Allez, persévère !
- **10/20** : Je crois que tu as compris l'essentiel ! Viens relire ce chapitre à l'occasion.
- **15/20** : BRAVO ! Voilà une note encourageante pour moi qui essaie de vous apprendre des trucs !
- **20/20** : IL EST VRAIMENT... IL EST VRAIMENT.... IL EST VRAIMENT PHHHEEEEEENOOOOOMMEEEEEENAAAALLLL!

Dans ce genre de cas, on utilise un **`switch`** pour alléger un peu le code, et surtout pour éviter des **`else if`** à répétition.

J'en conviens : nous pouvons très bien arriver au même résultat avec des **`if`** qu'avec un **`switch`**. Mais il faut le voir tout de même.

Je vais vous expliquer comment se construit la syntaxe d'un `switch` ; puis nous allons le mettre en pratique tout de suite après.

Syntaxe

Code : Java

```
switch (/*variable*/)
      {
        case /*argument*/:
          /*action*/;
          break;

        case /*argument*/:
          /*action*/;
          break;

        case /*argument*/:
          /*action*/;
          break;

         default:/*action*/;
      }
```

Cette expression s'exécute comme suit :

- la classe évalue l'expression figurant après le `switch` (ici `/* variable*/`)
- la classe cherche ensuite s'il existe une languette (`case /*valeur possible de la variable */:`), dans le bloc d'instructions, correspondant à la forme de `/*variable*/`
- s'il existe une languette, la requête figurant dans celle-ci sera exécutée
- sinon, on passe à la languette suivante !
- Si aucun cas n'a été trouvé, la classe va exécuter ce qui se trouve dans l'instruction **`default:/*action*/;`**, voyez ceci comme une sécurité.

Notez bien la présence de l'instruction **`break;`**. Celle-ci permet de sortir du **`switch`** si une languette a été trouvée pour le cas concerné. Pour mieux juger de l'utilité de cette instruction, enlevez tous les **`break;`**, et compilez votre programme. Vous verrez le résultat...

Voilà un exemple de `switch` que vous pouvez essayer :

Code : Java

```
int nbre = 5;

      switch (nbre)
      {
        case 1: System.out.println("Ce nombre est tout petit");
          break;

        case 2: System.out.println("Ce nombre est tout petit");
          break;

        case 3: System.out.println("Ce nombre est un peu plus
grand");
          break;

        case 4: System.out.println("Ce nombre est un peu plus
grand");
          break;

        case 5: System.out.println("Ce nombre est la moyenne");
          break;

        case 6: System.out.println("Ce nombre est tout de même
grand");
          break;

        case 7: System.out.println("Ce nombre est grand");
          break;

       default: System.out.println("Ce nombre est très grand,
puisqu'il est compris entre 8 et 10");

      }
```

Ici, vous devriez commencer à voir l'intérêt de **l'indentation** ==> je crois que je l'aurai assez dit...

Si vous avez essayé ce programme en enlevant l'instruction **`break;`**, vous avez dû vous rendre compte que le `switch` exécute le code contenu dans le **`case 5:`** mais aussi dans tous ceux qui suivent !
L'instruction `break;` permet de sortir de l'opération en cours. Dans notre cas, on sort de l'instruction `switch`, mais vous verrez une autre utilisation au chapitre suivant.

L'instruction **`switch`** ne prend que des entiers ou des caractères en paramètre... C'était important de le dire.

Je pense que c'est assez clair ! Vous pouvez voir le même résultat lorsque vous faites des "scores" différents dans vos QCM. Surtout, pensez bien à l'instruction **`break;`**, et aussi à vos **`;`**.

Si tout le monde suit, voyons à quoi ressemble les conditions ternaires !
Retour en haut

La condition ternaire

Celle-ci est un peu particulière mais très pratique.
Avec elle, vous pourrez condenser certaines parties de code, mais attention à ne pas en abuser sinon votre code sera indigeste.

La particularité des conditions ternaires réside dans le fait que trois opérandes (variable ou constante) sont mises en jeu mais aussi que ces conditions sont employées pour affecter des données dans une variable. Voici à quoi ressemble la structure de ce type de condition :

Code : Java

```
int x = 10, y = 20;
int max = (x < y) ? y : x ; //Maintenant max vaut 20
```

Décortiquons ce qu'il se passe :

- Nous cherchons à affecter une valeur à notre variable **max**, mais de l'autre côté de l'opérateur d'affectation se trouve une condition ternaire...
- Ce qui se trouve entre parenthèses est évalué : est-ce que **x** est plus petit que **y**, donc deux cas de figure se profilent à l'horizon :
  - Si la condition renvoie `true` (vrai), donc qu'elle est vérifiée, la valeur qui se trouve après le `?` sera affectée.
  - Sinon, la valeur se trouvant après le symbole `:` sera affectée.
- L'affectation est faite, vous pouvez utiliser votre variable max.

Attention : La condition que vous évaluez doit retourner soit vrai soit faux !

Pour vous faire voir l'utilité de ce genre d'instruction, voilà à quoi pourrait ressembler un code qui fait exactement la même chose que l'exemple que je vous ai fourni :

Code : Java

```
int x = 10, y = 20, max = 0;
if(x < y)
  max = y;
else
  max = x;
```

Vous pouvez aussi faire des calculs (ou autre chose) avant d'affecter les valeurs, donc ce code fonctionne :

Code : Java

```
int x = 10, y = 20;
int max = (x < y) ? y * 2 : x * 2 ; //Ici max vaut 2 * 20 soit 40
```

J'espère que vous y voyez plus clair...
Cependant, vous devez savoir autre chose, comme je vous l'ai dit lorsque vous avez lu le chapitre sur les opérateurs, vous pouvez utiliser le modulo pour savoir si un nombre est pair ou impair. Avec le code suivant, vous allez voir que la variable que nous souhaitons affecter n'a pas de lien avec la condition présente dans l'instruction ternaire.

Pour preuve nous allons affecter un `String` grâce à une condition sur deux `int` :

Code : Java

```
int x = 10;
String type = (x % 2 == 0) ? "C' est pair" : "C' est impair" ; //Ici type vaut "C' est pair"
x = 9;
type = (x % 2 == 0) ? "C' est pair" : "C' est impair" ; //Ici type vaut "C' est impair"
```

Avant de vous laisser, vous ne devez pas oublier que **la valeur que vous allez affecter à votre variable DOIT ETRE DU MEME TYPE QUE VOTRE VARIABLE ! !**

Vous devez aussi savoir que rien ne vous empêche de mettre une condition ternaire dans une condition ternaire :

Code : Java

```
int x = 10, y = 20;
int max = (x < y) ? (y < 10) ? y % 10 : y * 2 : x ; // max vaut 40
//Pas très facile à lire...
//Vous pouvez entourer votre deuxième instruction ternaire avec des parenthèses pour mieux voir
max = (x < y) ? ((y < 10) ? y % 10 : y * 2) : x ; // max vaut 40
```

Je vous propose maintenant un petit QCM de derrière les fagots...

Retour en haut

Je vous conseille d'utiliser ce que vous venez de voir avec ce que vous avez appris lors du chapitre précédent ! Vous verrez tout de suite qu'il est assez pénible de lancer l'application à chaque fois.
Rappelez-vous de ce que je vous ai dit sur l'exécution séquentielle d'un programme. Une fois arrivé à la fin, le programme s'arrête...

Si seulement on pouvait répéter des morceaux de codes... Ah ! Mais c'est ce que nous verrons dans le chapitre suivant...
Je crois que vous êtes d'attaque pour le prochain chapitre : **les boucles** !

GERONIMOOOOOO !

Retour en haut

Les boucles

Autre chapitre important en Java : les boucles !

Oui, mais ...c'est quoi, ces fameuses boucles ?

Une boucle est une instruction qui permet de faire plusieurs fois la même chose...
Très pratique lors de protocoles lourds en exécution. Comme lorsque nous demandons si un utilisateur souhaite refaire quelque chose ou non. Ou encore, lorsque nous devons trier des données dans un tableau... Eh bien, dans ce genre de cas, on se sert d'une boucle !

En gros, en français ça donnerait :
tant que nombre de lignes est inférieur à 100,
alors fais ceci, ou cela, mais n'oublie pas de faire ça...

Ne vous inquiétez pas : ce chapitre est facile à digérer.
Bon appétit !

La boucle while

Décortiquons précisément ce qui se passe dans une boucle.
Pour cela, nous allons voir comment elle se forme.

Une boucle commence par une déclaration. Ici : **`while`** qui veut dire, à peu de chose près, **TANT QUE**
Ensuite, nous avons une condition. C'est elle qui permet à la boucle de s'arrêter. Une boucle n'est pratique que si nous pouvons la contrôler, et donc lui faire répéter une instruction un certain nombre de fois. C'est à ça que servent les conditions.
Puis nous avons l'instruction. C'est ce que va répéter notre boucle ! Dans une boucle, il peut y avoir une ou plusieurs instructions.

Remarque : il peut même y avoir des boucles, dans une boucle...

À ce stade, la boucle va tester la condition, et en fonction de celle-ci, va recommencer ou s'arrêter.

Un exemple concret est toujours le bienvenu... Voici comment une boucle de type **`while`** se fait en Java.

D'abord, réfléchissons au "comment notre boucle va travailler". Pour cela, il faut déterminer notre exemple.
Nous allons afficher "bonjour", <un prénom> que vous allez taper à l'écran, puis vous demanderez si vous voulez recommencer.
Pour réussir ceci, il nous faut une variable pour recevoir le prénom, donc de type **`String`**, et une variable pour récupérer votre réponse et là, plusieurs choix s'offrent à nous : soit un caractère, soit une chaîne de caractères, soit un entier. Ici, nous prendrons une variable de type **`char`**.
Et c'est parti !

Code : Java

```
//Une variable vide
String prenom;
// On initialise celle-ci à O pour oui !
char reponse = 'O';
//Notre objet Scanner, n'oubliez pas l'import de java.util.Scanner
Scanner sc = new Scanner(System.in);
//Tant que la réponse donnée est égale à oui
while (reponse == 'O')
{
        //On affiche une instruction
        System.out.println("Donnez un prénom : ");
        //On récupère le prénom saisi
        prenom = sc.nextLine();
        // On affiche notre phrase avec le prénom
        System.out.println("Bonjour " +prenom+ " comment vas-tu ?");
        //On demande si la personne veut faire un autre essai
        System.out.println("Voulez-vous réessayer ?(O/N)");
        //On récupère la réponse de l'utilisateur
        reponse = sc.nextLine().charAt(0);
}

System.out.println("Au revoir...");
//Fin de la boucle
```

Avant tout, vous avez dû cligner des yeux sur cette ligne "**`reponse = sc.nextLine().charAt(0);`**". Rappelez-vous comment on récupère un **`char`** avec l'objet **`Scanner`**. Nous devons récupérer un objet **`String`** et ensuite prendre le premier caractère de celui-ci ! Eh bien cette syntaxe est une contraction de ce que je vous avais fait voir auparavant .

Cette façon de faire permet d'économiser une variable et donc de la mémoire ! Ici **`sc.nextLine()`** renvoie un `String` sur lequel on applique tout de suite la méthode qui retourne le premier caractère de la chaîne (**`charAt(0)`**), nous pouvons donc, sans crainte, initialiser une variable de type **`char`** avec le résultat obtenu.

Détaillons un peu ce qui ce passe.
Dans un premier temps, nous avons déclaré et initialisé nos variables.
Ensuite, la boucle teste la condition qui nous dit : *"Tant que la variable **`reponse`** contient OUI, on exécute la boucle"*. Celle-ci contient bien la chaîne OUI, donc nous entrons dans la boucle. Rappelez-vous qu'on teste l'égalité des conditions avec un

double égal ==.
Puis c'est l'exécution des instructions, dans l'ordre où elles apparaissent dans la boucle.
Et à la fin, c'est-à-dire à l'accolade fermante de la boucle, le compilateur retourne au début de la boucle.

Cette boucle n'est exécutée que lorsque la condition est remplie : ici, nous avons initialisé la variable "reponse" à 'O' pour que la boucle se fasse. Si nous ne l'avions pas fait, nous n'y serions jamais rentrés. Normal, puisque nous testons la condition avant de rentrer dans la boucle !

Voilà. C'est pas mal, mais il y a un petit problème, dans ce programme... Vous ne voyez pas ? Oh ! Je suis sûr qu'il y a des petits malins qui se sont amusés à mettre autre chose que 'O' ou 'N' en "reponse". Ces petits filous-là ont dû remarquer que nous sortons de la boucle si on tape autre chose que 'O'... Essayez de trouver comment pallier à ce problème....
Il faudrait forcer les utilisateurs à ne tapez que 'OUI' ou 'NON'... Mais non, pas en leur mettant un couteau sous la gorge, bande de barbares !

Avec une boucle !

Comment faire ? C'est très simple, vous allez voir ! Il suffit, comme je l'ai dit plus haut, de forcer les utilisateurs à rentrer soit 'NON'soit 'OUI' ! Avec un `while` ! Mais cela sous-entend de réinitialiser notre variable `reponse` à '' (caractère vide).
Nous allons utiliser ici la méthode. Occupons-nous de la condition de notre boucle, maintenant. Il faut répéter la phase "Voulez-vous réessayer ?" tant que la "reponse" donnée n'est pas 'OUI' et 'NON' : voilà, tout y est.
On appelle ce genre de condition des ***conditions multiples***. Vous avez vu les opérateurs logiques au chapitre 2, normalement... (et == `&&`) Nous en reparlerons dans le prochain chapitre...

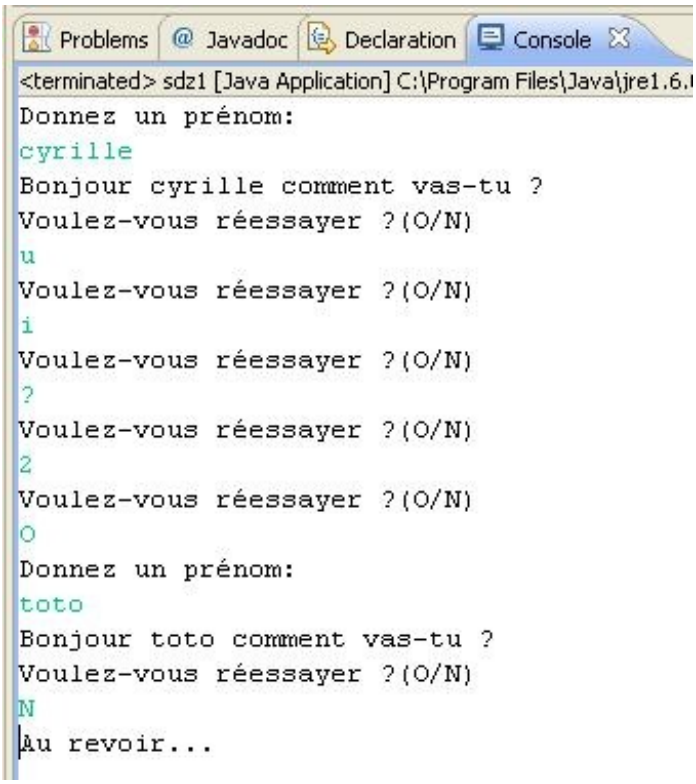
Voici notre programme dans son intégralité :

Code : Java

```
//Une variable vide
String prenom;
// On initialise celle-ci à O pour oui !
char reponse = 'O';
//Notre objet Scanner, n'oubliez pas l' import de java.util.Scanner
Scanner sc = new Scanner(System.in);
//Tant que la réponse donnée est égale à oui
while (reponse == 'O')
{
   //On affiche une instruction
   System.out.println("Donnez un prénom: ");
   //On récupère le prénom saisi
   prenom = sc.nextLine();
   // On affiche notre phrase avec le prénom
   System.out.println("Bonjour " +prenom+ " comment vas-tu ?");

   //réinitialisation de la variable réponse.
   //Sans ça, nous n' entrions pas dans la deuxième boucle
   reponse = ' ';

   //tant que la réponse n'est pas O ou N, on repose la question
   while(reponse != 'O' && reponse != 'N')
   {
       //On demande si la personne veut faire un autre essai
       System.out.println("Voulez-vous réessayer ?(O/N)");
       //On récupère la réponse de l'utilisateur
       reponse = sc.nextLine().charAt(0);
   }
}
//Fin de la boucle
System.out.println("Au revoir...");
```

Je vous le répète une dernière fois : **PENSEZ A L'INDENTATION !**

Vous pouvez tester ce code (c'est d'ailleurs vivement conseillé) : vous verrez que si vous ne rentrez pas la bonne lettre, le programme vous demandera sans cesse votre réponse !



Attention à bien écrire vos **conditions**, et à bien vérifier vos **variables** dans vos `while`, et dans toutes vos boucles en général. Sinon c'est le drame ! Essayez le programme précédent sans la réinitialisation de la variable **`reponse`**, et vous verrez le résultat... on ne rentre jamais dans la 2° boucle, car "`reponse`" = '`O`' (initialisé au début du programme). Là, vous ne pourrez jamais changer sa valeur... donc le programme ne s'arrêtera jamais ! On appelle ça une **boucle infinie**.

Voilà un autre exemple de boucle infinie, encore plus flagrante :

Code : Java

```
int a = 1, b = 15;
while (a < b)
{
        System.out.println("coucou " +a+ " fois !!");
}
```

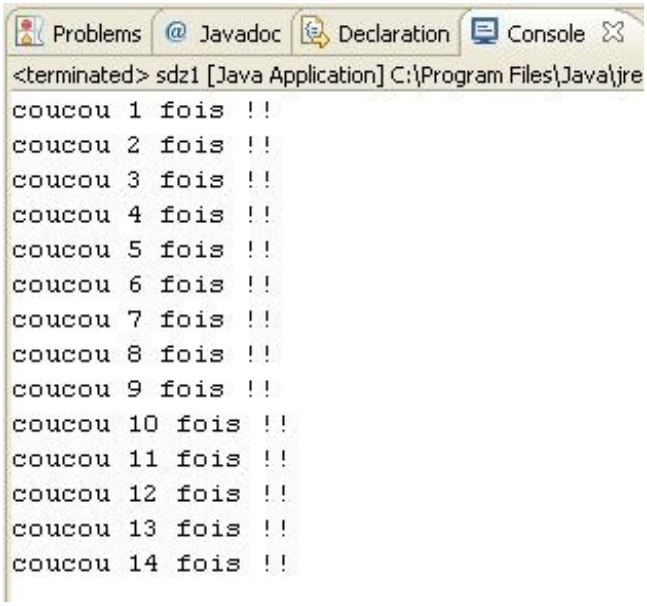
Si vous lancez ce programme, vous allez voir une quantité astronomique de **`coucou 1 fois !!`**, car, dans cette condition, **`a`** sera toujours inférieur à **`b`**.
Si nous voulions faire en sorte que ce programme fonctionne comme il faut, nous aurions dû rajouter une **instruction** dans le **bloc d'instructions** de notre **`while`**, pour changer la valeur de **`a`** à chaque tour de boucle...

Comme ceci :

Code : Java

```
int a = 1, b = 15;
while (a < b)
{
        System.out.println("coucou " +a+ " fois !!");
        a++;
}
```

Ce qui nous donnerait cela :



Notre **a** a bien augmenté de 1 à chaque tour. Et si vous me dites que vous n'avez jamais vu **a++;** je vous renvoie illico au second chapitre ==> sous chapitre 3 !

Qu'en dites-vous ? Pas trop mal, non ? Je dirais même bien !

Une petite astuce : lorsque vous n'avez qu'une instruction dans votre boucle, vous pouvez enlever les accolades { }, celles-ci deviennent superflues, tout comme les instructions **if**, **else if** ou **else**.

Vous auriez pu aussi utiliser cette syntaxe :

Code : Java

```
int a = 1, b = 15;
while (a++ < b)
   System.out.println("coucou " +a+ " fois !!");
```

Ici, l'opérateur '++' n'agit qu'après avoir évalué 'a'. Ce qui veut dire que l'effet de l'opérateur '++' (qui est une incrémentation) n'est perçu qu'une instruction après. Par contre, testez ce code :

Code : Java

```
int a = 1, b = 15;
while (++a < b)
   System.out.println("coucou " +a+ " fois !!");
```

Vous devez remarquer qu'il y a un tour de boucle en moins !

Eh bien, avec cette syntaxe, l'incrémentation est immédiate. C'est-à-dire que la boucle incrémente la variable **a** et, seulement après avoir fait cela, elle teste la condition !

Vous avez dû remarquer aussi que notre instruction n'affiche plus "**coucou 1 fois !!**". Cela à cause de l'incrément dans la condition. Au premier tour de boucle, on entre dans la condition et, quelque soit l'ordre d'incrémentation, à la fin du premier test de condition, **a vaut 2**. **Donc réfléchissez bien à vos**

conditions de boucles !

Avant d'aller voir un autre type de boucle, j'insiste sur le fait que vous devez **bien réfléchir à vos conditions, ainsi qu'à vos variables, avant de lancer une boucle**, sous peine de ne jamais y rentrer, ou comme on l'a vu, de faire une boucle infinie !
Bon, continuons avec la boucle `do{...}while()`.
Retour en haut

La boucle do....while

Vu que je viens de vous expliquer comment marche une boucle **`while`**, je ne vous expliquerai que très brièvement la boucle **`do... while`**.

Euh... t'es sûr de ton coup, là?

Bien entendu. En fait, ces deux boucles ne sont pas cousines, mais plutôt frangines (soeurs, si vous préférez...). Dans le fonctionnement, elles sont identiques, à deux détails près. Soeurs, mais pas jumelles, quoi...

Première différence

La boucle `do... while` s'exécutera **au moins une fois**, contrairement à sa soeur. C'est-à-dire que la phase de test de la condition se fait à la fin. Car la condition se met après le `while`.

Deuxième différence

Différence de syntaxe. Et elle se situe après la condition du `while`.
Exemple :

Code : Java

```
do{
        blablablablablablablabla
}while(a < b);
```

Vous voyez la différence ? Oui ? Non ?

Il y a un **;** après le `while`. C'est tout ! Par contre, ne l'oubliez pas, sinon le programme ne compilera pas.

Mis à part ces deux éléments, ces boucles fonctionnent exactement de la même manière. D'ailleurs, nous allons refaire les deux programmes de la boucle **`while`** ci-dessus, avec une boucle **`do... while`**. C'est parti !

Code : Java

```
//une variable vide !
String prenom = new String();
//pas besoin d'initialiser la variable car on entre au moins une
fois dans la boucle !
char reponse = ' ';

//Notre objet Scanner, n'oubliez pas l'import de java.util.Scanner
Scanner sc = new Scanner(System.in);

do{
        System.out.println("Donnez un prénom : ");
        prenom = sc.nextLine();
        System.out.println("Bonjour " +prenom+ ", comment vas-tu ?
");
        System.out.println("Voulez-vous réessayer ?(O/N)");
        reponse = sc.nextLine().charAt(0);
}while (reponse == 'O');

System.out.println("Au revoir...");
```

Et faites-moi confiance : ça marche ! Mais toujours le même problème de réponse... Voici donc le code complet :

Code : Java

```
        //une variable vide !
        String prenom = new String();
        //pas besoin d'initialiser la variable car on entre au moins une
        fois dans la boucle !
        char reponse = ' ';

        //Notre objet Scanner, n'oubliez pas l'import de java.util.Scanner
        Scanner sc = new Scanner(System.in);

        do{
                System.out.println("Donnez un prénom : ");
                prenom = sc.nextLine();
                System.out.println("Bonjour " +prenom+ ", comment vas-tu ?
        ");

                do{
                        System.out.println("Voulez-vous réessayer ? (O/N)");
                        reponse = sc.nextLine().charAt(0);
                }while(reponse != 'O' && reponse != 'N');

        }while (reponse == 'O');

        System.out.println("Au revoir...");
```

Vous voyez donc que ce code ressemble beaucoup à celui utilisé avec la boucle `while`, mais avec une petite subtilité. Avec celui-ci, plus besoin de réinitialiser la variable **`reponse`**, puisque, de toute manière, la boucle s'exécutera au moins une fois !

Normalement, vous devriez avoir compris du premier coup ! On va pouvoir se lancer sur la dernière boucle : la boucle **`for`**.
Retour en haut

La boucle for

Cette boucle est un peu particulière, puisque qu'elle prend tous ses attributs dans sa condition, et agit en conséquence. Je m'explique : jusqu'ici, nous avions fait des boucles avec :

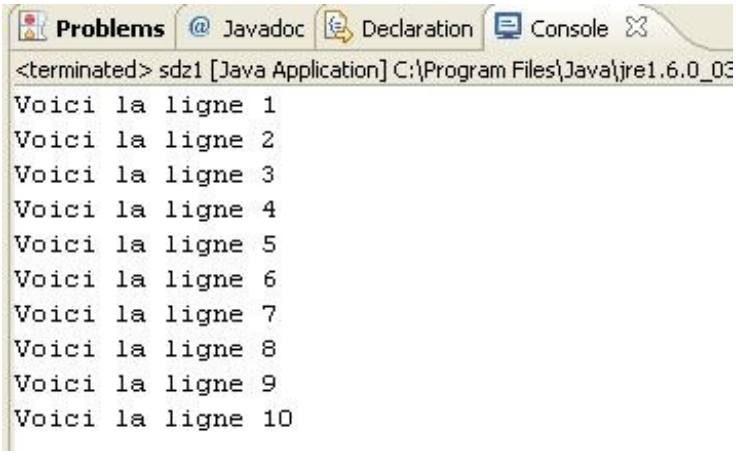
- déclaration d'une variable avant la boucle
- initialisation de cette variable
- incrémentation de celle-ci dans la boucle.

Eh bien on met tout ça dans la condition de la boucle **`for`**, et c'est tout. Mais je sais bien qu'un long discours ne vaut pas un exemple, alors voilà une boucle **`for`** sous vos yeux ébahis :

Code : Java

```
for(int i = 1; i <= 10; i++)
{
        System.out.println("Voici la ligne "+i);
}
```

Et ça nous donne :



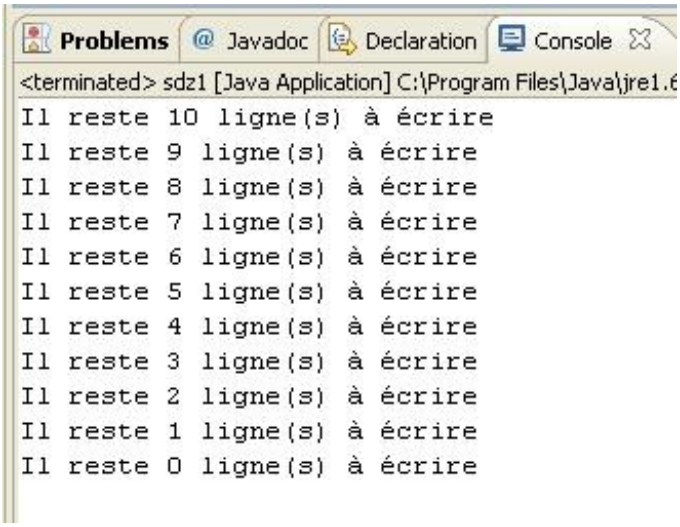
Vous aurez sûrement remarqué la présence des ';' dans la condition pour la séparation des champs. Et là, ne les oubliez surtout pas, sinon, le programme ne compilera pas.

Je vous fais la même remarque que pour la boucle **`while`** concernant les accolades...
Nous pouvons aussi tourner la boucle dans le sens inverse. C'est-à-dire qu'au lieu de partir de zéro pour aller à 10, nous allons aller de 10 pour atteindre 0. Comme ceci :

Code : Java

```
for(int i = 10; i >= 0; i--)
        System.out.println("Il reste "+i+" ligne(s) à écrire");
```

Ce qui nous donne :



Bien entendu, ces structures servent essentiellement à répéter des instructions rébarbatives ; mais elles servent aussi à faire des recherches dans certains cas de figure, ce que nous aborderons dans un prochain chapitre.

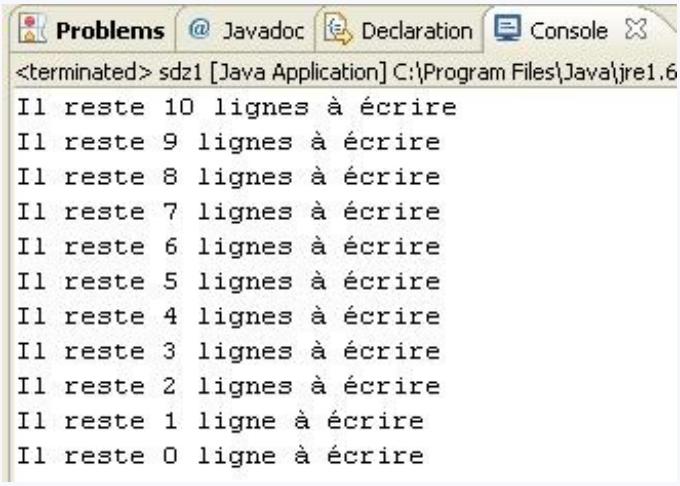
Bon : vu que nous sommes de bons ZérOs et que nous n'aimons pas les fautes d'orthographe, nous voulons mettre "**ligne**" au pluriel lorsqu'il nous en reste plusieurs à écrire, et au singulier lorsqu'il nous en reste 1 ou moins ! Il va de soit que nous allons utiliser les conditions pour réussir ce tour de force.
Je vous laisse réfléchir .

Secret (cliquez pour afficher)

Code : Java

```
for(int i = 10; i >= 0; i--)
{
        if(i > 1)
                System.out.println("Il reste "+i+" lignes à
écrire");
        else
                System.out.println("Il reste "+i+" ligne à
écrire");
}
```

Et le résultat de ce code :



Il existe une autre syntaxe de boucle **`for`** depuis le JDK 1.5, celle-ci se rapprocherait d'une boucle **`foreach`** présente dans d'autres langages (PHP, C#...). Nous verrons celle-ci lorsque nous aborderons les tableaux.

Un petit détail, tout de même... Ici, nous avons utilisé un entier bien défini pour gérer nos boucles, tel que 0 ou 10. Nous pouvons tout aussi bien faire les mêmes boucles avec une variable en guise d'attribut. Là, vous commencez à apercevoir leur intérêt. Ne vous inquiétez pas : vous allez voir tout ceci plus tard. Et plus précisément dans le chapitre sur **les tableaux**, qui arrive à pas de loup !

Retour en haut

Et voilà : la boucle est bouclée !!

Normalement, vous n'aurez pas besoin de digestif pour ce chapitre ! Mais on ne sait jamais...
Je crois qu'il est temps pour un petit TP !

Retour en haut

TP n°1 : un tableau de conversion Celsius - Fahrenheit !

Voilà un très bon petit TP qui va mettre en oeuvre tout ce que vous avez vu jusqu'ici. Mais vraiment **tout !**
Accrochez-vous, car là je vais vous demander de penser à des tonnes de choses, et vous serez tout seuls. Lâchés dans la nature... non, je plaisante. Oui, je sais, je déconne beaucoup, tout de même !
Mais je pense que nous apprendrons mieux dans la bonne humeur.

Bon : trêve de bavardage, au boulot.

Élaboration

Euh... Avant de foncer têtes baissées sur Eclipse, et commencer à coder, nous allons d'abord essayer de structurer notre futur programme. En plus, je ne vous ai même pas dit ce que j'attendais de vous...

Cahier des charges

Alors je veux :

- un code qui puisse se **répéter** autant de fois que nous le souhaitons
- que le code demande à quelle **température nous commençons** la conversion
- la **température de fin** de conversion
- le **pas** de conversion => c'est-à-dire par degré, ou tous les 2 degrés, ou les x degrés
- qu'on vérifie **les cas impossibles** : comme *un pas plus grand que l'intervalle de température*, ou *une température d'arrivée plus petite que la température de départ*
- qu'on demande à l'utilisateur **s'il est prêt**, ou si son imprimante est prête... enfin ce que vous voulez, ici
- **s'il est prêt, que l'on affiche** les conversions sous forme de **tableau visuel**
- qu'il n'y ait pas de décalage pour les différentes valeurs => **tout doit être parfaitement aligné**
- enfin, que l'on demande à l'utilisateur s'il veut faire une nouvelle conversion, donc **revenir au début, s'il le souhaite** !

Je vous avais prévenus que je serais exigeant ! Mais croyez-moi, vous êtes capables de le faire.
Je sais que vous y arriverez !

Élaboration

Comme je vous l'ai dit, essayez de réfléchir sur papier avant... Ce qu'il vous faut comme nombre de variables, les types de variables, comment va se dérouler le programme, les conditions et les boucles utilisées...

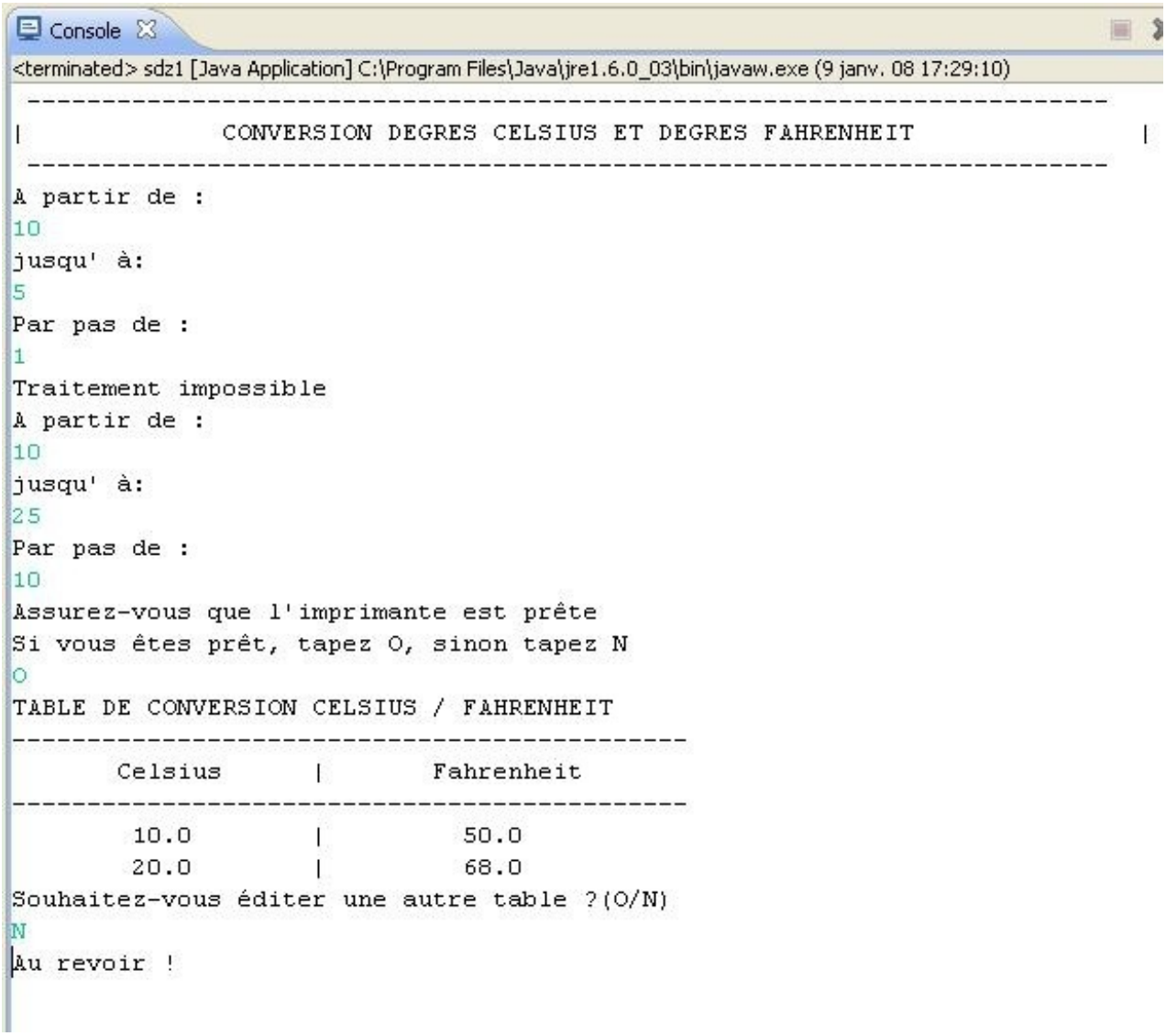
Pour info, voici la formule de conversion pour passer des degrés Celsius en degrés Fahrenheit :

F = 9/5 * C + 32

Je vais vous aiguiller un peu :

- pour ce genre de calcul, utilisez des variables de type `double`
- faites attention à la priorité des opérations
- de simples `if... else` suffisent ici : pas besoin d'un `switch`.

Voici un aperçu de ce que je vous demande :

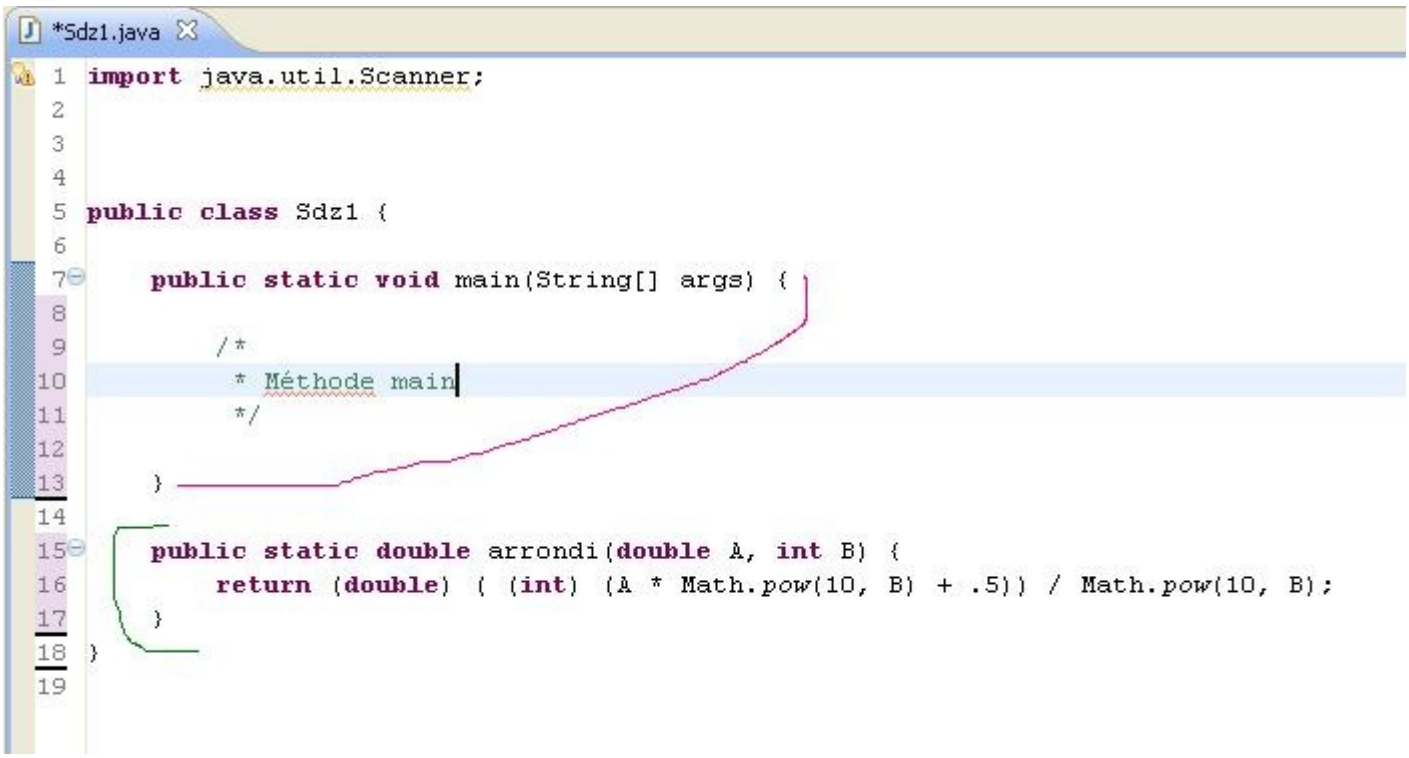


Vous voyez bien que tous mes chiffres sont alignés, malgré leur taille. Lors de l'affichage, il faudra donc utiliser une condition en fonction de la taille des chiffres (`if Celsius < 100){.....} else{.....}`).
Je vais également vous donner une fonction toute faite, qui vous permettra d'arrondir vos résultats. Je vous expliquerai le fonctionnement des fonctions exactement 2 chapitres plus loin. Mais en attendant, c'est facultatif. Vous pouvez très bien ne pas vous en servir. Pour ceux qui souhaitent tout de même l'utiliser, la voici :

Code : Java

```
public static double arrondi(double A, int B) {
     return (double) ( (int) (A * Math.pow(10, B) + .5)) /
Math.pow(10, B);
}
```

Elle est à placer entre les deux accolades fermantes de votre classe, comme ceci :



Vous pouvez l'utiliser de cette manière : imaginez que vous avez la variable **fahren** à arrondir, et que le résultat obtenu soit enregistré dans une variable **arrondFahren**, vous procéderez comme suit :

Code : Java

```
arrondFahren = arrondi(fahren, 1); // pour un chiffre après la virgule
arrondFahren = arrondi(fahren, 2);// pour 2 chiffres après la virgule... etc.
```

Bon : Je vous ai déjà assez aidés ! Place à la conception.

Retour en haut

Conception

Dernière recommandation

Essayez de bien **INDENTER** votre code ! Prenez votre temps. Essayez de penser à tous les cas de figures...

Maintenant à vos papiers, crayons, neurones et claviers, ...et bon courage !
Retour en haut

Correction

STOP !!! C'est fini ! Passons maintenant à la correction de ce premier TP.
Ça va ? Pas trop mal à la tête ? Je me doute qu'il a dû y avoir quelques tubes d'aspirine d'utilisés...
Mais vous allez voir qu'en définitive, ce TP n'était pas si compliqué.
Surtout, n'allez pas croire que ma correction est parole d'évangile... Il y avait différentes manières d'obtenir le même résultat. Voici tout de même une des corrections possibles.

Code : Java

```
class Sdz1 {
    public static void main(String[] args) {
       //Notre objet Scanner
Scanner sc = new Scanner(System.in);

//initialisation des variables
        double c, f=0;
int i,j=0;
char reponse=' ';

System.out.println(" ------------------------------------------------
-------------------------");
System.out.println("| CONVERSION DEGRES CELSIUS ET DEGRES FAHRENHEIT
|");
System.out.println(" ------------------------------------------------
-------------------------");

do{//tant que reponse = O//boucle principale

    do{//tant que l'imprimante n'est pas prête//boucle de test pour
savoir si l'utilisateur est prêt

        do {// tant que valeur impossible rentrée

//saisie des valeurs
            System.out.println("A partir de :");//affichage des
directives et récupération des données
            c = sc.nextDouble();

            System.out.println("jusqu' à:");
            i = sc.nextInt();

            System.out.println("Par pas de :");
            j = sc.nextInt();

            if (c > i || j > i || j == 0)
                System.out.println("Traitement impossible");

        }while(c > i || j > i || j == 0);

        do {//tant que la reponse n'est pas O ou N

            System.out.println("Assurez-vous que l'imprimante est
prête");
            System.out.println("Si vous êtes prêt, tapez O, sinon
tapez N");
            //sc.reset();
            reponse = sc.next().charAt(0);

        }while (reponse != 'O' && reponse != 'N');

    }while (reponse == 'N');

// Traitement des valeurs
    System.out.println("TABLE DE CONVERSION CELSIUS / FAHRENHEIT");
    System.out.println("-----------------------------------------
-");
    System.out.println(" Celsius | Fahrenheit ");
    System.out.println("-----------------------------------------
```

```
-");

    do{//tant que l'affichage n'est pas fini, on boucle les données
et les calculs

        f = ((9.0/5.0) * c) + 32.0;
        if (c < 10)//si le Celsius n'a qu'un chiffre, on affiche un
certain nombre d'espaces
            System.out.println(" "+c+" | "+arrondi(f,1));

        else
        {
            if(c < 100)//S'il y a un chiffre en plus, on enlève un
espace blanc...
                System.out.println(" "+c+" | "+arrondi(f,1));
            else
                System.out.println(" "+c+" | "+arrondi(f,1));
        }

        c = c + j;//On incrémente le degré Celsius avec le pas

    }while (c <= i);

    do{

        System.out.println("Souhaitez-vous éditer une autre table ?
(O/N)");
        reponse = sc.next().charAt(0);

    }while(reponse != 'O' && reponse != 'N');

}while(reponse == 'O');

System.out.println("Au revoir !");

//Fin de programme
}

public static double arrondi(double A, int B) {
        return (double) ( (int) (A * Math.pow(10, B) + .5)) /
Math.pow(10, B);
}



}
```

Expliquons un peu ce code

- Tout programme commence par une phase de **déclaration de variable.**
- Nous **affichons le titre** de notre programme.
- Ensuite, vous voyez 3 `do{` consécutifs, correspondant chacun à une condition à vérifier : **le choix de l'utilisateur pour faire une nouvelle conversion, vérification si l'utilisateur est prêt, vérification si les nombres sont cohérents.**
- Nous **affichons** les renseignements à l'écran, et nous **récupérons les saisies clavier** dans une variable.
- Si les chiffres sont incohérents, on **affiche une erreur, et on boucle** sur le début.
- Une boucle pour s'assurer que l'utilisateur rentre bien **O** ou **N**.
- À partir d'ici, les nombres sont cohérents, et l'utilisateur est prêt. Donc on **lance la conversion**.
- On **affiche le squelette** du tableau.
- Une boucle pour **afficher les différents résultats**.
- **Calcul de la conversion**.
- **Selon la taille du chiffre Celsius, on enlève (ou non) un espace à l'affichage**.
- Cette boucle sera répétée **tant que le degré de départ est plus petit que le degré de fin que vous avez choisi**.
- FIN DU PROGRAMME !

Ce programme n'est pas parfait, loin de là... La vocation de celui-ci était de vous faire utiliser ce que vous avez appris et je pense qu'il remplit bien sa fonction.

J'espère que vous avez apprécié ce TP.
Je sais qu'il n'était pas facile, mais avouez-le : il vous a bien fait utiliser tout ce que vous avez vu jusqu'ici !

Retour en haut

Voilà : votre premier TP est fait, et bien fait !!
Je vous conseille de vous reposer un peu, parce que ça a dû fumer dans votre boite crânienne...
Je viens de vous faire afficher un tableau (*rudimentaire à l'affichage*), mais maintenant nous allons travailler avec **des tableaux** en Java ! C'est parti...

Retour en haut

Les tableaux

Comme dans tout langage de programmation qui se respecte, Java travaille aussi avec des tableaux.
Vous verrez que ceux-ci s'avèrent bien pratiques...

Mais un tableau... Qu'est-ce que c'est, au juste ?

Très bonne question. Vous vous doutez bien (je suppose) que les tableaux dont nous parlons n'ont pas grand-chose à voir avec ceux que vous connaissez ! En programmation, un tableau n'est rien d'autre qu'***une variable un peu particulière*** ... nous allons pouvoir lui affecter ***plusieurs valeurs***, rangées de façon séquentielle, ***que nous pourrons appeler grâce à un indice, ou un compteur***, si vous préférez. Il nous suffira de donner l'emplacement du contenu dans notre variable tableau pour la sortir, travailler avec ou encore l'afficher.

Assez bavardé : mettons-nous joyeusement au travail !

Déclarer et initialiser un tableau

Je viens de vous expliquer, grosso-modo, ce qu'est un tableau en programmation. Si maintenant, je vous disais qu'il y a autant de types de tableaux que de types de variables ? Je crois voir quelques gouttes de sueur perler sur vos fronts...
Pas de panique ! Je dirais même que c'est très logique. Comme nous l'avons vu lors du 3° chapitre, une variable d'un type donné ne peut contenir que des éléments de ce type.
Exemple : une variable de type **`int`** ne peut pas recevoir une chaîne de caractères.

Il en est de même pour les tableaux... **un tableau d'entiers ne pourra pas recevoir des chaînes de caractères, et vice versa.**
Voyons tout de suite comment se déclare un tableau :

```
<type du tableau> <nom du tableau> [] = { <contenu du tableau>};
```

La déclaration ressemble beaucoup à celle d'un argument de classe quelconque, si ce n'est la présence des crochets **[]** après le nom de notre tableau, et les accolades {} encadrant l'initialisation de celui-ci.
Dans la pratique, ça nous donnerait quelque chose comme ceci :

Pour un tableau d'entiers

Code : Java

```
int tableauEntier[] = {0,1,2,3,4,5,6,7,8,9};
```

Pour un tableau de double

Code : Java

```
double tableauDouble[] = {0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0};
```

Pour un tableau de caractères

Code : Java

```
char tableauCaractere[] = {'a','b','c','d','e','f','g'};
```

Pour un tableau de chaînes de caractères

Code : Java

```
String tableauChaine[] = {"chaine1", "chaine2", "chaine3" ,
"chaine4"};
```

Vous remarquez bien que la déclaration et l'initialisation d'un tableau se font comme pour une variable normale (*si on peut dire qu'une variable est normale...*).
Nous utilisons des '' pour initialiser un tableau de caractères, des " " pour initialiser un tableau de `String`, etc.

Vous pouvez aussi déclarer un tableau vide !

Attention, votre tableau sera vide mais, il **doit** avoir un nombre de cases défini !

Par exemple, si vous voulez un tableau vide de six entiers :

Code : Java

```
int tableauEntier[] = new int[6];
//ou encore
int[] tableauEntier2 = new int[6];
```

Cette opération est très simple, car vraiment ressemblante à ce que vous faisiez avec vos variables ; je vous propose donc tout de suite de voir une belle variante de ceci : **les tableaux multi-dimensionnels** .

Retour en haut

Les tableaux multi-dimensionnels

Ici, les choses se compliquent un peu. Car un tableau multi-dimensionnel n'est rien d'autre qu'un tableau ayant comme contenu au minimum 2 tableaux... Je me doute bien que cette notion doit en effrayer plus d'un, mais en réalité, elle n'est pas si difficile que ça. Comme tout ce que je vous apprends en général !
Ben oui... Si j'y arrive, vous aussi vous pouvez y arriver ! Alors, on se lance ? GO !

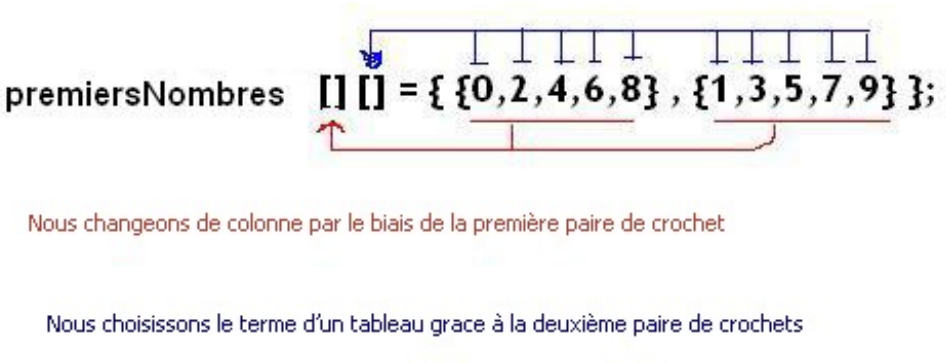
Je ne vais pas vous faire de grand laïus sur ce type de tableau, puisque je pense sincèrement qu'un exemple vous fera beaucoup mieux comprendre le concept. Imaginez un tableau avec deux lignes : la première contiendra les premiers nombres pairs, et le deuxième contiendra les premiers nombres impairs. Ce tableau s'appellera `premiersNombres`. Voilà ce que cela donnerait :

Code : Java

```
int premiersNombres[][] = { {0,2,4,6,8},{1,3,5,7,9} };
```

Nous voyons bien ici les deux *lignes* de notre tableau symbolisées par les doubles crochets **[][]**. Et comme je l'ai dit plus haut, ce genre de tableau n'est rien d'autre que plusieurs tableaux en un. Ainsi, pour passer d'une ligne à l'autre, nous jouerons avec la valeur du premier crochet.
Exemple :
`premiersNombres[0][0]` correspondra au premier élément de la colonne paire.
Et **`premiersNombres[1][0]`** correspondra au premier élément de la colonne impaire.

Allez ! Un petit schéma en guise de synthèse :



Surtout, n'oubliez pas de séparer vos différents tableaux par une ',' et de bien mettre le tout entre accolades. Sinon, c'est le plantage assuré, et de toutes façons, Eclipse n'en voudra pas.

Je pense que vous savez tout ce qu'il y a à savoir sur les tableaux. Maintenant, je vous propose de faire un peu mumuse avec...
Retour en haut

Utiliser et rechercher dans un tableau !

Un tableau simple

Avant d'attaquer, je dois vous dire un truc primordial (vous avez remarqué, c'est écrit en gros et en rouge).

Un tableau, comme ceux que nous avons fait ci-dessus, débute toujours à l'indice 0 !

Je m'explique : prenons l'exemple du tableau de caractères. Si vous voulez afficher la lettre 'a' à l'écran, vous devrez taper cette ligne de code :

Code : Java

```
System.out.println(tableauCaractere[0]);
```

Ce qui signifie tout bêtement qu'un tableau, ayant 4 éléments dans son contenu, aura comme entrées possibles **0, 1, 2 ou 3**. Le 0 correspond au premier élément, le 1 correspond au 2° élément, le 2 correspond au 3° élément, et le 3 correspond au 4° élément.

Une très grande partie des erreurs sur les tableaux sont souvent dues à un mauvais indice dans celui-ci. Donc : prenez garde...

Ce que je vous propose, c'est tout bonnement d'afficher un des tableaux ci-dessus dans son intégralité. Et le premier qui me met ce code-là :

Code : Java

```
char tableauCaractere[] = {'a','b','c','d','e','f','g'};
System.out.println(tableauCaractere[0]);
System.out.println(tableauCaractere[1]);
System.out.println(tableauCaractere[2]);
System.out.println(tableauCaractere[3]);
```

... je l'**ASSASSINE** ! Peut-être pas, quand même... Il y a une manière beaucoup plus *classe*, ou distinguée, mais surtout beaucoup plus pratique d'afficher le contenu d'un tableau.

Voici un parcours de tableau avec une boucle while

Code : Java

```
char tableauCaractere[] = {'a','b','c','d','e','f','g'};
    int i = 0;

    while (i < 4)
    {
            System.out.println("A l'emplacement " + i +" du tableau
nous avons = " +tableauCaractere[i]);
            i++;
    }
```

Même résultat que précédemment, mais avec une boucle for (à utiliser de préférence)

Code : Java

```
char tableauCaractere[] = {'a','b','c','d','e','f','g'};

    for(int i = 0; i < 4; i++)
    {
            System.out.println("A l'emplacement " + i +" du tableau
nous avons = " +tableauCaractere[i]);
    }
```

Euh... Comment fait-on si on ne connaît pas la taille de notre tableau à l'avance ?

Décidément, vous lisez dans mes pensées... J'allais y venir !
En fait, il existe une instruction qui retourne la taille d'un tableau.
Il s'agit de l'instruction **`<mon tableau>.length`**. Notre boucle **`for`** pourrait donc ressembler à ceci :

Code : Java

```
char tableauCaractere[] = {'a','b','c','d','e','f','g'};

    for(int i = 0; i < tableauCaractere.length; i++)
    {
            System.out.println("A l'emplacement " + i +" du tableau
nous avons = " +tableauCaractere[i]);
    }
```

Alors ? Ce n'est pas mieux comme ça ? D'accord, je reformule ma phrase pour ceux qui ont le sens de la contradiction. **C'est mieux comme ça !** Je vais vous donner une preuve que vous ne pourrez pas nier.
Essayez de faire une recherche dans un des tableaux ci-dessus (pas celui contenant des `String`, nous verrons ce cas dans le prochain chapitre). En gros : faites une saisie clavier, et regardez si celle-ci existe dans votre tableau... Dur, dur, sans boucle... COMMENT ÇA, C'EST DUR, MÊME AVEC UNE BOUCLE ?

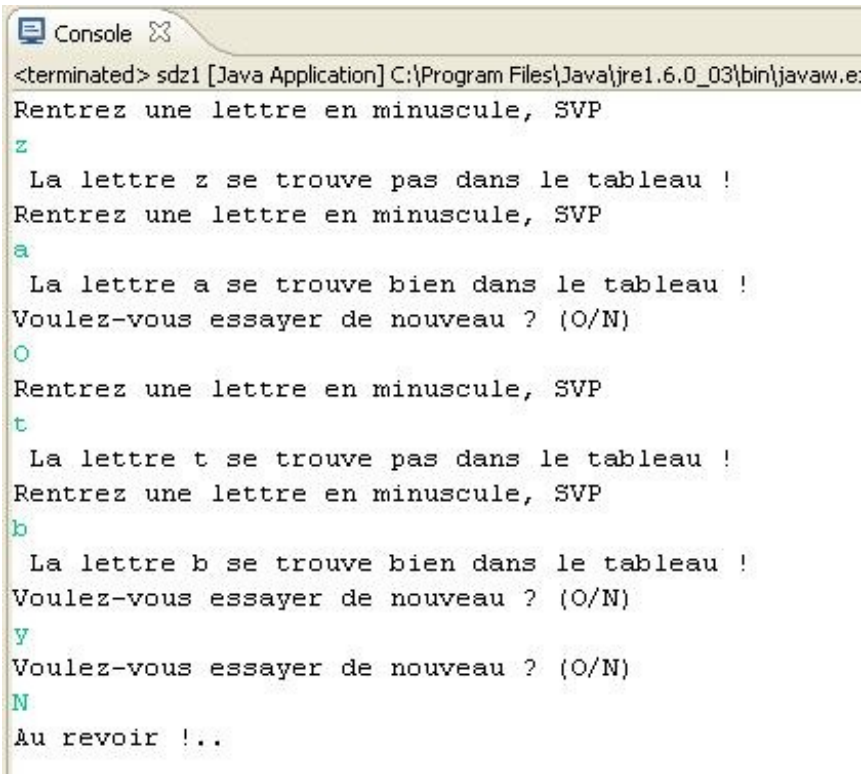
Dans ce cas, je vais vous aider. Gardez la même structure de code permettant de faire plusieurs fois la même action, et ensuite faites une boucle de recherche incluant la saisie clavier, un message si la saisie est trouvée dans le tableau, et un autre message si celle-ci n'est pas trouvée. Ce qui nous donne :

Code : Java

```
char tableauCaractere[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g'};
            int i = 0, emplacement = 0;
            char reponse = ' ',carac = ' ';
            Scanner sc = new Scanner(System.in);

            do {//boucle principale
              do {//on répète cette boucle tant que l'utilisateur
n'a pas rentré une lettre figurant dans le tableau
                i = 0;
                System.out.println("Rentrez une lettre en minuscule,
SVP ");

                carac = sc.nextLine().charAt(0);

                while(i < tableauCaractere.length && carac !=
tableauCaractere[i])//boucle de recherche dans le tableau
                  i++;

                if (i < tableauCaractere.length)//Si i < 7 c'est que
la boucle n'a pas dépassé le nombre de cas du tableau ==> il a
trouvé
                  System.out.println(" La lettre " +carac+ " se
trouve bien dans le tableau !");

                else//sinon
                  System.out.println(" La lettre " +carac+ " ne se
trouve pas dans le tableau !");

              }while(i >= tableauCaractere.length);//tant que la
lettre de l'utilisateur ne correspond pas à une lettre du tableau
```

```
                do{
                  System.out.println("Voulez-vous essayer de nouveau ?
(O/N)");
                  reponse = sc.nextLine().charAt(0);
                }while(reponse != 'N' && reponse != 'O');


              }while (reponse == 'O');

                  System.out.println("Au revoir !..");
```

Ce qui nous donne :



Explicitons un peu ce code, et plus particulièrement la recherche

Dans notre `while`, il y a deux conditions :

==> la première correspond au compteur. Tant que ce compteur est inférieur ou égal au nombre d'éléments du tableau, on incrémente notre compteur pour regarder la valeur suivante. Nous passons ainsi en revue tout ce qui se trouve dans notre tableau. **MAIS** si nous n'avions mis que cette condition, la boucle n'aurait fait que parcourir le tableau, sans voir si le caractère saisi correspond bien à un caractère de notre tableau, d'où la deuxième condition.

==> la deuxième correspond à la comparaison entre le caractère saisi et la recherche dans le tableau. Grâce à ceci, si le caractère saisi se trouve dans le tableau, la boucle prend fin, et donc **`i`** a une valeur inférieure à 7.

À ce stade, notre recherche est terminée. Ensuite, les conditions coulent de source ! Si nous avons trouvé une correspondance entre le caractère saisi et notre tableau, **`i`** aura une valeur inférieure à 7 (je vous rappelle qu'il n'y a que 6 entrées dans notre tableau, puisque nous avons 7 lettres dans celui-ci, et la première entrée a comme indice 0). Dans ce cas, nous affichons un message positif. Et dans le cas contraire, c'est l'instruction du **`else`** qui s'exécutera.

Vous avez dû remarquer la présence d'un **`i = 0;`** dans une boucle. Ceci est PRIMORDIAL, car sinon, lorsque vous reviendrez au début de celle-ci, **`i`** ne vaudra plus 0, mais la dernière valeur qu'il aura eue, après les différentes incrémentations. Si vous faites une nouvelle recherche, vous commencerez par l'indice contenu dans **`i`** ; ce que vous ne voulez pas, puisque vous voulez regarder depuis le début du tableau, donc 0.

Pour bien vous rendre compte de cela, essayez le programme ci-dessus sans cette instruction : vous verrez qu'il n'y a plus de recherche possible, ou même un gros plantage d'Eclipse...

En travaillant avec les tableaux, vous serez confrontés, un jour ou l'autre, au message suivant : **`java.lang.ArrayIndexOutOfBoundsException`**.
Ceci signifie qu'une exception a été levée car vous avez essayé de lire (ou d'écrire dans) une case qui n'a pas été définie dans votre tableau ! Nous verrons les exceptions lorsque nous aborderons la programmation orienté objet.

Exemple

Code : Java

```
String[] str = new String[10];
//L'instruction suivante va déclencher une exception
//car vous essayez d'écrire à la case 11 de votre tableau alors que
celui-ci n'en contient que 10
str[11] = "Une exception";
//De même, le code ci-dessous déclenchera la même exception car
vous essayez de lire
//une case non définie !
String string = str[24];
```

Faites donc bien attention à cela, car il s'agit de l'une des erreurs commises les plus fréquentes.

Un tableau multi-dimensionnel

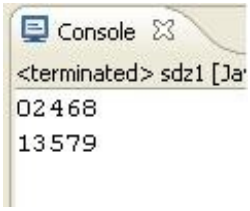
Nous allons travailler sur le tableau bi-dimensionnel vu plus haut.
Le principe est vraiment le même que pour un tableau simple. Mais ici, il n'y a que deux compteurs. Voici un code possible pour afficher les données par ligne, c'est-à-dire l'intégralité du *sous-tableau nombres pairs*, puis le *sous-tableau nombres impairs* :

Avec une boucle while

Code : Java

```
int premiersNombres[][] = { {0,2,4,6,8},{1,3,5,7,9} }, i = 0, j = 0;

   while (i < 2)
   {
     j = 0;
     while(j < 5)
     {
       System.out.print(premiersNombres[i][j]);
       j++;
     }
     System.out.println("");
     i++;
   }
```

Et voilà le résultat :



Je suppose que vous avez remarqué la drôle de déclaration de variable... Vous avez le droit de faire ainsi. Mais seules les variables ayant les crochets **[]** après leur nom seront considérées comme des tableaux, les autres resteront des variables toutes simples.

Détaillons un peu ce code

- Dans un premier temps, on initialise les variables.
- On entre ensuite dans la première boucle (qui se fera deux fois, donc **i** vaut 0 la première fois, et vaudra 1 pendant la deuxième), et on initialise **j** à 0.
- On entre ensuite dans la deuxième boucle, où **j** vaudra successivement 0, 1, 2, 3 et 4 pour afficher le contenu du tableau d'indice 0 (notre premier `i`).
- On sort de cette boucle ; notre **i** est ensuite incrémenté, et passe à 1.
- On reprend le début de la première boucle => initialisation de **j** à 0.
- On rentre dans la deuxième boucle, où le processus est le même que précédemment (mais là, **i** vaut 1).
- Enfin, nous sortons des boucles pour finir le programme.

Ce code affiche donc le contenu des deux tableaux... Encore heureux ! Notez bien que vous devez réinitialiser votre compteur **j** avant la boucle où il est utilisé comme argument, sinon celle-ci ne fonctionnera pas. Et cela, pour la même raison que pour un tableau normal.

Le même résultat avec une boucle for

Code : Java

```
int premiersNombres[][] = { {0,2,4,6,8},{1,3,5,7,9} };

   for(int i = 0; i < 2; i++)
   {

     for(int j = 0; j < 5; j++)
     {
       System.out.print(premiersNombres[i][j]);

     }
     System.out.println("");
   }
```

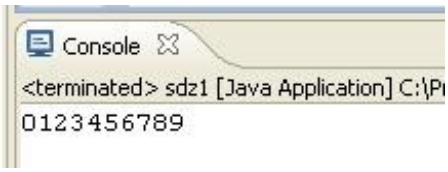
En bonus, voici un petit code qui va vous afficher la suite des nombres dans l'ordre, en piochant tantôt dans le tableau pair, tantôt dans le tableau impair :

Code : Java

```
int premiersNombres[][] = { {0,2,4,6,8},{1,3,5,7,9} };

   for(int i = 0; i < 5; i++)
```

```
        {

          for(int j = 0; j < 2; j++)
          {
            System.out.print(premiersNombres[j][i]);

          }

        }
```

Et voilà :



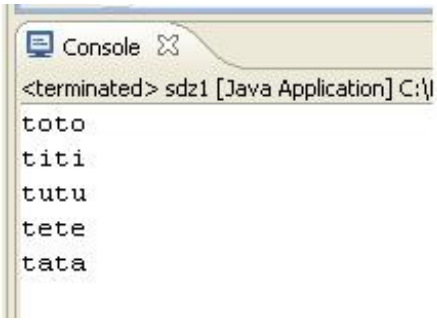
Vous avez remarqué que la différence entre ces deux codes était seulement l'ordre des conditions dans nos boucles...
Tu ne nous avais pas parlé d'une façon de faire une boucle **for** ?

Tout à fait, d'ailleurs la voici :

Code : Java

```
String tab[] = {"toto", "titi", "tutu", "tete", "tata"};

for(String str : tab)
   System.out.println(str);
```

Voici le résultat :



Faites bien attention à ne pas confondre les deux syntaxes ! La boucle **for**, ancienne génération, prend des **;** dans ses instructions, alors que la nouvelle version prend un **:**.

Comme je vous l'avais dit lorsque vous avez vu "***Les boucles***", cette syntaxe se rapproche de la boucle **foreach** présente dans d'autre langages.
Cette syntaxe signifie qu'à chaque tour de boucle, la valeur courante du tableau est mise dans la variable **str**.

Il faut **IMPÉRATIVEMENT** que la variable passée en premier paramètre de la boucle **for** soit de même type que la valeur de retour du tableau (une variable de type **String** pour un tableau de **String**, un **int** pour un tableau d'**int**...)

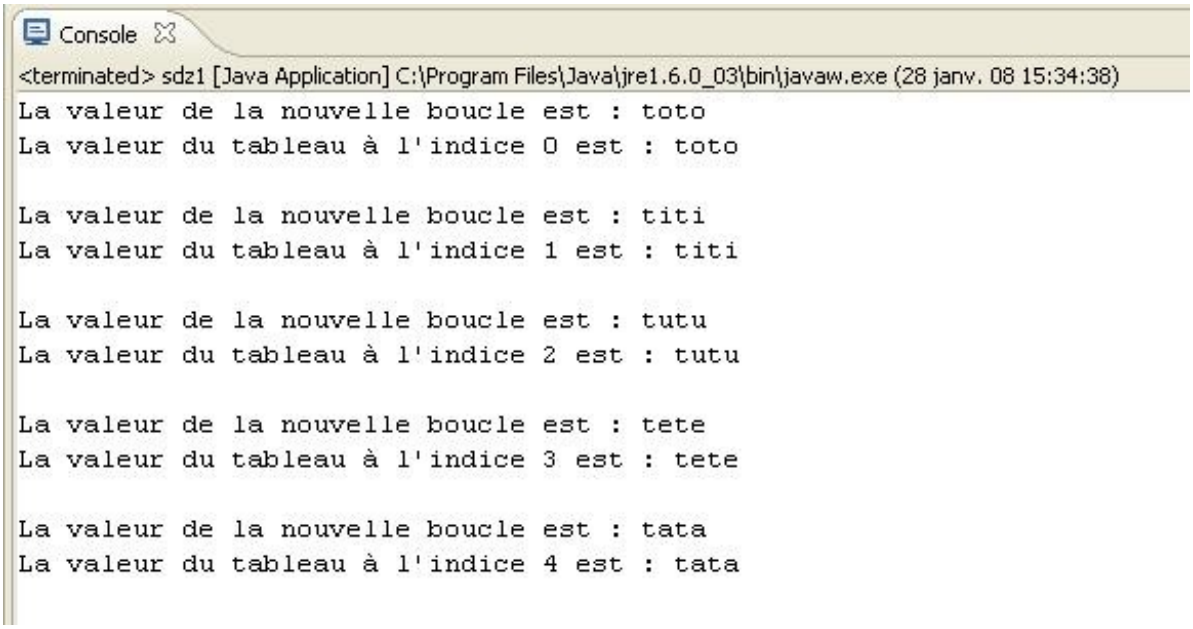
Pour vérifier que les valeurs retournées par la boucle correspondent bien à leurs indices, vous pouvez déclarer un entier (`i`, par exemple) initialisé à 0 et incrémenté à la fin de la boucle.

Allez, c'est mon jour de bonté :

Code : Java

```
String tab[] = {"toto", "titi", "tutu", "tete", "tata"};
int i = 0;

for(String str : tab)
{
   System.out.println("La valeur de la nouvelle boucle est : " +
str);
System.out.println("La valeur du tableau à l'indice " + i +" est : "
+ tab[i] + "\n");
   i++;
}
```

Ce qui nous donne :



Alors ? Convaincus ?
Vous pouvez voir que cette forme de boucle **`for`** est particulièrement adaptée au parcours de tableau !

D'accord, ça a l'air bien comme méthode, mais pour les tableaux à deux dimensions ?

Même si vous devriez trouver la réponse tout seuls, je vais tout de même vous la donner !

Je vous ai dit que la variable en premier paramètre devait être du même type que la valeur de retour du tableau. Dans le cas qui nous intéresse, que va retourner l'instruction de la boucle **`for`** si on utilise un tableau à deux dimensions ? Un **tableau**. Nous devrons donc mettre un tableau, du même type que notre tableau à dimensions, en première instruction de la boucle, et donc faire une deuxième boucle afin de parcourir le résultat obtenu !

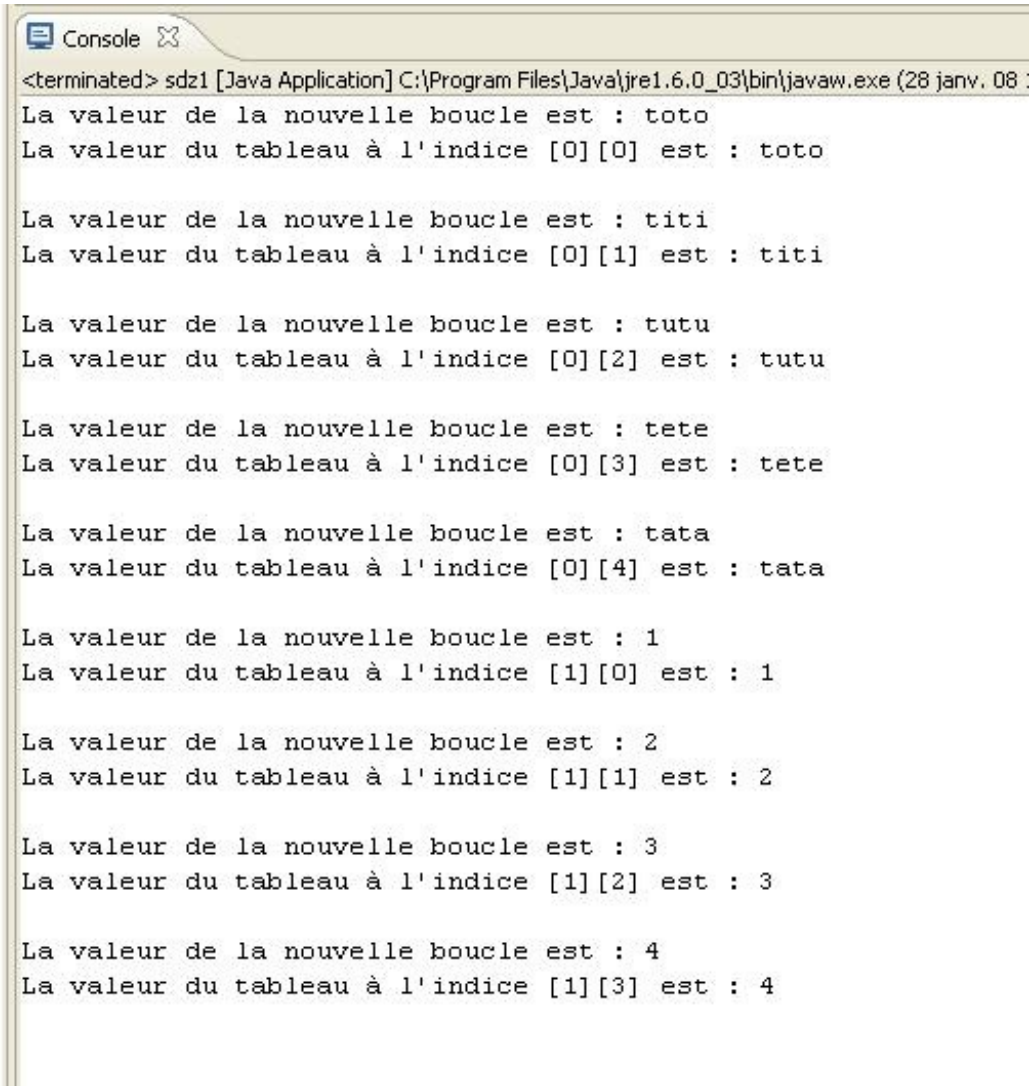
Voici un code qui permet d'afficher un tableau à deux dimensions de façon conventionnelle et selon la nouvelle version du JDK 1.5 :

Code : Java

```
String tab[][] = {{"toto", "titi", "tutu", "tete", "tata"}, {"1",
```

```
        "2", "3", "4"}};
                        int i = 0, j = 0;

                        for(String sousTab[] : tab)
                        {
                           i = 0;
                          for(String str : sousTab)
                          {
                                System.out.println("La valeur de la nouvelle
boucle est : " + str);
                                System.out.println("La valeur du tableau à
l'indice [" + j + "][" +i +"] est : " +  tab[j][i] + "\n");
                            i++;
                          }
                          j++;
                        }
```

Ce qui nous donne :



Vous pouvez donc voir que nous récupérons un tableau au parcours de la première boucle et que nous parcourons ce même tableau afin de récupérer les valeurs de celui-ci dans la deuxième. Simple, non ? En tout cas, je préfère nettement cette syntaxe !

Après, c'est à vous de voir...

Vous êtes maintenant parés pour utiliser les tableaux quels qu'ils soient !
Donc, sans plus attendre, je vous propose un petit QCM de derrière les fagots !

Retour en haut
Encore un chapitre bien dodu, mais je vois que vous vous en sortez comme des pros !

Le chapitre suivant sera l'un des plus fournis en informations ; donc, si vous sentez un peu de fatigue, reposez-vous un peu avant d'attaquer **les méthodes de classe**.
Retour en haut

Les méthodes de classe

Ce chapitre aura pour but de vous faire découvrir ce qu'on appelle, en Java, des méthodes de classe (fonctions en C/C++), de base et très utiles.

Java est un langage 100 % objet. Ceci signifie que tout ce qui fait partie du langage est objet !
Nous approfondirons cette notion lorsque nous ferons de la POO (Programmation Orienté Objet).
Pour le moment, retenez juste que, dans chaque objet, il y a des méthodes qui font des traitements spécifiques. Il s'agit tout bonnement d'une portion de code réutilisable !

C'est ce que nous allons aborder ici.

Mais ce chapitre ne serait pas drôle si nous ne nous amusions pas à en créer une ou deux pour le plaisir...
Et là, vous aurez beaucoup de choses à retenir...

Quelques méthodes bien utiles !

Voici le moment de vous présenter quelques méthodes dont, j'en suis convaincu, vous ne pourrez plus vous passer...

toLowerCase()

Cette méthode permet de transformer toute saisie clavier de type caractère en minuscules. Elle n'a aucun effet sur les nombres, puisqu'ils ne sont pas assujettis à cette contrainte. Vous pouvez donc utiliser cette fonction sur une chaîne de caractères comportant des nombres.
Elle s'utilise comme ceci :

Code : Java

```
String chaine = new String("COUCOU TOUT LE MONDE !"), chaine2 = new
String();
chaine2 = chaine.toLowerCase();//donne "coucou tout le monde !"
```

toUpperCase()

Celle-là est facile, puisqu'il s'agit de l'opposée de la précédente. Elle transforme donc une chaîne de caractères en majuscules. Et s'utilise comme suit :

Code : Java

```
String chaine = new String("coucou coucou"), chaine2 = new String();
chaine2 = chaine.toUpperCase();//donne "COUCOU COUCOU"
```

concat()

Très explicite, celle-là permet de concaténer deux chaînes de caractères.

Code : Java

```
String str1 = new String("Coucou "), str2 = new String("toi !"),
str3 = new String();
str3 = str1.concat(str2);//donne "Coucou toi !"
```

length()

Celle-là permet de donner la longueur d'une chaîne de caractères (en comptant les espaces blancs).

Code : Java

```
String chaine = new String("coucou ! ");
int longueur = 0;
longueur = chaine.length();//donne 9
```

equals()

Permet de voir si deux chaînes de caractères sont identiques. Donc, de faire des tests. C'est avec cette fonction que vous ferez vos tests de conditions, lorsqu'il y aura des `String`. Exemple concret :

Code : Java

```
String str1 = new String("coucou"), str2 = new String("toutou");

if (str1.equals(str2))//Si les deux chaînes sont identiques
        System.out.println("Les deux chaines sont identiques !");

else
        System.out.println("Les deux chaînes sont différentes !");
```

Vous pouvez aussi demander la non vérification de l'égalité... Grâce à l'opérateur de négation... vous vous en souvenez ? Il s'agit de '!'

Ce qui nous donne :

Code : Java

```
String str1 = new String("coucou"), str2 = new String("toutou");

if (!str1.equals(str2))//Si les deux chaînes sont différentes
        System.out.println("Les deux chaines sont différentes !");

else
        System.out.println("Les deux chaînes sont identiques !");
```

Le principe de ce genre de condition fonctionne de la même façon pour les boucles. Et dans l'absolu, cette fonction retourne un booléen. C'est pourquoi nous pouvons utiliser cette fonction dans les tests de condition.

Code : Java

```
String str1 = new String("coucou"), str2 = new String("toutou");
boolean Bok = str1.equals(str2);//ici Bok prendra la valeur false
```

charAt()

Le résultat de cette méthode sera un caractère, car il s'agit d'une méthode d'extraction de caractères, je dirais même d'UN caractère. Elle ne peut s'opérer que sur des `String`! Elle possède la même particularité que les tableaux, c'est-à-dire que, pour cette méthode, le premier caractère sera le numéro 0. Cette méthode prend un entier comme argument.

Code : Java

```
String nbre = new String("1234567");
char carac = ' ';
carac = nbre.charAt(4);//renverra ici le caractère 5
```

substring()

Comme son nom l'indique, elle permet d'extraire une sous-chaîne de caractères d'une chaîne de caractères. Cette méthode prend 2 entiers comme arguments. Le premier définit le début de la sous-chaîne à extraire **inclus**, le deuxième correspond au dernier caractère à extraire **exclus**. Et le premier caractère est aussi le numéro 0.

Code : Java

```
String chaine = new String("la paix niche"), chaine2 = new String();
chaine2 = chaine.substring(3,13);//permet d'extraire "paix niche"
```

indexOf()/lastIndexOf()

indexOf() permet d'explorer une chaîne de caractères depuis son début. **lastIndexOf()** depuis sa fin, mais renvoie l'index depuis le début de la chaine. Elle prend un ***caractère, ou une chaîne de caractères*** comme argument, et renvoie un **int**. Tout comme **charAt()** et **substring()**, le premier caractère est à la place 0. Je crois qu'ici un exemple s'impose, plus encore que pour les autres fonctions :

Code : Java

```
String mot = new String("anticonstitutionnellement");
int n = 0;

n = mot.indexOf('t');       // n vaut 2
n = mot.lastIndexOf('t');         // n vaut 24
n = mot.indexOf("ti");     // n vaut 2
n = mot.lastIndexOf("ti");       // n vaut 12
n = mot.indexOf('x');       // n vaut -1
```

Des méthodes concernant les mathématiques

Ces méthodes utilisent la classe `Math`, présente dans java.lang. Cette classe fait donc partie des fondements du langage et, par conséquent, aucun import particulier n'est nécessaire pour utiliser la classe `Math`.

random()

Cette méthode permet de générer un nombre aléatoire, entre 0 et 1 (elle renvoie donc un `double`). Vous ne voyez pas l'utilité ? Eh bien... Vous vous en rendrez compte lors de notre prochain TP...

Code : Java

```
double X = 0.0;
X = Math.random();//donne un nombre comme 0.0001385746329371058
```

Sinus, cosinus, tangente

Code : Java

```
double sin = Math.sin(120);
double cos = Math.cos(120);
double tan = Math.tan(120);
```

Ces méthodes retournent un nombre de type `double`.

Valeur absolue

Code : Java

```
double abs = Math.abs(-120.25);
```

Exposant

Code : Java

```
double d = 2;
double exp = Math.pow(d, 2);
//Ici on initialise la variable exp avec la valeur de d élevée au
carré
//La méthode pow() prend donc une valeur en premier paramètre
//et un exposant en second
```

Je ne vais pas vous faire un récapitulatif de toutes les méthodes présentes dans Java, sinon, dans 1000 ans, je serais encore derrière mon clavier...

Toutes ces méthodes sont très utiles, croyez-moi. Mais les plus utiles sont encore celles que nous faisons ! C'est tellement mieux quand ça vient de nous...

Retour en haut

Créer et utiliser sa propre méthode !

Reprenons la méthode que je vous avais donnée lors du premier TP. Pour mémoire :

Code : Java

```
public static double arrondi(double A, int B) {
     return (double) ( (int) (A * Math.pow(10, B) + .5)) /
Math.pow(10, B);
}
```

Décortiquons un peu ceci

- Tout d' abord, il y a le mot clé **`public`**. C'est ce qui définit la portée de la méthode. Nous y reviendrons lorsque nous programmerons des objets.
- Ensuite il y a **`static`**. Nous y reviendrons aussi.
- Juste après, nous voyons **`double`**. Il s'agit du type de retour de la méthode. Pour faire simple, ici, notre méthode va renvoyer un `double` !
- Il y a ensuite le **nom de la méthode**. C'est avec ce nom que nous l'appellerons.
- Puis, il y a **les arguments que la méthode prend**. C'est en fait les paramètres dont la méthode a besoin pour travailler. Ici, nous demandons d'arrondir le `double` A avec B chiffres derrière la virgule !
- Et vous pouvez voir, à l'intérieur de la méthode, une instruction **`return`**. C'est cette instruction qui effectue le renvoi de la valeur, ici, un **`double`**.

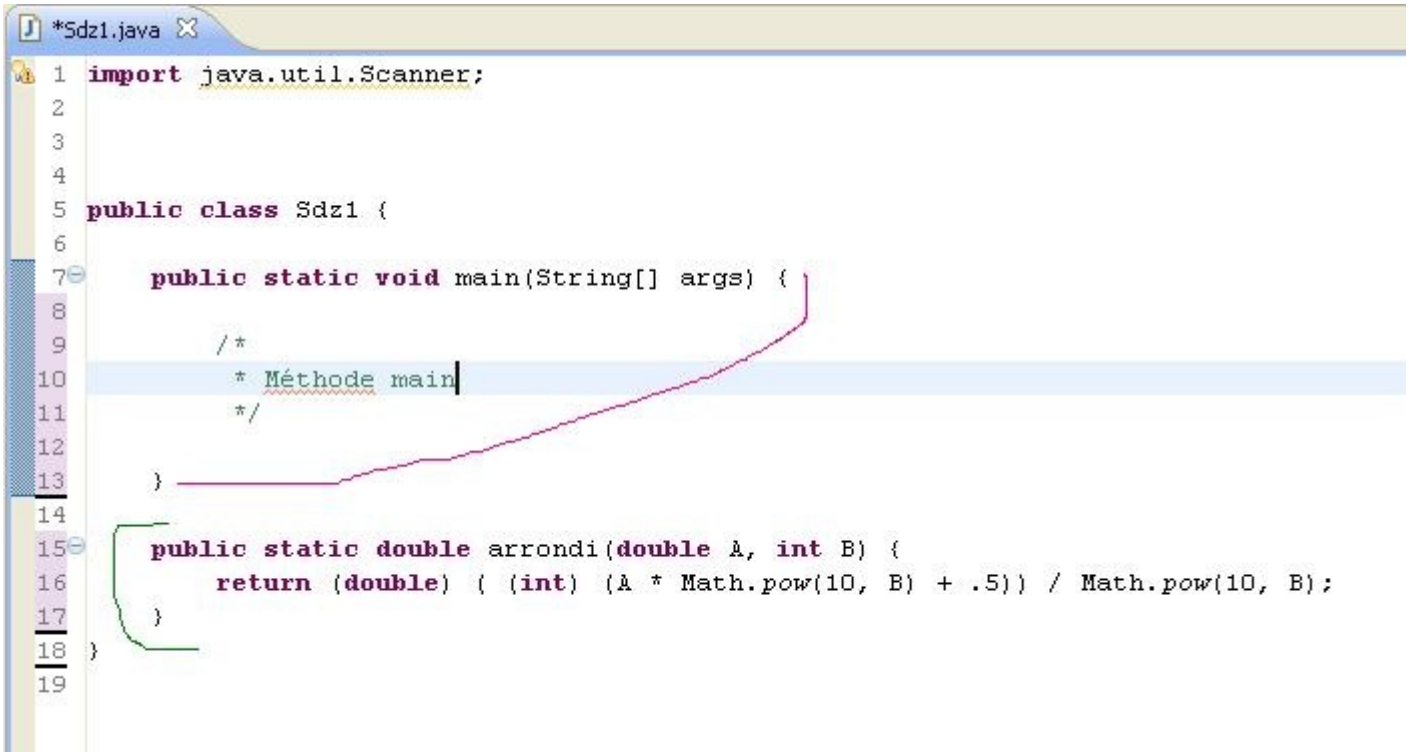
Dans ce chapitre, nous allons voir les différents types de renvois ainsi que les paramètres qu'une méthode peut prendre.

Vous devez savoir deux choses concernant les méthodes :

- **elles ne sont pas limitées en nombre de paramètres**
- il y a **trois grands types de méthodes** :
  - les méthodes qui ne renvoient rien. Elles sont de type **`void`**. Ces types de méthodes n'ont pas d'instruction **`return`** !
  - les méthodes qui retournent des types primitifs (`double`, `int`...). Elles sont de type **`double`**, **`int`**, **`char`**... Celles-ci ont une instruction **`return`**.
  - des méthodes qui retournent des objets. Par exemple, une méthode qui retourne un objet de type **`String`**. Celles-ci aussi ont une instruction **`return`**.

Pour le moment, nous ne faisions que des programmes comportant une classe. Celle-ci ne comportant qu'une méthode : la méthode **`main`**.
Le moment est donc venu de créer vos propres méthodes. Que vous ayez utilisé la méthode **`arrondi`** dans votre TP ou non, vous avez du voir que celle-ci se place à l'extérieur de la méthode **`main`**, mais dans votre classe !

Pour rappel, voici le screen qu'il y avait dans le TP 1 :



Si vous placez une de vos méthodes à l'intérieur de la méthode main ou à l'extérieur de votre classe : **LE PROGRAMME NE COMPILERA PAS ! !**

Bon, d'accord ! C'est enregistré. Mais concrètement... À quoi vont nous servir ces **méthodes** ?

Dites-vous bien qu'un développeur est de nature assez feignante... Sans déconner. Il s'est vite aperçu qu'il y avait du code redondant dans ses programmes... Des morceaux de code qui faisaient toujours la même chose et qui n'avaient, comme seule différence, la (ou les) variable(s) qu'ils traitaient.

Vu que nous venons de voir les tableaux, nous allons donc faire des méthodes concernant ces derniers !
Vous devez certainement vous rappeler de la façon de parcourir un tableau...
Et si nous faisions une méthode qui permette d'afficher le contenu d'un tableau sans que nous soyons obligés de retaper la portion de code contenant la boucle ? Je me doute que vous n'en voyez pas l'intérêt maintenant car, mis à part les plus courageux d'entre vous, vous n'avez fait qu'un ou deux tableaux de votre main lors de la partie précédente. Et si je vous demande de déclarer 22 tableaux, si je vous dis : **"Allez, bande de ZérOs ! Parcourez-moi tout ça !"**

Vous n'allez tout de même pas faire 22 boucles **`for`** ! De toute façon, je vous l'interdis ! Nous allons faire une méthode. Celle-ci va :

- prendre notre tableau en paramètre (ben oui... c'est lui que nous voulons parcourir),
- parcourir notre tableau à notre place,
- faire tous les `System.out.println()`,
- ne rien renvoyer.

Avec ce que nous avons défini, nous savons donc que notre méthode sera de type **`void`**, qu'elle prendra un tableau en paramètre (pour le moment, on travaillera avec des tableaux de **`String`**) et le contenu vous vous doutez de ce que ça va être... Une boucle. Cette fameuse boucle que nous ne serons plus obligés de répéter autant de fois que de tableaux de **`String`** !

Ce qui va nous donner :

Code : Java

```
static void parcourirTableau(String[] tab)
```

```
    {
      for(String str : tab)
        System.out.println(str);
    }
```

Et voici un exemple de code entier :

Code : Java

```
public class Sdz1
{
   public static void main(String[] args)
   {
      String[] tab = {"toto", "tata", "titi", "tete"};
      parcourirTableau(tab);

   }

   static void parcourirTableau(String[] tabBis)
    {
      for(String str : tabBis)
        System.out.println(str);
    }
}
```

Je sais que ça vous trouble encore, mais sachez que les méthodes ajoutées dans la classe **main** doivent être déclarées **static** ! Fin du mystère lors de la programmation orientée objet !

Bon. Vous voyez que la méthode parcourt le tableau passé en paramètre. Si vous créez plusieurs tableaux et appelez la méthode avec ces derniers, vous vous apercevrez que la méthode affiche le contenu de chaque tableau !

Voici une méthode au même effet que la méthode **parcourirTableau**, à la différence que celle-ci retourne une valeur. Ici, une chaîne de caractères !

Code : Java

```
static String toString(String[] tab)
{
  System.out.println("Méthode toString() ! ! !\n--------------------
-------");
   String retour = "";

   for(String str : tab)
       retour += str + "\n";

   return retour;
}
```

Et voici un code complet :

Code : Java

```
public class Sdz1 {

        public static void main(String[] args)
           {
              String[] tab = {"toto", "tata", "titi", "tete"};
              parcourirTableau(tab);
```

```
                System.out.println(toString(tab));
            }

            static void parcourirTableau(String[] tab)
             {
               for(String str : tab)
                 System.out.println(str);
             }

            static String toString(String[] tab)
            {
                   System.out.println("Méthode toString() ! ! !\n----
------------------------");
                    String retour = "";

                    for(String str : tab)
                        retour += str + "\n";

                    return retour;
            }


    }
```

Vous voyez que la deuxième méthode retourne une chaîne de caractères que nous devons afficher à l'aide de l'instruction **`System.out.println()`**. Nous affichons la valeur renvoyée par la méthode **`toString()`**. Alors que la méthode **`parcourirTableau`**, elle, écrit au fur et à mesure le contenu du tableau dans la console !
Notez que j'ai ajouté une ligne d'écriture dans la console dans la méthode **`toString()`** afin de vous montrer où était appelée cette dernière.

L'instruction **`System.out.println()`** est très pratique pour déboguer vos programmes ! Dans une méthodes, l'utilisation de cette instruction peut faire ressortir des erreurs de conception ou de développement !

Il nous reste un point important à aborder.
Imaginez un instant que vous ayez plusieurs types d'éléments à parcourir ! Des tableaux à une dimension, mais aussi d'autres à deux dimensions, et même des objets comme des **`ArrayList`** (nous les verrons, ne vous inquiétez pas...). Sans aller si loin. Vous n'allez pas donner un nom différent à la méthode **`parcourirTableau`** à chaque type primitif !

Vous avez dû remarquer que la méthode que nous avons créée ne prend qu'un tableau de **`String`** en paramètre ! Pas de tableau d'**`int`** ni de **`long`**.

Si seulement nous pouvions utiliser la même méthode pour différents types de tableaux !

C'est là qu'entre en jeu ce qu'on appelle : la surcharge.
Retour en haut

La surcharge de méthode

Grâce à ceci, vous n'aurez plus à retenir 10 000 noms de méthodes qui font la même chose !

Le principe

La surcharge de méthode consiste à garder un nom de méthode (donc un type de traitement à faire, pour nous, lister un tableau) et de changer la liste ou le type de ses paramètres.

Dans le cas qui nous intéresse, nous voulons que notre méthode **`parcourirTableau`** puisse parcourir n'importe quel type de tableau.

Comment faire ?

Nous allons surcharger notre méthode afin qu'elle puisse travailler avec des **`int`** par exemple :

Code : Java

```
static void parcourirTableau(String[] tab)
  {
     for(String str : tab)
        System.out.println(str);
  }

static void parcourirTableau(int[] tab)
  {
     for(int str : tab)
        System.out.println(str);
  }
```

Avec ces méthodes, vous pourrez parcourir de la même manière :

- les tableaux d'entiers
- les tableaux de chaînes de caractères.

Mais vous pouvez aussi faire de même avec les tableaux à 2 dimensions.
Voici à quoi pourrait ressembler son code (je ne rappelle pas le code des deux méthodes ci-dessus) :

Code : Java

```
static void parcourirTableau(String[][] tab)
  {
     for(String tab2[] : tab)
     {
        for(String str : tab2)
          System.out.println(str);
     }
  }
```

La surcharge de méthode fonctionne aussi en ajoutant des paramètres à la méthode.
Cette méthode est donc valide :

Code : Java

```
static void parcourirTableau(String[][] tab, int i)
  {
     for(String tab2[] : tab)
```

```
        {
            for(String str : tab2)
              System.out.println(str);
        }
    }
```

Bon d'accord, nous ne nous servons pas de la variable **i**, mais c'était un exemple. Comme ça, vous avez vu les cas de surcharge de méthode !

Ce qu'il faut retenir de ce chapitre

- Les méthodes **se définissent dans une classe**.
- Les méthodes ne peuvent **pas être imbriquées**. Elles sont déclarées **les unes après les autres**.
- Les méthodes peuvent être surchargées, en **changeant le type de paramètres** que celle-ci attend, **le nombre de ses paramètres ou les deux** !
- Pour Java, le fait de surcharger une méthode lui indique qu'il s'agit de 2, 3 ou X méthodes différentes, car **les paramètres d'appel sont différents**. Par conséquent, **Java ne se trompe jamais d'appel de méthode, puisqu'il se base sur les paramètres passés**.

Je crois que vous êtes prêts pour un petit QCM des familles !

Retour en haut

J'espère que vous aurez appris beaucoup de choses dans ce chapitre. En tout cas, je sais que vous en aurez besoin, et pas plus tard que pour la partie suivante !

Prêts pour la **programmation orientée objet** ? Here we go !

Retour en haut

J'ose espérer que vous avez apprécié ce tuto sur les bases du langage Java ! En tout cas, je me suis bien amusé en le faisant.

Maintenant, nous allons rentrer dans les méandres de la programmation orientée objet !
Alors ?... Toujours prêts ?

Retour en haut

Partie 2 : Java Orienté Objet

Dans cette partie, nous allons aborder la programmation orientée objet. Concept de programmation extrêmement puissant et pratique.
En effet, vous verrez qu'avec ce type de programmation, vous pourrez créer, utiliser, recréer divers objets et les utiliser dans un but que vous seuls aurez décidé.

J'aborderai ce que j'ai pu voir durant ma formation, et uniquement cela... Mais je vous rassure : il y a déjà du travail...

Cette partie sera extrêmement riche en concepts, vocabulaire et méthodologie. Entre autres, vous saurez programmer en orienté objet, vous pourrez enregistrer vos objets dans des fichiers...

J'ajouterai aussi quelques notions de modélisation. Ceci dans le but de vous familiariser avec la façon de schématiser des objets et leurs interactions entre eux. Nous y reviendrons, mais il s'agira de diagrammes de classes utilisés avec le langage UML (Unified Modeling Language).

Une longue introduction ne servirait à rien... passons donc tout de suite à la première partie.
Retour en haut

Les premiers pas en "Orienté Objet"

Dans la première partie de ce tuto sur la programmation en Java, nous avons travaillé avec une seule classe.
Vous allez voir et apprendre qu'en programmation orientée objet, nous travaillerons avec plusieurs classes : en fait, autant de classes que d'objets.

Rappelez-vous de la première partie : **vous avez déjà utilisé des objets...**
Oui ! Lorsque vous faisiez ceci :

Code : Java

```
String str = new String("tiens... un objet String");
```

Ici `str` est un objet *String*. Vous avez créé un objet de la classe `String` : on dit que vous avez créé une instance de la classe `String()`. Nous savons cela grâce à l'opérateur `new`, qui s'utilise pour instancier une classe.
L'objet `String`, instancié ci-dessus, a bien ses propres données : la chaîne de caractères ***"tiens... un objet String"***. Vous devez savoir aussi que les variables de type `String()` (mais nous préfèrerons parler d'objet à partir de maintenant) ont des méthodes associées, comme `subString()`.

Je vous sens quelque peu perplexes... mais néanmoins rassurés... Attaquons sans plus attendre !

Les classes

Une classe peut être comparée à un moule, qui, lorsque nous le remplissons, nous donne un objet ayant la forme du moule, et toutes ses caractéristiques. Comme quand vous étiez enfants, quand vous vous amusiez avec de la pâte à modeler.

Si vous avez bien suivi la première partie de ce tuto, vous devriez savoir que notre classe contenant la méthode `main` ressemble à ceci :

Code : Java

```
class ClasseMain{

public static void main(String[] args){

//Vos données, variables, différents traitements....

}// fin de la méthodes main

}//fin de votre classe
```

Créez cette classe et cette méthode `main` (vous savez faire, maintenant).

Puisque nous allons travailler en POO, nous allons devoir créer une seconde classe dans ce fameux projet ! Nous allons donc tout de suite créer une classe **`Ville`**.

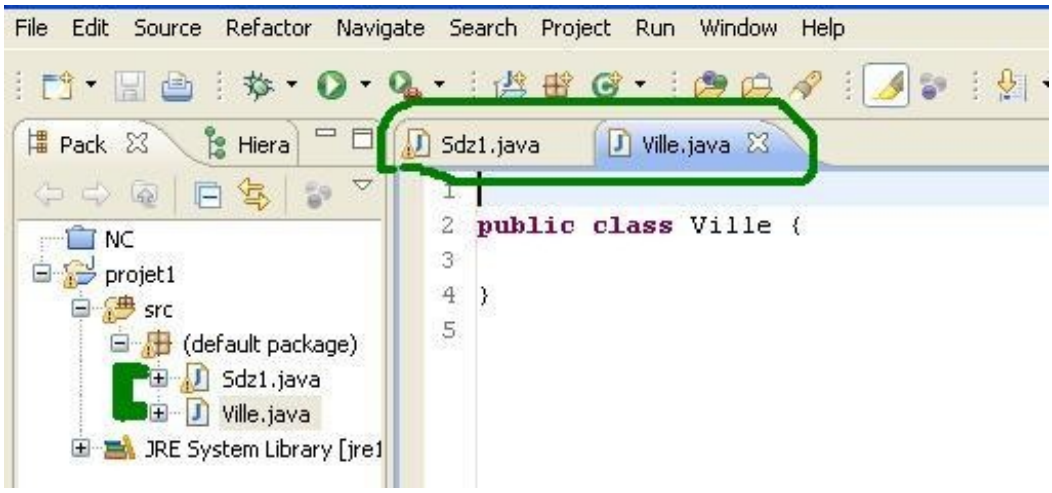
Dans un but de simplicité, j'ai pensé que créer des objets **`Ville`** vous permettrait d'assimiler plus facilement le concept objet. Nous allons donc créer des objets **`Ville`** avant la fin de ce chapitre...

Allez dans File / New / Class ou utilisez le raccourci dans la barre d'outils, comme ceci :

Nommez votre classe : `Ville` (convention de nommage ! !). Mais cette fois vous ne devez pas créer la méthode `main`.

Il ne peut y avoir qu'une seule méthode `main` par projet ! Souvenez-vous que celle-ci est **le point de départ de votre programme** ! Et un programme ne commence qu'à **un seul endroit** !
Pour être tout à fait précis, il peut exister plusieurs méthode `main` dans votre projet (oui, même une par classe...) **MAIS une seule sera considérée comme le point de départ de votre programme !**

Au final, vous devriez avoir ceci :



Voilà ! Maintenant vous avez votre classe, avec la méthode `main`, et votre classe `Ville` encore vide mais qui va bientôt pouvoir créer des objets `Ville`.

Un dernier mot cependant. Ici, notre classe ville est précédée du mot clé "**public**". Vous devez savoir que lorsque nous créons une classe comme nous l'avons fait, Eclipse nous facilite la tache en mettant ce mot clé. Nous verrons plus tard qu'il existe d'autres mots clé pour définir la **portée d'une classe** (nous y viendrons). Si nous enlevons ce mot clé, l'interpréteur Java considèrera, tacitement, votre classe comme **public**.

Retenez que, par défaut et sans instructions contraires, toutes les classes sont **public** !

C'est-à-dire ?

Nous verrons cela à la fin du chapitre... Dès que nous aurons créé des objets !
Retenez que :

Code : Java

```
public class UneClasse{

}//fin de votre classe
```

et

Code : Java

```
class UneClasse{

}//fin de votre classe
```

Sont deux codes équivalents !

Ce mot-clé, `public`, vous l'avez déjà rencontré lors du chapitre sur les méthodes.
Mais lorsque `public` précède une méthode, il s'agit d'un droit d'accès à une méthode membre d'une classe... c'est-à-dire qu'avec le mot-clé `public` devant une méthode, celle-ci sera accessible par toutes les classes utilisant cette dernière. Bien entendu, nous aurons besoin de ce genre de méthode, mais nous aurons aussi besoin d'autres méthodes dont nous ne nous servirons que dans la classe où celle-ci sera créée...

Dans ce cas de figure, nous utiliserons le mot clé `private`. Ce qui signifie que notre méthode ne pourra être appelée que dans la classe où elle a vu le jour !

Il en va de même pour les variables. Nous allons voir, dans la deuxième sous-partie, que nous pouvons protéger des variables grâce au mot-clé `private`. Le principe sera le même que pour les méthodes... Ces variables ne seront accessibles que dans la classe où elles seront nées...

Bon !... Toutes les conditions sont réunies pour pouvoir commencer activement la programmation orientée objet ! Et si nous allions créer notre première ville ?

Retour en haut

Les constructeurs

Derrière ce mot un peu barbare, se cache une notion toute bête. Vu que notre objectif dans ce chapitre est de construire un objet **`Ville`**, il va falloir définir les données qu'on va lui attribuer.

Nous dirons qu'un objet **`Ville`** a :

- un nom sous forme de chaîne de caractères,
- un nombre d'habitants sous la forme d'un entier,
- un pays apparenté sous la forme d'une chaîne de caractères.

Je suis bien d'accord ! Mais comment fait-on pour dire à notre programme que notre objet a tout ça ?

Tout simplement en mettant des variables (*dites d'instances*) dans notre classe.
Celle-ci va contenir une variable dont le rôle sera de stocker le nom, une autre stockera le nombre d'habitants et la dernière se chargera du pays !
Voilà à quoi ressemble notre classe **`Ville`** à présent :

Code : Java

```
public class Ville{

  String nomVille;
  String nomPays;
  int nbreHabitant;

}
```

Rappelez-vous que, par défaut, les variables d'instances présentes dans une classe sont **`public`**.

Pourquoi tu dis *variables d'instances* ?

Tout simplement parce que dans nos futures classes Java, qui définiront des objets, il y aura plusieurs types de variables dans celles-ci (nous approfondirons ceci dans ce chapitre).
Pour le moment, sachez qu'il y a trois grands types de variables dans une classe objet :

- **les variables d'instances** : ce sont elles qui définiront les caractéristiques de notre objet ;
- **les variables de classes** : celles-ci sont communes à toutes les instances de votre classe ;
- **les variables locales** : ce sont des variables que nous utiliserons pour travailler dans notre objet.

Dans l'immédiat, nous allons travailler avec des variables d'instances afin de créer des objets différents. Il nous reste plus qu'à créer notre premier objet mais pour ce faire, nous allons devoir créer ce qu'on appelle des **constructeurs**.

Un constructeur est une méthode (ou méthode d'instance, vu qu'elle interagit avec une instance de votre classe) qui va se charger de créer un objet et, le cas échéant, d'initialiser ses variables de classe ! Cette méthode a pour rôle de dire à la ***JVM*** de réserver de l'allocation mémoire pour notre futur objet et donc, par extension, d'en réserver pour toutes les variables d'instances et variables de classes de cette dernière !

Notre premier constructeur sera ce qu'on appelle communément **un constructeur par défaut**.
C'est-à-dire qu'il ne prendra aucun paramètre mais permettra tout de même d'instancier un objet et vu que nous sommes perfectionnistes, nous allons initialiser nos variables d'instances.
Voici votre premier constructeur :

Code : Java

```
public class Ville{
    /**
```

```
    * Stocke le nom de notre ville
    */
      String nomVille;
      /**
    * Stocke le nom du pays de notre ville
    */
      String nomPays;
      /**
    * Stocke le nombre d'habitants de notre ville
    */
      int nbreHabitant;


      /**
    * Constructeur par défaut
    */
      public Ville(){
          System.out.println("Création d'une ville !");
          nomVille = "Inconnu";
          nomPays = "Inconnu";
          nbreHabitant = 0;
      }

    }
```

Vous avez remarqué que le constructeur est en fait une méthode qui n'a aucun type de retour (`void`, `double`...) et qui porte le même nom que notre classe !

Ceci est une règle immuable : le (les) constructeur(s) d'une classe doit (doivent) porter le même nom que la classe !

Une classe peut avoir plusieurs constructeurs ?

Bien sûr !
Il s'agit de la même méthode, mais surchargée ! Dans notre premier constructeur nous n'avons passé aucun paramètre, mais nous allons bientôt en mettre .

Vous pouvez d'ores et déjà créer une instance de ville.
Mais tout d'abord, rappelez-vous qu'une instance d'objet se fait grâce au mot-clé `new`. Comme lorsque vous créez une variable de type `String`. Vous avez sûrement déjà dû faire ce genre de déclaration :

Code : Java

```
String mot = new String();
```

Maintenant, vu que nous allons créer des objets **`Ville`**, nous allons procéder comme pour les `String`.

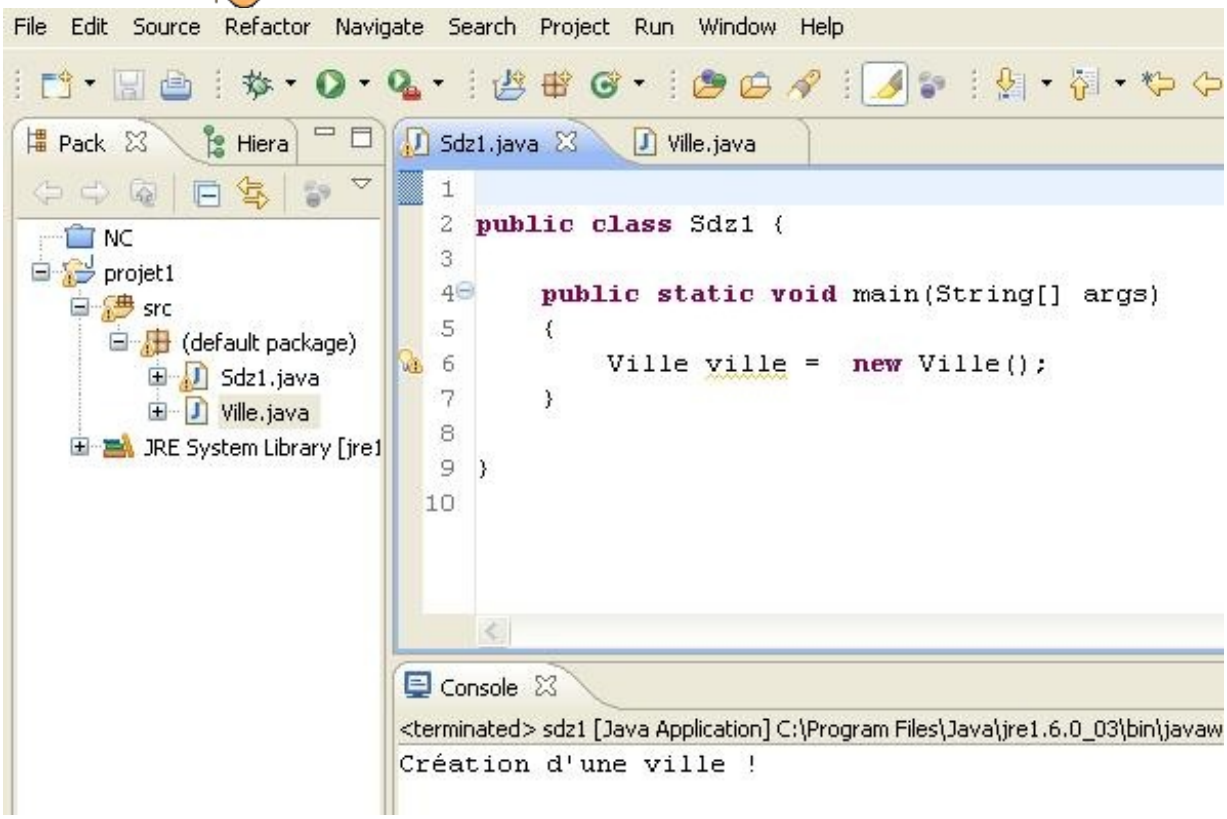
Vérifions que l'instanciation se fait bien. Allez dans votre classe contenant la méthode **`main`** et instancions un objet **`Ville`**. Je suppose que vous avez deviné que le type de notre objet ne sera pas **`double`**, **`int`** ou **`long`** mais bien **`Ville`** !

Code : Java

```
public class Sdz1{

 public static void main(String[] args){

     Ville ville = new Ville();
```

```
    }
}
```

Exécutez ce code, et voilà !



Félicitations ! Vous avez créé votre premier objet ! !

Je sais bien que, pour le moment, il ne sert à rien...
Mais vous devez passer par là afin de comprendre le principe de la **POO**.

Maintenant, nous devons mettre des données dans notre objet, ceci afin de pouvoir commencer à travailler...
Le but final serait d'avoir une déclaration d'objet se faisant comme ceci :

Code : Java

```
Ville ville1 = new Ville("Marseille", 123456789, "France");
```

Vous avez remarqué qu'ici les paramètres sont renseignés : eh bien il suffit de faire une méthode qui récupère ces paramètres, et initialise les variables de notre objet. Notre constructeur d'initialisation sera créé.

Voici le constructeur de notre objet **`ville`**, celui qui permet d'avoir des objets avec des paramètres différents. Comme je suis sympa, voici toute la classe telle qu'elle est maintenant :

Code : Java

```
public class Ville {
```

```
public class Ville {

  /**
* Stocke le nom de notre ville
*/
  String nomVille;
  /**
* Stocke le nom du pays de notre ville
*/
  String nomPays;
  /**
* Stocke le nombre d'habitants de notre ville
*/
  int nbreHabitant;


  /**
* Constructeur par défaut
*/
  public Ville(){
          System.out.println("Création d'une ville !");
          nomVille = "Inconnu";
          nomPays = "Inconnu";
          nbreHabitant = 0;
  }

  /**
* Constructeur d'initialisation
* @param pNom
* Nom de la Ville
* @param pNbre
* Nombre d'habitants
* @param pPays
* Nom du pays
*/
  public Ville(String pNom, int pNbre, String pPays)
  {
          System.out.println("Création d'une ville avec des
paramètres !");
          nomVille = pNom;
          nomPays = pPays;
          nbreHabitant = pNbre;
  }

}
```

Dans ce cas, l'exemple de déclaration et d'initialisation d'un objet **`ville`** que je vous ai montré un peu plus haut fonctionne sans aucun souci ! MAIS il vous faudra respecter scrupuleusement l'ordre des paramètres passés lors de l'initialisation de votre objet, sinon, c'est l'erreur de compilation à coup sûr !
Ainsi :

Code : Java

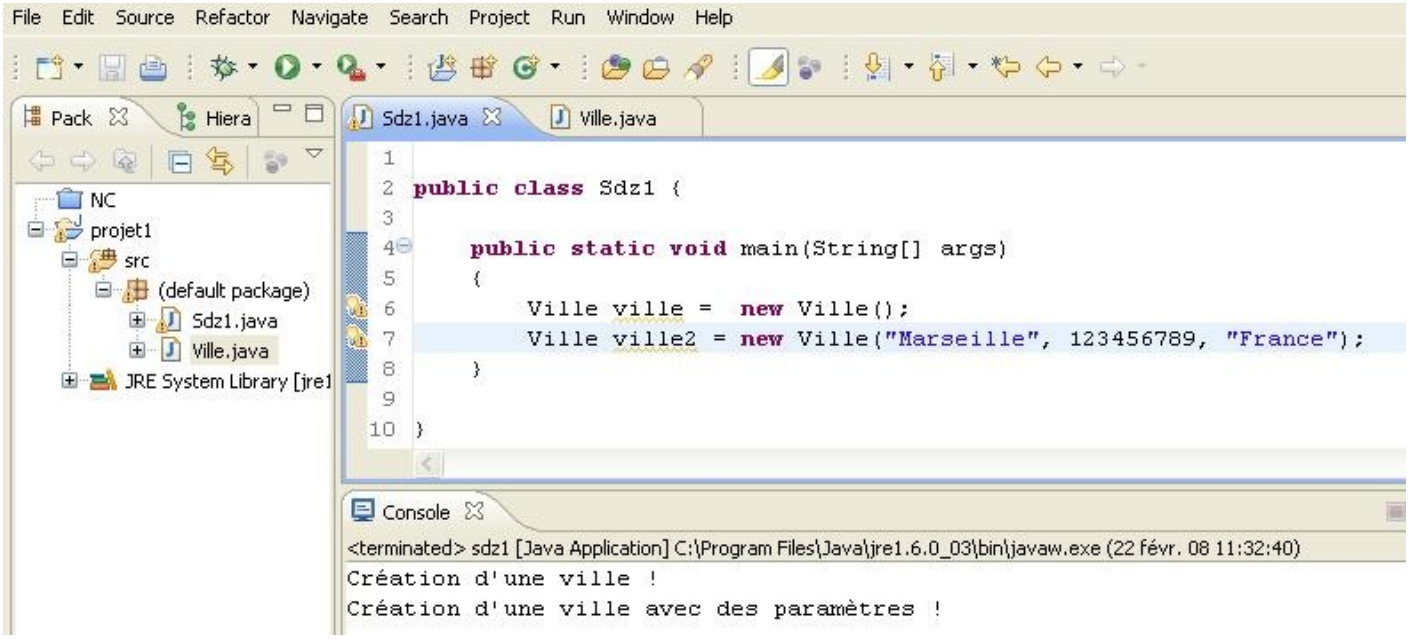
```
Ville ville1 = new Ville("marseille", 123456789, "France");//l'ordre
est respecté => aucun souci
Ville ville2 = new Ville(12456, "France", "Lille");//Erreur dans
l'ordre des paramètres => erreur de compilation au final
```

Testez ce code dans votre méthode `main` :

Code : Java

```
Ville ville =  new Ville();
Ville ville2 = new Ville("Marseille", 123456789, "France");
```

Voici le résultat :



Vous venez de **surcharger le constructeur** !

Par contre, notre objet à un gros défaut...

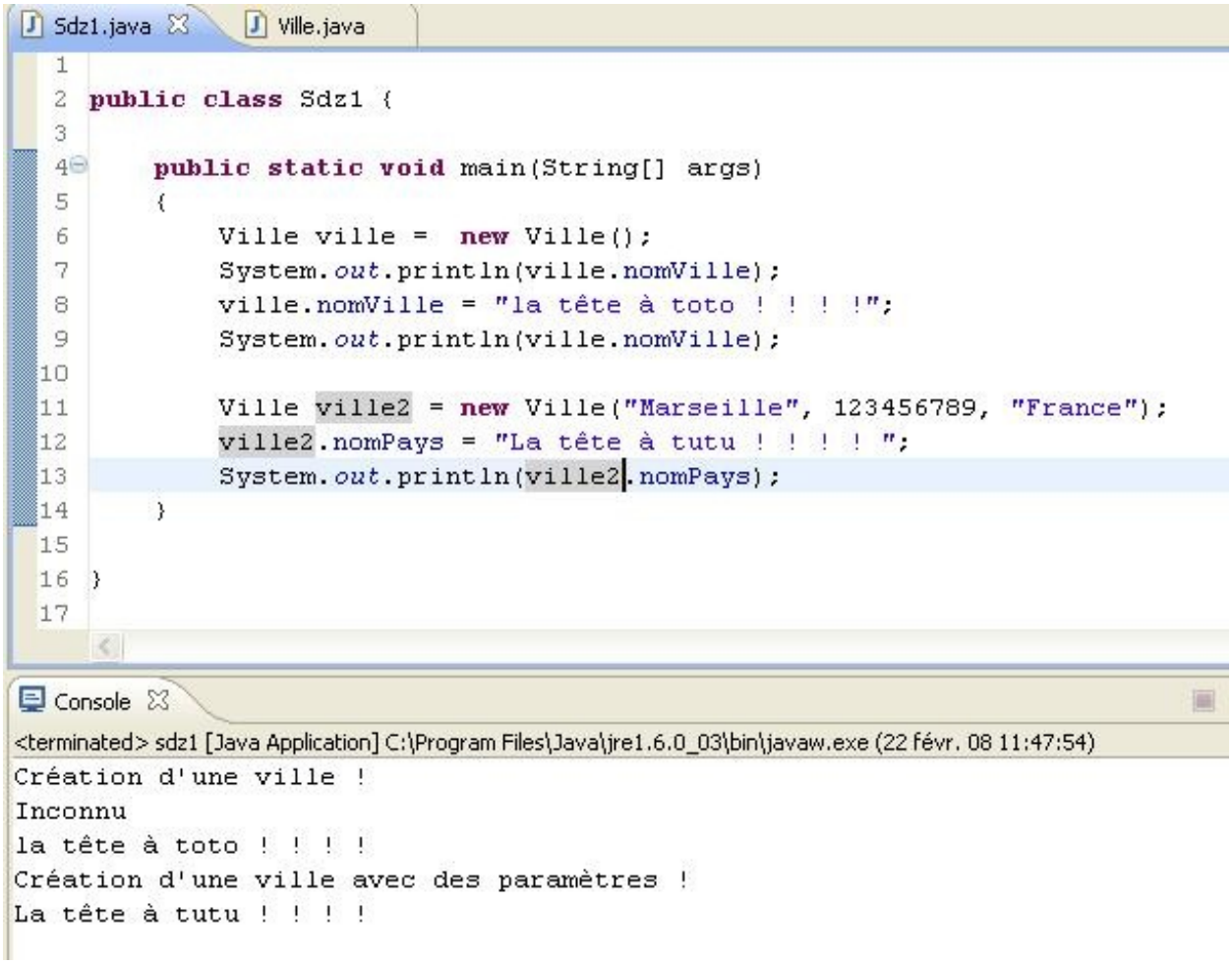
Les variables d'instances qui le caractérisent sont accessibles dans votre classe contenant votre `main` !

Ceci veut dire que vous pouvez modifier les attributs d'une classe directement. Testez ce code et vous verrez :

Code : Java

```
public class Sdz1 {

	public static void main(String[] args)
	{
		Ville ville =  new Ville();
		System.out.println(ville.nomVille);
		ville.nomVille = "la tête à toto ! ! ! !";
		System.out.println(ville.nomVille);

		Ville ville2 = new Ville("Marseille", 123456789,
"France");
		ville2.nomPays = "La tête à tutu ! ! ! ! ";
		System.out.println(ville2.nomPays);
	}

}
```

Et le résultat :



Vous voyez que nous pouvons accéder aux variables d'instances en utilisant le "**.**". Comme lorsque vous appelez la méthode **`subString()`** de l'objet **`String`**.
C'est très risqué et la plupart des programmeurs Java vous le diront.
Pour la plupart des cas, nous allons contrôler les modifications des variables de classe de manière à ce qu'un code ne fasse pas n'importe quoi avec nos objets !

C'est pour cela que nous protégeons nos variables d'instances en les déclarant **`private`**. Comme ceci :

Code : Java

```
public class Ville {

  /**
* Stocke le nom de notre ville
*/
  private String nomVille;
  /**
* Stocke le nom du pays de notre ville
*/
  private String nomPays;
  /**
* Stocke le nombre d'habitants de notre ville
*/
  private int nbreHabitant;


  /**
* Constructeur par défaut
*/
  public Ville(){
          System.out.println("Création d'une ville !");
          nomVille = "Inconnu";
          nomPays = "Inconnu";
```

```
                nbreHabitant = 0;
        }

        /**
     * Constructeur d'initialisation
     * @param pNom
     * Nom de la Ville
     * @param pNbre
     * Nombre d'habitants
     * @param pPays
     * Nom du pays
     */
        public Ville(String pNom, int pNbre, String pPays)
        {
                System.out.println("Création d'une ville avec des
    paramètres !");
                nomVille = pNom;
                nomPays = pPays;
                nbreHabitant = pNbre;
        }

    }
```

Et, si vous n'avez pas effacé les lignes de code où nous avons modifié les attributs de nos objets `Ville`, vous devez voir qu'Eclipse n'apprécie pas du tout que vous tentiez d'accéder à des variables de classe **privé** !

Ces attributs ne sont plus accessibles en dehors de la classe où ils sont déclarés !

À partir de maintenant, ce ne sera plus le programme instanciant une classe qui ira voir ou modifier les attributs de notre objet, mais notre objet qui renverra les informations (ou les modifiera) lorsque le programme lui demandera !

Bon... Vous avez fait le plus dur ! Si, si, je vous assure !

Maintenant, il va falloir se servir de ces objets... Eh oui ! Le but est tout de même d'utiliser nos objets dans des programmes. Pour pouvoir accéder aux données de nos objets, nous allons utiliser ce que l'on appelle des ***ACCESSEURS*** et pour modifier les données, on appelle ça des ***MUTATEURS***. Donc que dire, si ce n'est :
"À l'abordage, moussaillons ! "

Retour en haut

Votre objet sait parler : accesseurs et mutateurs

Un accesseur est une méthode qui va nous permettre d'accéder aux variables de nos objets en lecture et un mutateur, en écriture !

Reprenons là où nous nous étions arrêtés.

Vous avez une classe `Ville` qui crée vos objets.
Vous avez une classe avec une méthode `main`, prête à accueillir vos objets, et tout ce que vous voudrez en faire ! Mais voilà... pas moyen de faire quelque chose de ces satanés objets ! Eh bien... le calvaire est presque terminé ! Grâce aux accesseurs, vous pourrez afficher les variables de vos objets, et grâce aux mutateurs, les modifier.

Voilà à quoi ressemblent les accesseurs et les mutateurs :

Code : Java

```
public class Ville {

  /**
* Stocke le nom de notre ville
*/
  private String nomVille;
  /**
* Stocke le nom du pays de notre ville
*/
  private String nomPays;
  /**
* Stocke le nombre d'habitants de notre ville
*/
  private int nbreHabitant;


  /**
* Constructeur par défaut
*/
  public Ville(){
          System.out.println("Création d'une ville !");
          nomVille = "Inconnu";
          nomPays = "Inconnu";
          nbreHabitant = 0;
  }

  /**
* Constructeur d'initialisation
* @param pNom
* Nom de la Ville
* @param pNbre
* Nombre d'habitants
* @param pPays
* Nom du pays
*/
  public Ville(String pNom, int pNbre, String pPays)
  {
          System.out.println("Création d'une ville avec des paramètres !");
          nomVille = pNom;
          nomPays = pPays;
          nbreHabitant = pNbre;
  }



//*************************************************************************
  // ACCESSEURS

//*************************************************************************

  /**
```

```
        * Retourne le nom de la ville
        * @return le nom de la ville
        */
          public String getNom()
          {
                  return nomVille;
          }

          /**
        * Retourne le nom du pays
        * @return le nom du pays
        */
          public String getNomPays()
          {
                  return nomPays;
          }

          /**
        * Retourne le nombre d'habitants
        * @return nombre d'habitants
        */
         public int getNombreHabitant()
         {
                 return nbreHabitant;
         }


         //*********************************************************************
        *
         // MUTATEURS
         //*********************************************************************
        *

         /**
        * Définit le nom de la ville
        * @param pNom
        * nom de la ville
        */
         public void setNom(String pNom)
         {
                  nomVille = pNom;
         }

         /**
        * Définit le nom du pays
        * @param pPays
        * nom du pays
        */
         public void setNomPays(String pPays)
         {
                  nomPays = pPays;
         }

         /**
        * Définit le nombre d'habitants
        * @param nbre
        * nombre d'habitants
        */
        public void setNombreHabitant(int nbre)
        {
                 nbreHabitant = nbre;
        }

        }
```

Nos accesseurs sont bien des méthodes, et elles sont `public` pour que vous puissiez y accéder dans une autre classe que

celle-ci (la classe `main`, par exemple..).

Les accesseurs sont du même type que la variable qu'ils doivent retourner. Ça semble logique, non ?
Les mutateurs sont, par contre, de type `void`. Ce mot clé signifie ***"rien"*** ; en effet, ces méthodes ne retournent aucune valeur, elles se contentent de les mettre à jour.

Je vous ai fait faire la différence entre accesseurs et mutateurs mais, généralement, lorsqu'on parle d'accesseurs, ce terme englobe aussi les mutateurs.
Autre chose : il s'agit ici d'une question de convention de nommage. Les accesseurs commencent par **`get`** et les mutateurs par **`set`**, comme vous pouvez le voir ici. On parle d'ailleurs de **Getters** et de **Setters**.

Essayez ce code dans votre méthode `main` :

Code : Java

```
Ville v = new Ville();
        Ville v1 = new Ville("marseille", 123456, "france");
        Ville v2 = new Ville("rio", 321654, "brésil");


        System.out.println("\n v = "+v.getNom()+" ville de
"+v.getNombreHabitant()+ " habitants se situant en
"+v.getNomPays());
        System.out.println(" v1 = "+v1.getNom()+" ville de
"+v1.getNombreHabitant()+ " habitants se situant en
"+v1.getNomPays());
        System.out.println(" v2 = "+v2.getNom()+" ville de
"+v2.getNombreHabitant()+ " habitants se situant en
"+v2.getNomPays()+"\n\n");

        /*Nous allons interchanger les Villes v1 et v2
tout ça par l'intermédiaire d'un autre objet Ville
*/

       Ville temp = new Ville();
       temp = v1;
       v1 = v2;
       v2 = temp;

       System.out.println(" v1 = "+v1.getNom()+" ville de
"+v1.getNombreHabitant()+ " habitants se situant en
"+v1.getNomPays());
       System.out.println(" v2 = "+v2.getNom()+" ville de
"+v2.getNombreHabitant()+ " habitants se situant en
"+v2.getNomPays()+"\n\n");

       /*nous allons maintenant interchanger leurs noms
cette fois par le biais de leur accesseurs
*/

      v1.setNom("Hong Kong");
      v2.setNom("Djibouti");

      System.out.println(" v1 = "+v1.getNom()+" ville de
"+v1.getNombreHabitant()+ " habitants se situant en
"+v1.getNomPays());
      System.out.println(" v2 = "+v2.getNom()+" ville de
"+v2.getNombreHabitant()+ " habitants se situant en
"+v2.getNomPays()+"\n\n");
```

À la compilation, vous devriez obtenir ceci :

```
Création d'une ville !
Création d'une ville avec des paramètres !
Création d'une ville avec des paramètres !

 v = Inconnu ville de  0 habitants se situant en Inconnu
 v1 = marseille ville de  123456 habitants se situant en france
 v2 = rio ville de  321654 habitants se situant en brésil


Création d'une ville !
 v1 = rio ville de  321654 habitants se situant en brésil
 v2 = marseille ville de  123456 habitants se situant en france


 v1 = Hong Kong ville de  321654 habitants se situant en brésil
 v2 = Djibouti ville de  123456 habitants se situant en france
```

Vous voyez bien que les constructeurs ont fonctionné, que les accesseurs tournent à merveille, et que vous pouvez commencer à travailler avec vos objets `Ville`.

Mais la gymnastique des doigts à effectuer pour afficher les caractéristiques de nos objets... pfiou !... *T'aurais pu faire plus simple* ! Si seulement notre objet pouvait faire ça tout seul...

Il peut le faire !
Qu'est-ce qu'on attend ? Ben rien... on y va !

Retour en haut

Travaillez avec votre objet : les méthodes

Bon, alors là... c'est une formalité ! Vous savez quasiment tout.

Mais attends... tu vas nous reparler des méthodes, alors que tu ne fais que ça depuis le début de ce chapitre ?

Eh bien oui ! Ceci fait partie de la méthodologie que l'on m'a enseignée en formation, même si tout ce que nous avons fait depuis le début se rapporte aux méthodes, il faut différencier :

- les constructeurs => méthodes servant à créer des objets
- les accesseurs => méthodes servant à accéder aux données de nos objets
- et les méthodes de classe => méthodes servant à la gestion de nos objets.

Les méthodes de classe vont vous permettre de gérer, éditer, afficher... faire tout ce que vous voudrez avec vos objets.
Il ne vous reste plus qu'à trouver des méthodes qui fassent quelque chose pour vos objets...

Avec nos objets ville, nous sommes un peu limités en choix de méthodes... Mais nous pouvons tout de même en faire une ou deux pour l'exemple :

- faire un système de catégories de villes par rapport à leur nombre d'habitants (<1000 => A, <10 000 => B...). Ceci déterminé à la construction ou à la redéfinition du nombre d'habitants, donc ajouter une variable d'instance de type **char** à notre classe, appelons-la categorie. Penser à ajouter le traitement aux bons endroits.
- faire une méthode de description de notre objet Ville
- une méthode pour comparer deux objets par rapport à leur nombre d'habitants.

Nous voulons que la classe **Ville** gère la façon de déterminer la catégorie elle-même et non que cette action puisse être opérée de l'extérieur. La méthode qui fera ceci sera donc déclarée **private**.

Par contre, un problème va se poser ! Vous savez déjà qu'en Java on appelle des méthodes grâce à notre variable qui nous sert de référence, celle-ci suivie de l'opérateur "•", puis du nom de la dite méthode.
Par exemple :

Code : Java

```
String str = new String("opizrgpinbzegip");
str = str.subString(0,4);
```

Comment fait-on référence aux données de notre objet dans notre classe ?

En fait, c'est tout simple ! Encore un mot clé à retenir... Cette fois, il s'agit du mot-clé this.
Voici tout d'abord le code de notre *class* Ville en entier, c'est-à-dire avec nos méthodes associées :

Code : Java

```
public class Ville {

  /**
* Stocke le nom de notre ville
*/
  private String nomVille;
  /**
* Stocke le nom du pays de notre ville
*/
```

```
  private String nomPays;
  /**
* Stocke le nombre d'habitants de notre ville
*/
  private int nbreHabitant;
  /**
* Stocke le type de notre ville
*/
  private char categorie;

  /**
* Constructeur par défaut
*/
  public Ville(){
          nomVille = "Inconnu";
          nomPays = "Inconnu";
          nbreHabitant = 0;
          this.setCategorie();
  }

  /**
* Constructeur d'initialisation
* @param pNom
* Nom de la Ville
* @param pNbre
* Nombre d'habitants
* @param pPays
* Nom du pays
*/
  public Ville(String pNom, int pNbre, String pPays)
  {
          nomVille = pNom;
          nomPays = pPays;
          nbreHabitant = pNbre;
          this.setCategorie();
  }



//**************************************************************************
  // ACCESSEURS

//**************************************************************************

  /**
* Retourne le nom de la ville
* @return le nom de la ville
*/
  public String getNom()
  {
          return nomVille;
  }

  /**
* Retourne le nom du pays
* @return le nom du pays
*/
  public String getNomPays()
  {
          return nomPays;
  }

  /**
* Retourne le nombre d'habitants
* @return nombre d'habitants
*/
 public int getNombreHabitant()
 {
         return nbreHabitant;
 }
```

```
 /**
* Retourne la catégorie de la ville
* @return catégorie de la ville
*/
 public char getCategorie()
 {
         return categorie;
 }

 //*************************************************************************
*
 // MUTATEURS
 //*************************************************************************
*

 /**
* Définit le nom de la ville
* @param pNom
* nom de la ville
*/
 public void setNom(String pNom)
 {
          nomVille = pNom;
 }

 /**
* Définit le nom du pays
* @param pPays
* nom du pays
*/
 public void setNomPays(String pPays)
 {
          nomPays = pPays;
 }

 /**
* Définit le nombre d'habitants
* @param nbre
* nombre d'habitants
*/
public void setNombreHabitant(int nbre)
{
         nbreHabitant = nbre;
         this.setCategorie();
}



//*************************************************************************
// METHODES DE CLASSE
//*************************************************************************



  /**
* Définit la catégorie de la ville
*/
  private void setCategorie() {

      int bornesSuperieures[] = {0, 1000, 10000, 100000, 500000, 1000000, 50000
10000000};
        char categories[] = {'?', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'};

        int i = 0;
        while (i < bornesSuperieures.length && this.nbreHabitant >= bornesSuper
                i++;

        this.categorie = categories[i];
```

```
    }

    /**
   * Retourne la description de la ville
   * @return description ville
   */
    public String decrisToi(){
        return "\t"+this.nomVille+" est une ville de "+this.nomPays+", elle compo
  "+this.nbreHabitant+
                " => elle est donc de catégorie : "+this.categorie;
    }

    /**
   * Retourne une chaîne de caractères selon le résultat de la comparaison
   * @param v1
   * objet Ville
   * @return comparaison de deux ville
   */
    public String comparer(Ville v1){
        String str = new String();

        if (v1.getNombreHabitant() > this.nbreHabitant)
            str = v1.getNom()+" est une ville plus peuplée que "+this.nomVille;

        else
            str = this.nomVille+" est une ville plus peuplée que "+v1.getNom();

        return str;

    }

}
```

Pour simplifier, **this** fait référence à l'objet courant !

Pour expliciter le fonctionnement du mot clé this, prenons l'exemple de la méthode **comparer(Ville V1)**.
La méthode va s'appeler comme suit :

Code : Java

```
Ville V = new Ville("lyon", 654, "france");
Ville V2 = new Ville("lille", 123, "france");

V.comparer(V2);
```

Dans cette méthode, nous voulons comparer les nombres d'habitants des deux objets ville. Pour accéder à la variable nbreHabitant de l'objet V2, il suffit d'utiliser la syntaxe **V2.getNombreHabitant()** ; nous ferons donc référence à la propriété **nbreHabitant** de l'objet **V2**.
Mais l'objet V, lui, est l'objet appelant de cette méthode. Pour se servir de ses variables, on utilise alors **this.nbreHabitant**, ce qui a pour effet de faire appel à la variable **nbreHabitant** de l'objet appelant la méthode **comparer(Ville V)**.

Explicitons un peu ces trois méthodes.

La méthode

setCategorie()
Elle ne prend aucun paramètre, et ne renvoie rien : elle se contente de mettre à jour la variable de classe **categorie**. Elle regarde, par rapport au nombre d'habitants de l'objet appelant, utilisé grâce au mot clé this, dans quelle tranche se trouve la ville en question. Selon le nombre d'habitants, le caractère renvoyé changera. Nous l'appelons lorsque nous construisons un objet Ville avec ou sans paramètre, mais aussi lorsque nous redéfinissons le nombre d'habitants : de cette manière, la catégorie est mise à

jour automatiquement, sans faire appel à la méthode.
Vous constaterez aussi que nous n'avons pas créé de mutateur pour la variable d'instance **categorie** : nous avons en effet décidé que c'est à l'objet de gérer cette variable !

La méthode

decrisToi()
Celle-ci nous renvoie un objet de type *String*. Elle fait référence aux variables qui composent l'objet appelant la méthode, toujours grâce à this, et nous renvoie donc une chaîne de caractères qui nous décrit l'objet, en énumérant ses composants.

La méthode

comparer(Ville V1)
Elle prend une ville en paramètre, pour pouvoir comparer les variables **nbreHabitant** de l'objet appelant la méthode, et de celui passé en paramètre pour nous dire laquelle est la plus peuplée !

Bien entendu, vous pouvez créer vos propres méthodes, avec leurs paramètres, leurs types, etc. Je ne vous oblige en rien à faire exactement les mêmes que moi...

Les choses doivent vous sembler plus claires...
Si nous faisions un petit test... Essayez le code suivant dans votre méthode main :

Code : Java

```
Ville v = new Ville();
Ville v1 = new Ville("marseille", 1236, "france");
Ville v2 = new Ville("rio", 321654, "brésil");

System.out.println("\n\n"+v1.decrisToi());
System.out.println(v.decrisToi());
System.out.println(v2.decrisToi()+"\n\n");
System.out.println(v1.comparer(v2));
```

Ce qui devrait vous donner :

```
marseille est une ville de france, elle comporte : 1236 => elle est donc de catégorie : B
Inconnu est une ville de Inconnu, elle comporte : 0 => elle est donc de catégorie : ?
rio est une ville de brésil, elle comporte : 321654 => elle est donc de catégorie : D


rio est une ville plus peuplée que marseille
```

Je viens d'avoir une idée ! Et si nous essayons de savoir combien de villes nous avons créé ?
Comment faire ?
Avec une variable de classe !
Retour en haut

Les variables de classes

Comme je vous le disais au début de ce chapitre, il y a plusieurs types de variables dans une classe.
Nous avons vu les variables d'instances qui forment la carte d'identité d'un objet et maintenant, voici les variables de classes.

Celles-ci peuvent être très pratiques. Dans notre exemple, nous allons pouvoir compter le nombre d'instances de notre classe **Ville**, mais vous pouvez en utiliser pour d'autres choses (un taux de TVA dans une classe qui calcule le prix TTC, par exemple).

La particularité de ce type de variable, c'est qu'elles seront communes à toutes les instances de la classe !

Créons sans plus attendre notre compteur d'instance. Il s'agira d'une variable de type **int** que nous appellerons **nbreInstance** ; celle-ci sera **public** et nous ferons aussi son homologue en **private** : appellons-la **nbreInstanceBis** (un accesseur sera nécessaire pour accéder à cette dernière).
Pour qu'une variable soit une variable de classe, elle doit être précédée du mot clé **static**. Ce qui nous donnerait dans notre classe **Ville** :

Code : Java

```
public class Ville {

  /**
* Variables publiques qui comptent les instances
*/
  public static int nbreInstance = 0;
  /**
* Variable privée qui comptera aussi les instances
*/
  private static int nbreInstanceBis = 0;

  /**
* Stocke le nom de notre ville
*/
  private String nomVille;
  /**
* Stocke le nom du pays de notre ville
*/
  private String nomPays;
  /**
* Stocke le nombre d'habitants de notre ville
*/
  private int nbreHabitant;
  /**
* Stocke le type de notre ville
*/
  private char categorie;

  /**
* Constructeur par défaut
*/
  public Ville(){
          //On incrémente nos variables à chaque appel aux constructeurs
          nbreInstance++;
          nbreInstanceBis++;

          nomVille = "Inconnu";
          nomPays = "Inconnu";
          nbreHabitant = 0;
          this.setCategorie();
  }

  /**
* Constructeur d'initialisation
* @param pNom
* Nom de la Ville
* @param pNbre
* Nombre d'habitants
* @param pPays
```

```
     * Nom du pays
     */
      public Ville(String pNom, int pNbre, String pPays)
      {
            //On incrémente nos variables à chaque appel aux constructeurs
              nbreInstance++;
              nbreInstanceBis++;

              nomVille = pNom;
              nomPays = pPays;
              nbreHabitant = pNbre;
              this.setCategorie();
      }



    //*********************************************************************
      // ACCESSEURS

    //*********************************************************************

      public static int getNombreInstanceBis()
      {
              return nbreInstanceBis;
      }

      /**
     * Retourne le nom de la ville
     * @return le nom de la ville
     */
      public String getNom()
      {
              return nomVille;
      }

      /**
     * Retourne le nom du pays
     * @return le nom du pays
     */
      public String getNomPays()
      {
              return nomPays;
      }

      /**
     * Retourne le nombre d'habitants
     * @return nombre d'habitants
     */
     public int getNombreHabitant()
     {
             return nbreHabitant;
     }

     /**
     * Retourne la catégorie de la ville
     * @return catégorie de la ville
     */
     public char getCategorie()
     {
             return categorie;
     }

     //*********************************************************************
    *
     // MUTATEURS
     //*********************************************************************
    *

     /**
    * Définit le nom de la ville
```

```
	* @param pNom
	* nom de la ville
	*/
	 public void setNom(String pNom)
	 {
	          nomVille = pNom;
	 }

	 /**
	* Définit le nom du pays
	* @param pPays
	* nom du pays
	*/
	 public void setNomPays(String pPays)
	 {
	          nomPays = pPays;
	 }

	 /**
	* Définit le nombre d'habitants
	* @param nbre
	* nombre d'habitants
	*/
	public void setNombreHabitant(int nbre)
	{
	          nbreHabitant = nbre;
	          this.setCategorie();
	}



	//*********************************************************************
	// METHODES DE CLASSE
	//*********************************************************************



	  /**
	* Définit la catégorie de la ville
	*/
	  private void setCategorie() {


	      int bornesSuperieures[] = {0, 1000, 10000, 100000, 500000, 1000000, 50000
	10000000};
	        char categories[] = {'?', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'};

	        int i = 0;
	        while (i < bornesSuperieures.length && this.nbreHabitant >= bornesSuper
	                i++;

	        this.categorie = categories[i];


	  }

	  /**
	* Retourne la description de la ville
	* @return description ville
	*/
	  public String decrisToi(){
	      return "\t"+this.nomVille+" est une ville de "+this.nomPays+", elle compo
	"+this.nbreHabitant+
	              " => elle est donc de catégorie : "+this.categorie;
	  }

	  /**
	* Retourne une chaîne de caractères selon le résultat de la comparaison
	* @param v1
	* objet Ville
```

```
     * @return comparaison de deux ville
     */
      public String comparer(Ville v1){
          String str = new String();

          if (v1.getNombreHabitant() > this.nbreHabitant)
              str = v1.getNom()+" est une ville plus peuplée que "+this.nomVille;

          else
              str = this.nomVille+" est une ville plus peuplée que "+v1.getNom();

          return str;

      }

    }
```

Vous avez dû remarquer que l'accesseur de notre variable de classe déclarée privée est aussi déclaré **static**, et ceci est une règle !

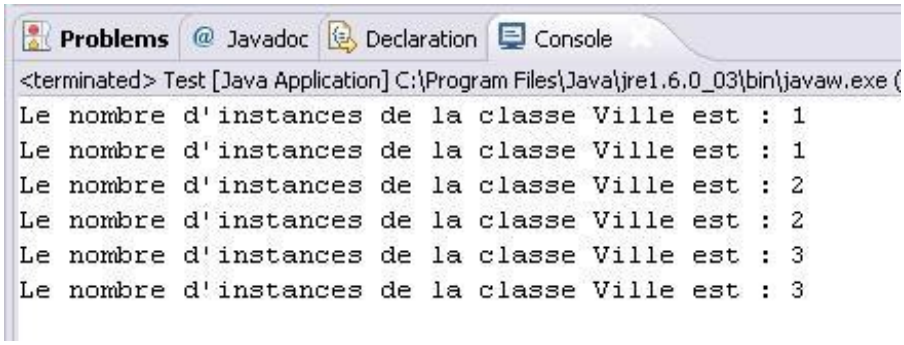
Toutes les méthodes de classes n'utilisant que des variables de classes doivent être déclarées **static** ! On les appelle des **méthodes de classes** car elles sont globales à toutes vos instances !
Par contre ceci n'est plus vrai si une méthode utilise des variables d'instances et des variables de classes...

Et maintenant, si vous testez le code suivant, vous allez voir l'utilité des variables des classes :

Code : Java

```
Ville v = new Ville();

System.out.println("Le nombre d'instances de la classe Ville est : "
+ Ville.nbreInstance);
System.out.println("Le nombre d'instances de la classe Ville est : "
+ Ville.getNombreInstanceBis());

Ville v1 = new Ville("marseille", 1236, "france");

System.out.println("Le nombre d'instances de la classe Ville est : "
+ Ville.nbreInstance);
System.out.println("Le nombre d'instances de la classe Ville est : "
+ Ville.getNombreInstanceBis());

Ville v2 = new Ville("rio", 321654, "brésil");

System.out.println("Le nombre d'instances de la classe Ville est : "
+ Ville.nbreInstance);
System.out.println("Le nombre d'instances de la classe Ville est : "
+ Ville.getNombreInstanceBis());
```

Résultat :



Vous voyez que le nombre augmente à chaque instanciation ! Et je suppose que le fait que j'ai utilisé le nom de classe **`Ville`** pour l'appel à nos variables de classes a dû vous surprendre.
Mais vous deviez savoir ceci...

Il vous faut savoir aussi que vous auriez également utilisé n'importe quelle instance pour faire ces appels. Vous auriez eu le même résultat - peu importe l'instance choisie - car cette donnée ne dépend pas de l'instance, mais de la classe elle-même.

Avant la fin de chapitre, j'ai une révélation à vous faire...

Retour en haut

Astuce Eclipse

Voici une astuce qui, je pense, va vous simplifier la vie et quelques minutes de codage...
Je vous vois déjà en train de vous imaginer coder une classe magnifique avec une centaine de variables d'instances ainsi qu'une dizaine de variables de classes, et d'un seul coup...

BOUM ! !

Vous tombez à la renverse lorsque vous devez faire tous les accesseurs de cette fabuleuse classe !

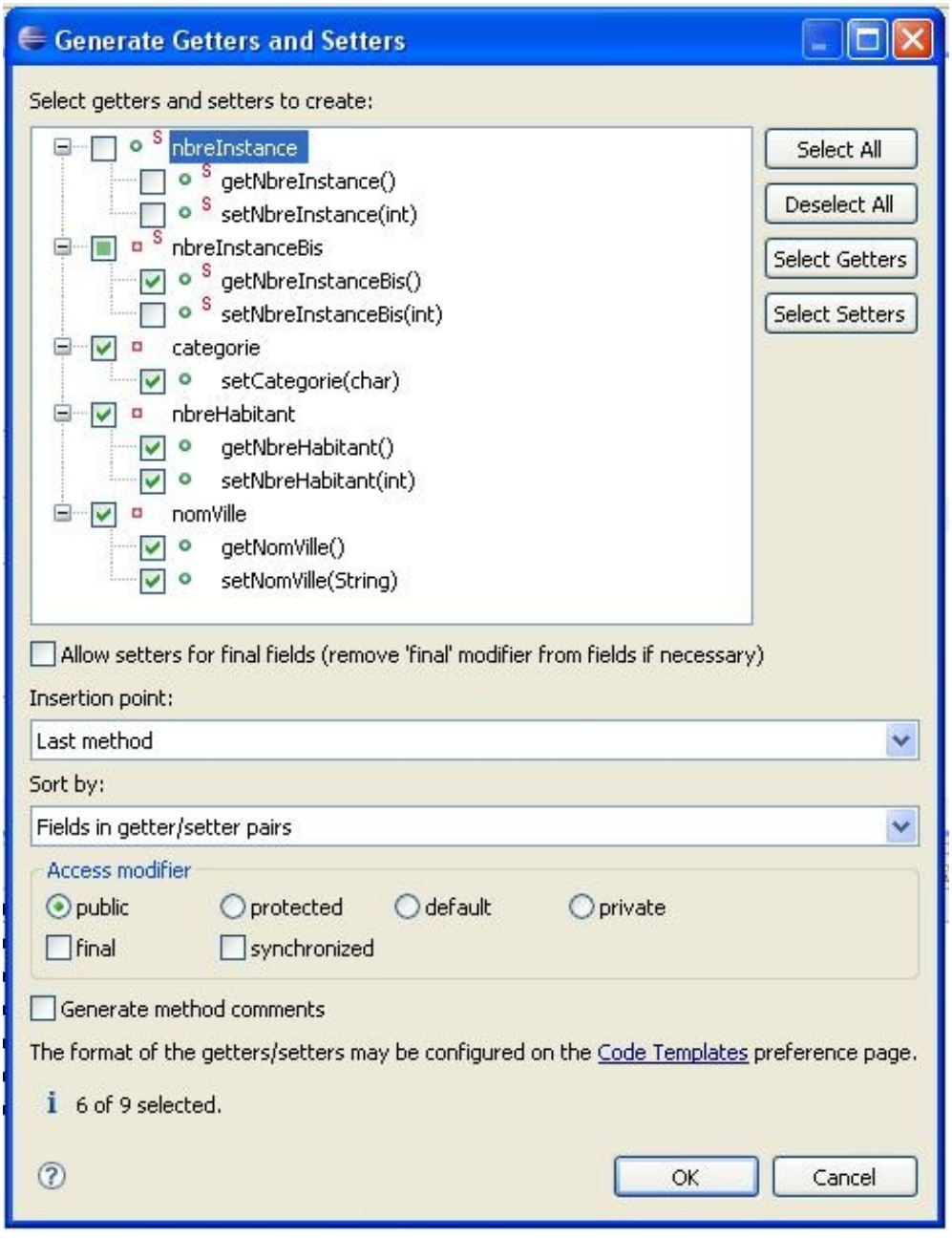
Voici quelque chose qui va vous plaire.

Eclipse vous fait les accesseurs automatiquement !

Bon : il faut tout de même créer toutes les variables au préalable, mais bon...

Nous allons prendre notre classe comme exemple.
Conservez toutes les déclarations de variables d'instances et de classes, et effacez tous les accesseurs (mutateurs compris).

Ensuite, rendez-vous dans le menu Source et choisissez l'option Generate Getters and Setters. Eclipse vous propose alors la liste des variables présentes dans la classe **`Ville`**.



Ici, j'ai coché ce qui m'intéressait et j'ai ensuite cliqué sur OK. Mes accesseurs sont tout prêts !

Alors, j'suis gentil, non ?

Comme vous pouvez le constater, Eclipse génère des accesseurs différents de ceux que nous avions faits aux préalable... Pensez-donc à mettre à jour votre méthode **`main`** si vous voulez faire des tests...

Je profite de cet intermède Eclipse pour vous expliquer pourquoi j'ai mis autant de commentaires autour de mes variables et de mes méthodes.

Par contre, ceci concerne les personnes qui ont téléchargé le **JDK** lors de la première partie !

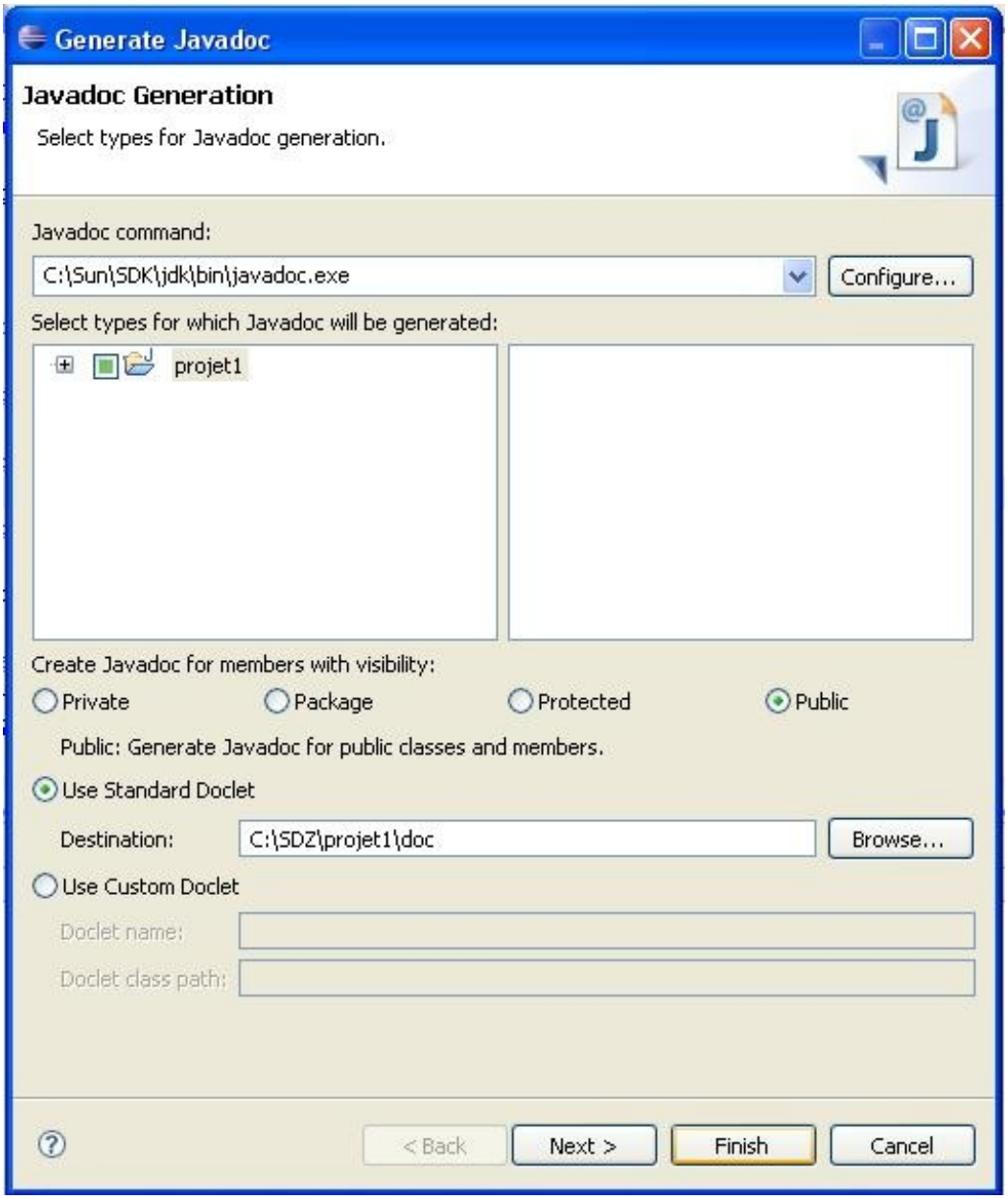
Comme je l'ai mentionné dans la première partie, il y a une troisième syntaxe de commentaires et vous venez de la voir tout au long de ce chapitre. Il s'agit de commentaires **javadoc** !

Cette syntaxe est quasi identique aux commentaires multilignes à une différence près : la présence d'une deuxième * au début du commentaire.
Vous avez dû constater que la couleur du commentaire était différente.

- Commentaire standard : commentaire de couleur verte.
- Commentaire javadoc : commentaire de couleur bleue.

À quoi ça sert ?

Tout simplement à documenter vos programmes et surtout à pouvoir générer une documentation automatiquement. Dans Eclipse, il vous suffit d'aller dans Project et de cliquer sur Generate Javadoc.
Vous devez avoir une popup comme ceci :



Si la commande Javadoc n'est pas renseignée, il vous faut aller chercher, dans le répertoire ***bin*** de votre JDK, le fichier javadoc.exe (comme sur la capture d'écran, chez moi c'est dans C:\SUN\SDK\jdk\bin\javadoc.exe).
Il ne vous reste plus qu'à cliquer sur Finish et à tout approuver.

Vous avez maintenant, dans votre dossier de code source, un dossier **doc**, où se trouve toute la javadoc de votre projet. Double cliquez sur ***index.html*** dans ce dossier, et voilà !

Pour les personnes ayant essayé la compilation en ligne de commande, il vous suffit d'aller dans votre dossier de source Java (là où se trouvent vos fichier .java) et de taper la commande :

Code : Bash

```
javadoc Ville.java
```

Retour en haut

Ce qu'il faut retenir

Après tout ceci, un petit topo ne fera pas de mal...

- Une classe permet de définir des objets. Ceux-ci ont des attributs (variable(s) d'instances) et des méthodes (méthodes de classes + accesseurs).
- Les objets permettent d'encapsuler du code et des données.
- Une classe, par défaut, est déclarée `public`.
- **Le ou les constructeurs d'une classe doivent porter le même nom que la classe, et pas de type de retour !**
- Il est recommandé de déclarer ses variables d'instance `private`.
- Dans une classe, on accède aux variables de celle-ci grâce au mot clé `this`.
- Une variable de classe est une variable devant être déclarée `static`.
- **Les méthodes n'utilisant que des variables de classes doivent elles aussi être déclarées `static`.**
- On instancie un nouvel objet grâce au mot clé `new`.
- L'ordre des paramètres passé dans le constructeur **doit être respecté**.
- Eclipse vous aide à développer vos applications.

Voilà ! Vous venez d'apercevoir les méandres de la programmation orientée objet...
Je suis conscient que ce chapitre fut très riche en nouveautés, vocabulaire, concepts et méthodologie... mais n'est-ce pas ce que je vous avait dit ?
Maintenant, ce que je vous propose, c'est un petit QCM digestif ! Ça passe toujours mieux après le digeo !
Retour en haut
Vous venez de voir l'un des chapitres les plus conséquents de cette partie...
Normalement, vous pouvez désormais créer et gérer des objets... Mais... *(parce qu'il y a un mais)*... vous allez voir, au fil de la partie suivante, que la programmation orientée objet offre tout un éventail de possibilités. L'une des plus importantes n'est autre que la notion d'***héritage***.

Sans perdre une seconde, je propose à ceux qui se sentent d'attaque de passer à la suite !
Retour en haut

L'héritage !

Je vous arrête tout de suite... Vous ne toucherez rien !
Pas de rapport d'argent entre nous... Non, la notion d'héritage en programmation est toute autre, quoique ressemblante à celle que vous connaissez. C'est l'un des fondements de la programmation orientée objet !

Imaginons que dans le programme fait précédemment, nous voulions créer un autre type d'objet :
des objets `Capitale`.
Ceux-ci ne seront ni plus ni moins que des objets `Ville` avec un paramètre en plus... Disons un ***président*** !

Et donc, au moment de créer votre classe `Capitale`, au lieu de tout redéfinir, nous allons dire que celle-ci est héritée de `Ville`. Trêve de bavardage ! À l'assaut !

La notion d'héritage

Comme je vous l'ai déjà dit lors de l'introduction, la notion d'héritage est l'un des fondements de la programmation orientée objet. Grâce à elle, nous pourrons créer des classes héritées (appelées aussi *classes dérivées*) de nos classes mères (appelées aussi *classes de base*).
Nous pourrons créer autant de classes dérivées, par rapport à notre classe de base, que nous le souhaitons. Nous pourrons aussi nous servir d'une classe dérivée comme d'une classe de base pour créer une autre classe dérivée...

Ce que vous devez savoir aussi, c'est que la notion d'héritage est l'un des piliers de la programmation événementielle (autre nom de *programmation graphique*). Ceci sera abordé dans la troisième partie de ce tuto.

Pour l'instant, restons dans la programmation procédurale !

Reprenons l'exemple dont je vous parlais dans l'introduction. Nous allons donc créer une nouvelle classe, nommée `Capitale` héritée de `Ville`.
Vous vous rendrez vite compte que les objets `Capitale` auront tous les attributs et méthodes associés des objets `Ville` !

Code : Java

```
class Capitale extends Ville {

}
```

C'est le mot-clé `extends` qui informe notre application que la classe `Capitale` est héritée de **`Ville`**. Pour vous le prouver, essayez ce morceau de code dans votre `main` :

Code : Java

```
Capitale cap = new Capitale();
 System.out.println("\n\n"+cap.decrisToi());
```

Vous devriez avoir ceci :

Ceci est bien la preuve que notre objet **`Capitale`** possède les avantages de notre objet **`Ville`**. Les objets hérités peuvent accéder à toutes les méthodes **`public`** de leur classe mère, ici la méthode **`decrisToi()`**.
Dans ce cas rudimentaire, notre objet **`Capitale`** ne possède que le constructeur par défaut et les méthodes associées.

En fait, lorsque vous déclarez une classe, si vous ne spécifiez pas de constructeur, la **JVM** créera au moment de l'interprétation le constructeur par défaut. C'est le cas ici. De plus, notre classe **`Capitale`** hérite de la classe **`Ville`**, ceci a pour effet que le constructeur de notre objet appelle, de façon tacite, le constructeur de la classe mère.
C'est pour cela que les variables d'instances ont pu être initialisées ! Par contre, dans notre classe **`Capitale`**, nous ne pouvons pas utiliser directement les attributs de la classe **`Ville`**.

Essayez ceci dans votre classe :

Code : Java

```
public class Capitale extends Ville{

   public Capitale(){
      this.nomVille = "toto";
   }
```

```
    }
```

Vous allez avoir une belle erreur de compilation !

Pourquoi ?

Tout simplement parce les variables de la classe **Ville** sont déclarés **private**.
Comme seules les méthodes et les variables déclarées **public** peuvent être utilisées dans une classe héritée, le compilateur rejette votre demande lorsque vous tentez d'accéder à des ressources privées d'une classe mère !

Comment y remédier tout en gardant la protection sur les variables de ma classe mère ?

C'est ici que je vais vous apprendre un nouveau mot clé : **protected**.
En remplaçant la déclaration des variables et des méthodes privées de la classe **Ville** en **protected**, cela aura pour effet de toujours protéger l'accès à celles-ci depuis du code utilisant un objet **Ville** ; mais cela permet aux classes qui héritent de cette dernière d'y avoir accès !
Donc, une fois toutes les variables et méthodes privées de la classe mère re-déclarées en **protected**, notre objet **Capitale** aura accès à celles-ci !

Ainsi, voici votre classe **Ville** revue et corrigée :

Code : Java

```
public class Ville {

  /**
* Variable publique compteur d'instances
*/
  public static int nbreInstance = 0;
  /**
* Variable privée compteur d'instances
*/
  protected static int nbreInstanceBis = 0;

  /**
* Stocke le nom de notre ville
*/
  protected String nomVille;
  /**
* Stocke le nom du pays de notre ville
*/
  protected String nomPays;
  /**
* Stocke le nombre d'habitants de notre ville
*/
  protected int nbreHabitant;
  /**
* Stocke le type de notre ville
*/
  protected char categorie;

  /**
* Constructeur par défaut
*/
  public Ville(){
          //On incrémente nos variables à chaque appel au constructeur
          nbreInstance++;
          nbreInstanceBis++;

          nomVille = "Inconnu";
          nomPays = "Inconnu";
          nbreHabitant = 0;
          this.setCategorie();
  }
```

```
  /**
* Constructeur d'initialisation
* @param pNom
* Nom de la Ville
* @param pNbre
* Nombre d'habitants
* @param pPays
* Nom du pays
*/
  public Ville(String pNom, int pNbre, String pPays)
  {

          nbreInstance++;
          nbreInstanceBis++;

          nomVille = pNom;
          nomPays = pPays;
          nbreHabitant = pNbre;
          this.setCategorie();
  }



//******************************************************************************
  // ACCESSEURS

//******************************************************************************

  public static int getNombreInstanceBis()
  {
          return nbreInstanceBis;
  }

  /**
* Retourne le nom de la ville
* @return le nom de la ville
*/
  public String getNom()
  {
          return nomVille;
  }

  /**
* Retourne le nom du pays
* @return le nom du pays
*/
  public String getNomPays()
  {
          return nomPays;
  }

  /**
* Retourne le nombre d'habitants
* @return nombre d'habitants
*/
 public int getNombreHabitant()
 {
         return nbreHabitant;
 }

 /**
* Retourne la catégorie de la ville
* @return catégorie de la ville
*/
 public char getCategorie()
 {
         return categorie;
 }
```

```
        //*******************************************************************************
        *
         // MUTATEURS
         //*******************************************************************************
        *

         /**
        * Définit le nom de la ville
        * @param pNom
        * nom de la ville
        */
         public void setNom(String pNom)
         {
                  nomVille = pNom;
         }

         /**
        * Définit le nom du pays
        * @param pPays
        * nom du pays
        */
         public void setNomPays(String pPays)
         {
                  nomPays = pPays;
         }

         /**
        * Définit le nombre d'habitants
        * @param nbre
        * nombre d'habitants
        */
        public void setNombreHabitant(int nbre)
        {
                 nbreHabitant = nbre;
                 this.setCategorie();
        }



        //*******************************************************************************
        // METHODES DE CLASSE
        //*******************************************************************************



          /**
        * Définit la catégorie de la ville
        */
          protected void setCategorie() {


              int bornesSuperieures[] = {0, 1000, 10000, 100000, 500000, 1000000, 50000
        10000000};
                char categories[] = {'?', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'};

                int i = 0;
                while (i < bornesSuperieures.length && this.nbreHabitant >= bornesSuper
                        i++;

                this.categorie = categories[i];


          }

          /**
        * Retourne la description de la ville
        * @return description ville
        */
          public String decrisToi(){
              return "\t"+this.nomVille+" est une ville de "+this.nomPays+", elle compo
```

```
"+this.nbreHabitant+
                " => elle est donc de catégorie : "+this.categorie;
  }

  /**
* Retourne une chaîne de caractères selon le résultat de la comparaison
* @param v1
* objet Ville
* @return comparaison de deux ville
*/
  public String comparer(Ville v1){
      String str = new String();

      if (v1.getNombreHabitant() > this.nbreHabitant)
          str = v1.getNom()+" est une ville plus peuplée que "+this.nomVille;

      else
          str = this.nomVille+" est une ville plus peuplée que "+v1.getNom();

      return str;

  }

}
```

Un point important avant de continuer.

Contrairement au C++, Java ne gère pas les héritages multiples : une classe dérivée (ou encore *classe fille*) ne peut hériter que d'une seule classe mère !

Vous n'aurez donc JAMAIS ce genre de classe :

Code : Java

```
class Toto extends Titi, Tutu{
}
```

À présent, continuons la construction de notre objet hérité !

Retour en haut

Construction d'un objet hérité

Il va de soi que cette opération va se concrétiser avec nos chers constructeurs.
Notre classe **Ville** ne changera plus d'un poil, mais nous allons par contre agrémenter notre classe `Capitale`.

Comme je vous l'avais dit, ce qui différenciera nos objets **Capitale** de nos objets **Ville** sera la présence d'un champ nouveau : le nom du président. Ce qui signifie que nous devons créer un constructeur par défaut et un constructeur d'initialisation pour notre objet `Capitale`.

Avant de foncer tête baissée, il faut que vous sachiez que nous pouvons faire appel aux variables de la classe mère dans nos constructeurs... Et ceci grâce au mot-clé **super**. Ce qui aura pour effet de récupérer les éléments de l'objet de base, et de les envoyer à notre objet hérité.

Démonstration :

Code : Java

```
class Capitale extends Ville {

private String president;

 /**
*Constructeur par défaut
*/
  public Capitale(){
    //Ce mot clé appelle le constructeur de la classe mère.
    super();
    president = "aucun";
  }
}
```

Si vous testez à nouveau le petit exemple que je vous avais montré un peu plus haut, vous vous apercevrez que le constructeur par défaut fonctionne toujours... Et pour cause, car ici, **super()** appelle le constructeur par défaut de l'objet `Ville` dans notre constructeur de `Capitale`, puis nous avons rajouté le président par défaut.

Mais la méthode **decrisToi()** ne prend pas en compte le nom du président...
Eh bien le mot-clé **super()** fonctionne aussi pour les méthodes de classe. Ce qui nous donne une méthode **decrisToi()** un peu différente... car nous allons rajouter le champ président dans notre description.

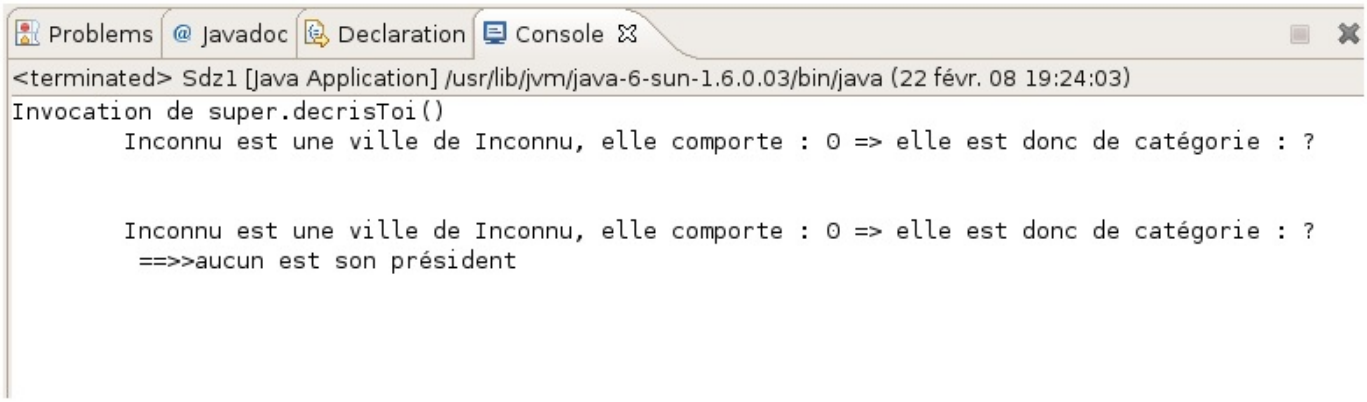
Voyez plutôt :

Code : Java

```
class Capitale extends Ville {

private String president;

 /**
*Constructeur par défaut
*/
  public Capitale(){
    //Ce mot clé appel le constructeur de la classe mère.
    super();
    president = "aucun";
  }

  /**
*Description d'une capitale
*/
   public String decrisToi(){
      String str = super.decrisToi() + "\n \t ==>>" + this.president
+ " est son président";
      //Pour bien vous montrer, j'ai ajouté la ligne ci-dessous,
mais vous n'êtes pas obligés...
```

```
            System.out.println("Invocation de super.decrisToi()");
            System.out.println(super.decrisToi());
            return str;
        }
    }
```

Si vous relancez les mêmes instructions présentes dans le `main`, depuis le début, vous aurez quelque chose comme ça :



Il y a du mieux, non ?
Bon, d'accord, nous n'avons toujours pas fait le constructeur d'initialisation de `Capitale`... Eh bien ? Qu'est-ce que nous attendons ?

Code complet de notre classe

Capitale

Code : Java

```
public class Capitale extends Ville {


        private String president;

         /**
*Constructeur par défaut
*/
          public Capitale(){
            //Ce mot clé appelle le constructeur de la classe mère.

            super();
            president = "aucun";
          }

          /**
* Constructeur d'initialisation de capitale
*/
          public Capitale(String nom, int hab, String pays, String
president){
          super(nom, hab, pays);
          this.president = president;
          }

          /**
*Description d'une capitale
*/
           public String decrisToi(){
```

```
                String str = super.decrisToi() + "\n \t ==>>" +
        this.president + " est son président";
                return str;
            }

            /**
        * @return le nom du président
        */
            public String getPresident() {
                    return president;
            }

            /**
        * Définit le nom du président
        * @param president
        */
            public void setPresident(String president) {
                    this.president = president;
            }

        }
```

Donc : dans le constructeur d'initialisation de notre `Capitale`, vous remarquez la présence de **`super(nom, hab, pays);`**. Difficile de ne pas le voir...
Ici, cette ligne de code joue le même rôle que pour le constructeur par défaut. Sauf qu'ici, le constructeur auquel `super` fait référence prend trois paramètres... donc `super` doit prendre ces paramètres.

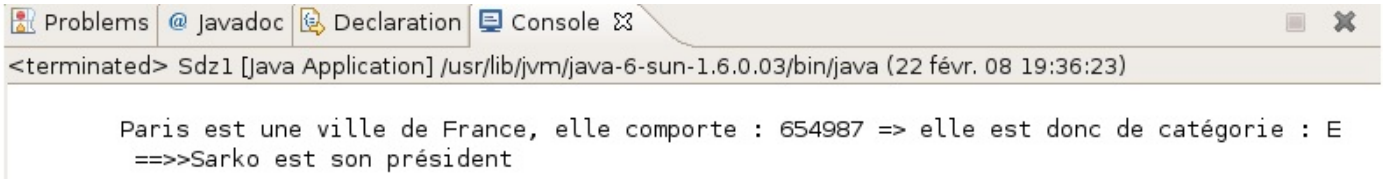
Si vous ne lui mettez aucun paramètre, `super()` renverra le constructeur par défaut de la classe `Ville`...

Testez ce code :

Code : Java

```
Capitale cap = new Capitale("Paris", 654987, "France", "Sarko");
    System.out.println("\n"+cap.decrisToi());
```

Vous devriez voir apparaître sous vos yeux ébahis : :



Je vais encore vous interpeler mais... ce que vous venez de faire sur la méthode **`decrisToi()`** s'appelle : **une méthode polymorphe**, ce qui nous conduit tout de suite à la suite !

Retour en haut

Le polymorphisme

Voici encore un des concepts fondamentaux de la programmation orientée objet : Le polymorphisme.
Ce concept complète parfaitement celui de l'héritage.

Comme vous l'avez vu, le polymorphisme n'est pas si compliqué qu'il pourrait sembler l'être !

Nous pouvons le caractériser en disant qu'il permet de manipuler des objets sans vraiment connaître leur type.

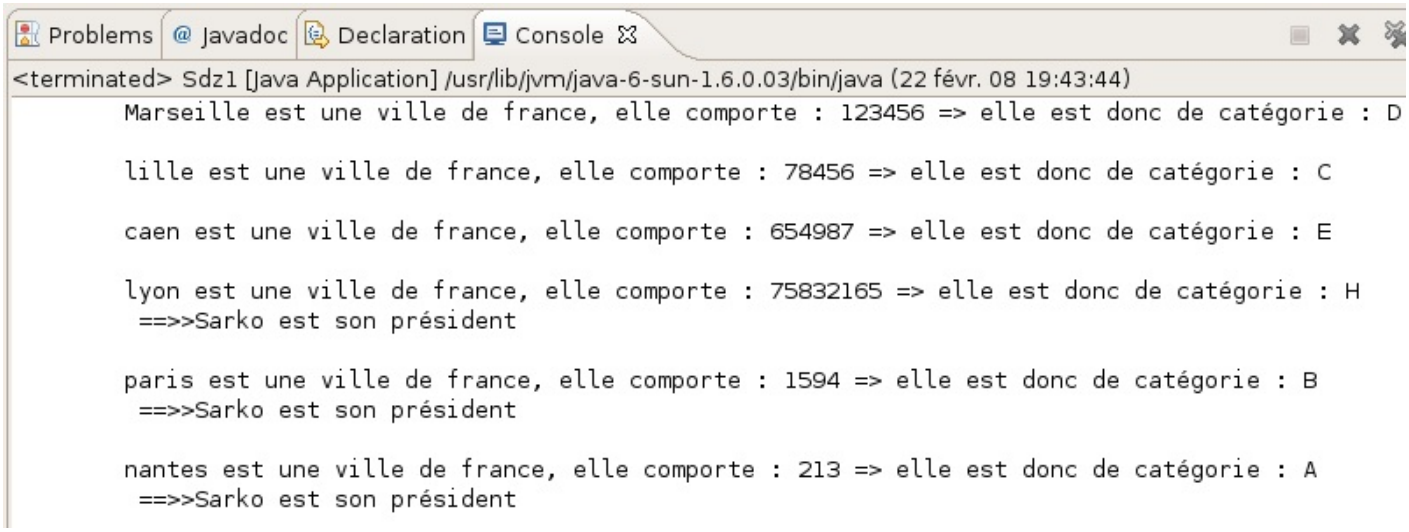
Dans notre exemple, vous avez vu qu'il suffisait d'utiliser la méthode **decrisToi()** sur un objet **Ville** ou sur un objet **Capitale**, et cela sans se soucier de leur type. On pourrait construire un tableau d'objets, et appeler la **decrisToi()** sans se soucier de son contenu : villes, capitales, ou les deux.

D'ailleurs nous allons le faire. Essayez ce code :

Code : Java

```
//Def d'un tableau de ville null
   Ville[] tableau = new Ville[6];

   //Définition d'un tableau de noms de Villes et d'un tableau de
nombres d'habitants
   String[] tab = {"Marseille", "lille", "caen", "lyon", "paris",
"nantes"};
   int[] tab2 = {123456, 78456, 654987, 75832165, 1594,213};

   /* Les 3 premiers éléments du tableau seront des Villes,
et le reste, des capitales
*/
   for(int i = 0; i < 6; i++){
     if (i <3){
       Ville V = new Ville(tab[i], tab2[i], "france" );
       tableau[i] = V;
     }

     else{
       Capitale C = new Capitale(tab[i], tab2[i], "france",
"Sarko");
       tableau[i] = C;
     }
   }

   //il ne nous reste plus qu'à décrire tout notre tableau !
   for(Ville v : tableau){
     System.out.println(v.decrisToi()+"\n");
   }
```

Résultat :



Une petite nouveauté, la création d'un tableau d'un certain nombre d'entrées **vides**. Rien de bien compliqué à cela, vous voyez que la syntaxe est toute simple.
Nous créons un tableau de villes, avec des villes et des capitales (nous avons le droit de faire ça, car les objets Capitale sont aussi des objets Ville...), dans notre première boucle **for**.
Dans la seconde, nous affichons la description de ces objets... et vous voyez que la méthode ***polymorphe*** **decrisToi()** fait bien son travail !

Dans ta boucle, tu n'utilises que des objets **Ville**.

Tout à fait. On appelle ceci la **covariance des variables** !
Cela signifie qu'une variable objet peut contenir un objet qui hérite du type de cette variable. Dans notre cas, un objet de type **Ville** peut contenir un objet de type **Capitale**. Dans ce cas, on dit que **Ville** est la **super classe** par rapport à **Capitale**.
La covariance est efficace dans le cas où la classe héritant redéfinit certaines des méthodes de sa super classe.

Attention à ne pas confondre la **surcharge de méthode** avec **une méthode polymorphe**.

Pour dire les choses simplement :

- une méthode surchargée a des paramètres que la méthode de base n'a pas, ou a le même nombre de paramètres, mais de types différents ;
- une méthode polymorphe a un squelette identique à celle de base, mais un traitement différent. Celle-ci fait référence à une autre classe et donc, par extension, à une autre instance de cette classe. **On peut dire que les méthodes polymorphes sont typiques des classes héritées !**

Vous devez savoir encore une chose sur l'héritage. Lorsque vous créez une classe (Ville par exemple), celle-ci est une classe héritée de la classe Object présente dans Java.
Cette écriture est donc tout à fait correcte :

Code : Java

```
class Ville extends Object{
..........
}
```

Toutes nos classes héritent donc des méthodes de la classe Object, comme **equals()**, qui prend un objet en paramètre, et qui permet de tester l'égalité d'objets. Vous vous en êtes d'ailleurs servis pour tester l'égalité de String() dans la première partie de ce tuto.
Donc, si nous redéfinissons une méthode de la classe **Object** dans la classe **Ville**, nous pourrions utiliser la covariance.

La méthode de la classe **Object** qui est le plus souvent redéfinie est la méthode **toString()**, qui retourne un **String** et qui a pour rôle de décrire l'objet en question (tout comme notre méthode **decrisToi()**). Nous allons donc faire un copier / coller de notre procédure de la méthode **decrisToi()** dans une nouvelle méthode de la classe **Ville** : **toString()**.

Voici :

Code : Java

```
public String toString(){
  return "\t"+this.nomVille+" est une ville de "+this.nomPays+",
elle comporte : "+this.nbreHabitant+
        " => elle est donc de catégorie : "+this.categorie;
 }
```

Nous faisons de même dans la classe **Capitale** :

Code : Java

```
public String toString(){
    String str = super.toString() + "\n \t ==>>" + this.president +
" est son président";
    return str;
  }
```

Maintenant, testez ce code :

Code : Java

```
//Def d'un tableau de ville null
   Ville[] tableau = new Ville[6];

   //Définition d'un tableau de noms de Villes et d'un tableau de
nombres d'habitants
   String[] tab = {"Marseille", "lille", "caen", "lyon", "paris",
"nantes"};
   int[] tab2 = {123456, 78456, 654987, 75832165, 1594,213};

   /* Les 3 premiers éléments du tableau seront des Villes
et le reste des capitales
*/
   for(int i = 0; i < 6; i++){
     if (i <3){
       Ville V = new Ville(tab[i], tab2[i], "france" );
       tableau[i] = V;
     }

     else{
       Capitale C = new Capitale(tab[i], tab2[i], "france",
"Sarko");
       tableau[i] = C;
     }
   }

   //il ne nous reste plus qu'à décrire tout notre tableau !
   for(Object obj : tableau){
     System.out.println(obj.toString()+"\n");
   }
```

Vous pouvez constater qu'il fait exactement la même chose que le précédent ; nous n'avons pas à nous soucier du type d'objet pour afficher sa description. Je pense que vous commencez à apercevoir la puissance de Java !

ATTENTION : si vous ne redéfinissez pas ou ne polymorphez pas une méthode d'une classe mère dans une classe fille (exemple de **toString()**), à l'appel de celle-ci avec un objet fille, c'est la méthode de la classe mère qui sera invoquée ! !

Une précision : si vous avez un objet **v** de type **Ville** par exemple, que vous n'avez pas redéfini la méthode **toString()** et que vous testez ce code :

Code : Java

```
System.out.println(v);
```

Cette instruction appelle automatiquement la méthode **toString()** de la classe **Object** ! Mais vu que vous avez redéfini la méthode **toString()** dans votre classe **Ville**, ces deux instructions sont équivalentes :

Code : Java

```
System.out.println(v.toString());
//Est équivalent à
System.out.println(v);
```

Pour plus de clarté, je conserverai la première syntaxe ! Mais vous devez savoir ceci !
En clair, vous avez accès aux méthodes **public** et **protected** de la classe **Object** dès que vous créez une classe objet (héritage tacite).
Vous pouvez donc utiliser les dites méthodes ; mais si vous ne les redéfinissez pas... l'invocation se fera sur la classe mère avec les traitements de la classe mère.

Si vous voulez un bel exemple de ce que je viens de vous dire, vous n'avez qu'à retirer la redéfinition de la méthode **toString()** dans les classes **Ville** et **Capitale** : vous verrez que le code de la méthode **main** fonctionne toujours, mais le résultat n'est plus du tout le même car, à l'appel de la méthode **toString()**, la JVM va regarder si celle-ci existe dans la classe appelante et, si elle ne la trouve pas, elle remonte dans la hiérarchie jusqu'à arriver à la classe **Object**...

Attention 2 : ce code fonctionne bien mais, si vous remplacez la méthode **toString()** par la méthode **decrisToi()**, le programme ne fonctionne plus... Et cela pour une bonne raison : la méthode **decrisToi()** n'existe pas dans la classe **Object**.

Vous devez savoir qu'une méthode est invoquable par un objet QUE si celui-ci définit ladite méthode !

Donc, ce code ne fonctionne pas :

Code : Java

```
public class Sdz1 {

        public static void main(String[] args){

                //Def d'un tableau de ville null
                   Ville[] tableau = new Ville[6];

                   //Définition d'un tableau de noms de Villes et
d'un tableau de nombres d'habitants
                   String[] tab = {"Marseille", "lille", "caen",
"lyon", "paris", "nantes"};
                   int[] tab2 = {123456, 78456, 654987, 75832165,
1594,213};

                   /* Les 3 premiers éléments du tableau seront des
Villes,
et le reste, des capitales
*/
                   for(int i = 0; i < 6; i++){
                     if (i <3){
                       Ville V = new Ville(tab[i], tab2[i], "france"
);
                       tableau[i] = V;
```

```
                        }

                        else{
                          Capitale C = new Capitale(tab[i], tab2[i],
"france", "Sarko");
                          tableau[i] = C;
                        }
                      }

                      //il ne nous reste plus qu'à décrire tout notre
tableau !
                      for(Object v : tableau){
                        System.out.println(v.decrisToi()+"\n");
                      }

        }
}
```

Pour que cela fonctionne, vous devez dire à la JVM que la référence de type **Object** est en fait une référence de type **Ville**. Comme ceci :

Code : Java

```
public class Sdz1 {

        public static void main(String[] args){

                //Def d'un tableau de ville null
                   Ville[] tableau = new Ville[6];

                   //Définition d'un tableau de noms de Villes et
d'un tableau de nombres d'habitants
                   String[] tab = {"Marseille", "lille", "caen",
"lyon", "paris", "nantes"};
                   int[] tab2 = {123456, 78456, 654987, 75832165,
1594,213};

                   /* Les 3 premiers éléments du tableau seront des
Villes,
et le reste, des capitales
*/
                   for(int i = 0; i < 6; i++){
                     if (i <3){
                       Ville V = new Ville(tab[i], tab2[i], "france"
);
                       tableau[i] = V;
                     }

                     else{
                       Capitale C = new Capitale(tab[i], tab2[i],
"france", "Sarko");
                       tableau[i] = C;
                     }
                   }

                   //il ne nous reste plus qu'à décrire tout notre
tableau !
                   for(Object v : tableau){

System.out.println(((Ville)v).decrisToi()+"\n");
                   }

        }
}
```

Vous transtypez la référence **v** en **Ville** par cette syntaxe :

Code : Java

```
((Ville)v).decrisToi();
```

Ici, l'ordre des opérations s'effectue comme ceci :

- vous transtypez la référence **v** en **Ville**
- vous appliquez la méthode **decrisToi()** à la référence appelante, ici, une référence **Object** changée en **Ville**.

Vous voyez donc l'intérêt des méthodes polymorphes. Avec celles-ci, vous n'avez plus à vous soucier du type de variable appelante ; cependant, n'utilisez le type **Object** qu'avec parcimonie.

Il existe encore un type de méthode dont je ne vous ai pas encore parlé. Il s'agit des méthodes dites **final**. Ces méthodes sont figées et vous ne pourrez **JAMAIS** redéfinir une méthode déclarée **final**. Un exemple de ce type de méthode est la méthode **getClass()** de la classe **Object** : vous ne pourrez pas redéfinir cette méthode et heureusement, car celle-ci retourne un objet **Capitale** dans le fonctionnement de Java (nous verrons cela plus tard).

Il existe aussi des classes déclarées **final**. Vous avez compris que ces classes sont immuables... Et vous ne pouvez donc pas faire hériter un objet d'une classe déclarée **final** !

Retour en haut

Ce qu'il faut retenir

Encore un petit topo des familles. Je pense qu'avec ce genre de chapitre, ce n'est pas du luxe...

- Une classe hérite d'une autre classe par le biais du mot clé `extends`.
- **Une classe ne peut hériter que d'une seule et unique classe !**
- Si nous ne définissons pas de constructeur dans nos classes, **la JVM se charge d'en créer un à l'interprétation**.
- Si aucun constructeur n'est défini dans une classe fille, **la JVM en créera un et appellera automatiquement le constructeur de la classe mère.**
- La classe fille hérite de toutes les propriétés et méthodes `public` et `protected` de la classe mère.
- Les méthodes et propriétés `private` d'une classe mère ne sont pas accessibles dans la classe fille.
- On peut redéfinir (changer tout le code) d'une méthode héritée.
- On peut utiliser le polymorphisme sur une méthode par le biais du mot clé `super`.
- Le polymorphisme permet, entre autres, d'ajouter des traitements spécifiques à une méthode en continuant d'utiliser les traitements déjà définis (utilisation de `super`).
- Grâce à l'héritage et au polymorphisme, nous pouvons utiliser la **covariance** des variables !
- Si une méthode d'une classe mère n'est pas redéfinie ou polymorphée, à l'appel de cette méthode par le biais d'un objet enfant, **c'est la méthode de la classe mère qui sera appelée !**
- Vous ne pouvez pas hériter d'une classe déclarée `final`.
- Une méthode déclarée `final` est **non redéfinissable**.

Je crois que vous êtes prêts pour un petit QCM... Mais prenez le temps de digérer tout ceci !
Faites des essais, testez, comparez, c'est le meilleur moyen de bien comprendre les choses.
Retour en haut
Si vous êtes prêts pour la suite, allons tout de suite voir ce qui se passe : ***Apprendre à modéliser***.
Retour en haut

Apprendre à modéliser

Dans ce chapitre nous allons voir le principe de modélisation d'objet.
Ne vous leurrez pas, ça sera assez indigeste, mais faites-moi confiance...

UML, mais qu'est-ce donc ?

UML est le sigle signifiant **U**nified **M**odeling **L**anguage : traduisez par "*langage de modélisation unifié*".
Il ne s'agit pas d'un langage de programmation mais plutôt d'une méthodologie de modélisation comme la méthode *merise*, etc.

À quoi ça sert ?

Je sais que vous êtes des Zér0s avertis en matière de programmation, ainsi qu'en informatique en général, mais mettez-vous dans la peau d'une personne totalement dénuée de toute connaissance dans le domaine.
Il fallait trouver un langage commun entre les commerciaux, les responsables de projets informatiques, les développeurs afin que tout ce petit monde se comprenne. Avec UML, c'est le cas.

En fait, avec UML, vous pouvez modéliser toutes les parties de développement d'une application informatique, de sa conception à la mise en route, tout ceci grâce à des diagrammes. Il est vrai que certains de ces diagrammes sont plus adaptés à des informaticiens, mais il en existe qui permettent de voir comment interagit l'application dans son contexte de fonctionnement... Et dans ce genre de cas, la connaissance de l'entreprise pour laquelle l'application est prévue est de mise. On utilise donc un mode de communication commun à tout le monde : UML.

Il existe bien sûr des outils de modélisation afin de créer de tels diagrammes. Personnellement, j'utilise argoUML mais il en existe d'autres, comme :

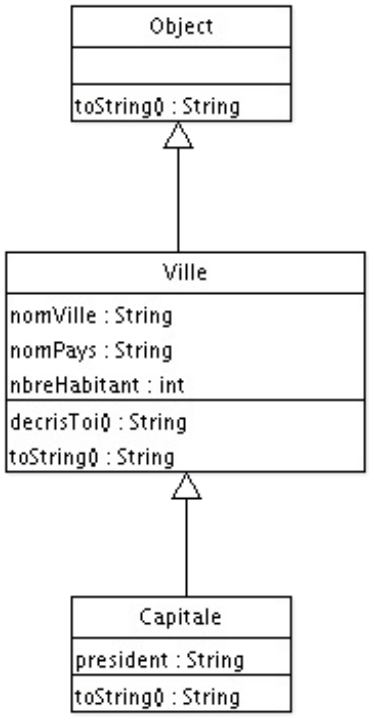
- boUML
- Together
- Poseidon
- Pyut
- ...

ArgoUML a le mérite d'être gratuit et fait en Java... donc multi-plates-formes.

Avec ces outils, vous pouvez réaliser les différents types de diagrammes qu'UML vous propose :

- diagramme de ***use case*** (cas d'utilisation) : permet de déterminer les différents cas d'utilisation d'un programme informatique ;
- diagramme de classes : celui dont nous allons nous servir ; permet de modéliser des classes ainsi que les interactions entre elles ;
- des diagrammes de séquences : permettent de visualiser le déroulement d'une application dans un contexte donné ;
- et d'autres encore...

Voici un exemple de diagramme de classe :

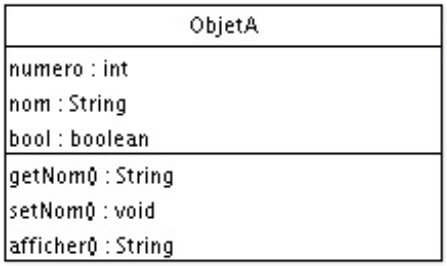


Vous avez dû remarquer qu'il s'agissait des classes que nous avons utilisées lors des chapitres précédents. Je ne vous cache pas non plus qu'il s'agit d'une version simplifiée... En effet, vous pouvez constater que je n'ai pas mis toutes les méthodes déclarées **`public`** de la classe **`Object`** ainsi que des classes que nous avons codées.
Je ne vais pas vous apprendre à utiliser argoUML non plus, mais plutôt à savoir lire un diagramme car, dans certains cas, il s'avère pratique de modéliser les classes et l'interaction entres celles-ci. Ne serait-ce que pour avoir plus de recul sur notre travail. Mais aussi parce qu'il y a des concepts de programmation qu'il est plus facile d'expliquer avec un diagramme qu'avec de longs discours...

Retour en haut

Modéliser un objet

À présent, nous allons apprendre à lire un diagramme de classes.
Vous avez deviné qu'une classe est modélisée sous cette forme :



Voici une classe nommée **`ObjetA`** qui a comme attributs :

- `numero` de type **`int`**
- `nom` de type **`String`**
- et `bool` de type **`boolean`**.

Et comme méthodes :

- `getNom()` qui retourne une chaîne de caractères
- `setNom()` qui ne renvoie rien
- `afficher()` qui renvoie elle aussi une chaîne de caractères.

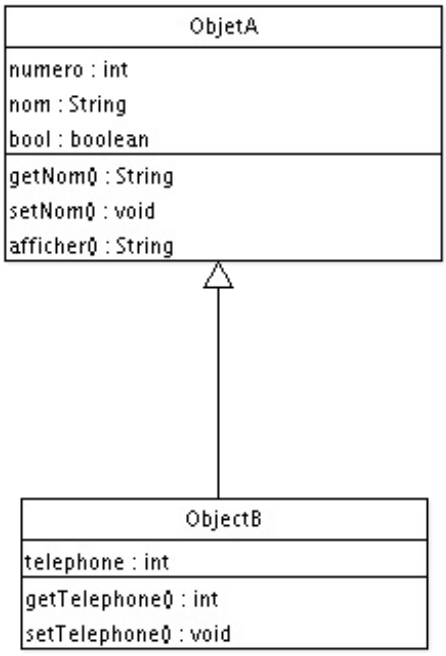
La portée des attributs et des méthodes n'est pas modélisée ici...
Vous voyez que la modélisation d'un objet est toute simple et très compréhensible !

Maintenant, voyons les interactions entre objets.
Retour en haut

Modéliser les interactions entre objets

Vous allez voir que les interactions sont aussi très simples à modéliser.
En fait, comme vous avez pu le voir sur l'exemple, les interactions sont modélisées par des flèches de types différents. Nous allons voir maintenant celles que nous pouvons d'ores et déjà utiliser, dans l'état actuel de nos connaissances (au fur et à mesure, nous verrons d'autres flèches).

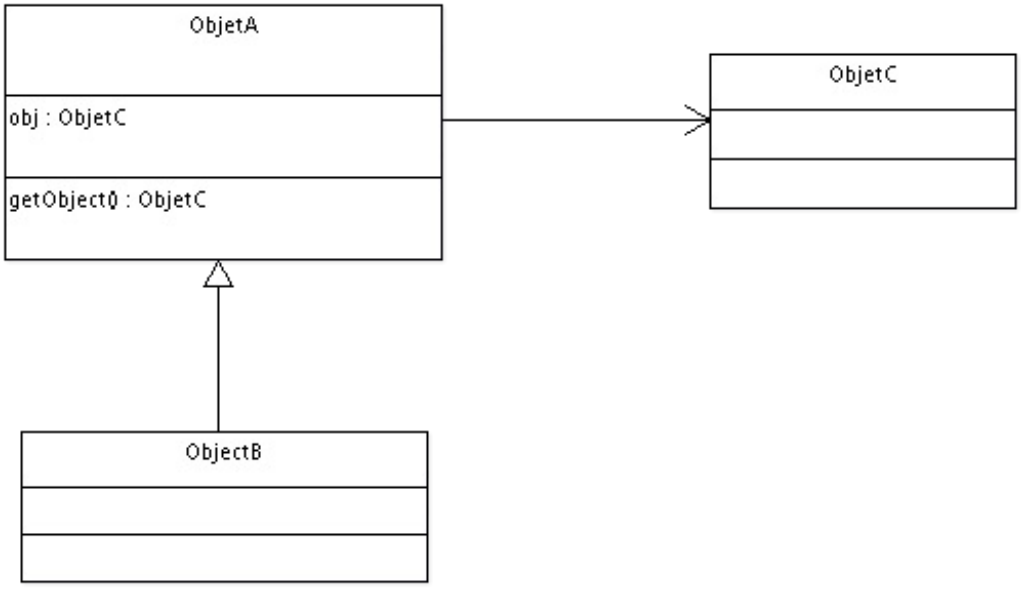
Regardez ceci :



Sur ce diagramme, vous pouvez voir un deuxième objet qui a lui aussi des paramètres. Mais ne vous y trompez pas, **ObjetB** possède aussi les attributs et les méthodes de la classe **ObjectA**. Et d'après vous, pourquoi ?
Car la flèche qui relie nos deux objets signifie "**extends**". En gros, vous pouvez lire ce diagramme comme suit :

l'**ObjetB hérite de** l'**ObjetA**, ou encore **ObjetB est un objetA**.

Nous allons voir une autre flèche d'interaction. Je sais que nous n'avons pas encore vu ce cas de figure, mais il est simple à comprendre.
Comme nous pouvons mettre des objets de type **String** dans des classes que nous développons, nous pouvons aussi mettre comme variable d'instance, ou de classe, un objet que nous avons codé. Voici un diagramme modélisant ce cas de figure :



Dans cet exemple simpliste, vous voyez que nous avons toujours notre héritage entre un objet A et un objet B mais dans ce cas,

l'**ObjetA** (et donc l'**ObjetB**) ont une variable de classe de type **ObjetC**, ainsi qu'une méthode ayant un type de retour **ObjetC** (car la méthode va retourner un **ObjetC**).
Vous pouvez lire ce diagramme comme suit :

l'**ObjetA a un ObjetC**.
Ici, il n'y a qu'un seul **objetC** : "**a UN**".

Voici le code Java correspondant à ce diagramme.

Fichier ObjetA.java

Code : Java

```
public class ObjetA{
   protected ObjetC obj = new ObjetC();

   public ObjetC getObject(){
     return obj;
   }
}
```

Fichier ObjetB.java

Code : Java

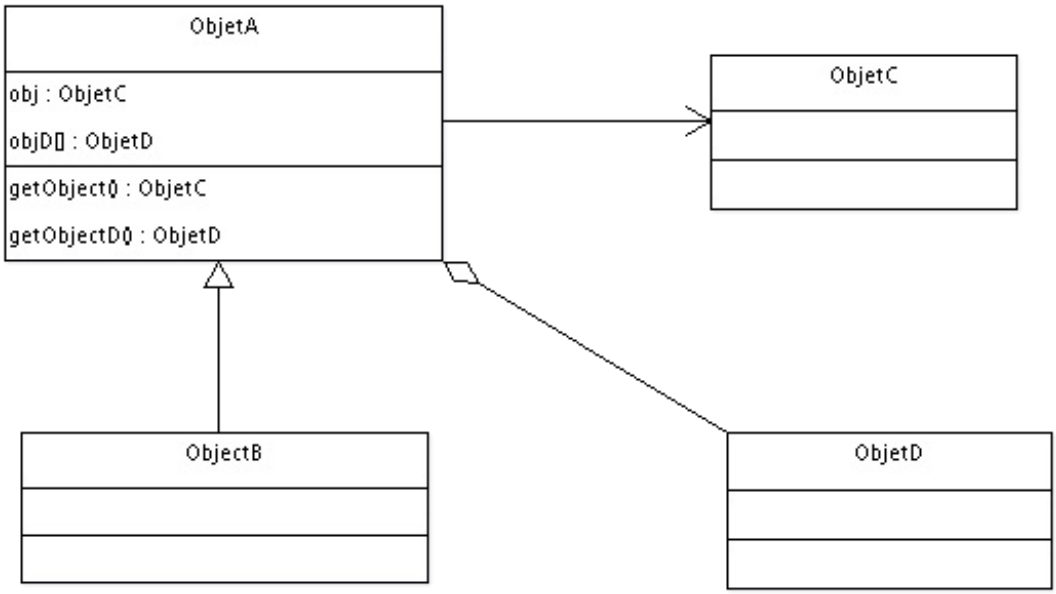
```
public class ObjetB extends ObjetA{

}
```

Fichier ObjetC.java

Code : Java

```
public class ObjetC{

}
```

Il y a encore une dernière flèche que nous pouvons voir car **elle ne diffère** que légèrement de la première.
Voici un diagramme la mettant en oeuvre :



Nous avons ici le même diagramme que précédemment, à l'exception de l'**ObjetD**. Ici, nous devons lire le diagramme comme suit :
l'**ObjetA est composé d'ObjetD**.
Ici, il y aura plusieurs instances d'**ObjetD** dans **ObjetA**.

Vous pouvez d'ailleurs remarquer que la variable d'instance correspondante est de type tableau...

Voici le code Java correspondant :

Fichier ObjetA.java

Code : Java

```
public class ObjetA{
   protected ObjetC obj = new ObjetC();
   protected ObjetD[] objD = new ObjetD[10];

   public ObjetC getObject(){
     return obj;
   }

   public ObjectD[] getObjectD(){
      return objD;
   }
}
```

Fichier ObjetB.java

Code : Java

```
public class ObjetB extends ObjetA{

}
```

Fichier ObjetC.java

Code : Java

```
public class ObjetC{
```

```
}
```

*Fichier **ObjetD.java***

Code : Java

```
public class ObjetD{

}
```

Il est bien évident que ces classes ne font strictement rien.. Mais je les ai utilisées à titre d'exemple pour la modélisation...

Voilà, c'en est fini pour le moment. Attendez-vous donc à avoir des diagrammes dans vos prochains chapitres...
Il n'y aura pas de QCM car j'estime qu'il n'y a rien de difficile ici.
Retour en haut
Après ce que nous avons vu au cours de ce chapitre et des précédents, nous allons tout de suite voir les ***classes abstraites*** !
Retour en haut

Les classes abstraites

Nous voici de retour dans l'un des fondements du langage Java.
En effet, vous verrez ici que les classes abstraites sont vraiment pratiques.
Sans plus attendre : let's go !

Qu'est-ce que c'est ?

Ne vous en faites pas... Ici ***abstraite*** n'a rien de commun avec un certain mouvement de peinture.
En fait, une classe ***abstraite*** est quasiment comme une classe normale. Oui, comme une classe que vous avez maintenant l'habitude de coder.
Ceci dit, elle a tout de même une particularité :

vous ne pouvez pas l'instancier !

Vous avez bien entendu . Imaginons que nous ayons une classe **A** déclarée **abstraite**. Ce code ne compilera pas :

Code : Java

```
public class Test{

   public static void main(String[] args){

       A obj = new A();//Erreur de compilation ! !

   }
}
```

À quoi ça sert, alors ?

J'attendais cette question... C'est pour ça que je n'ai pas commencé directement par un exemple de classe abstraite.
Tout d'abord, je vais vous donner un exemple de situation (de programme, en fait).
Imaginez que vous êtes en train de réaliser un programme qui gère différents types d'animaux (oui, je sais : l'exemple est bête, mais il a le mérite d'être simple à comprendre).

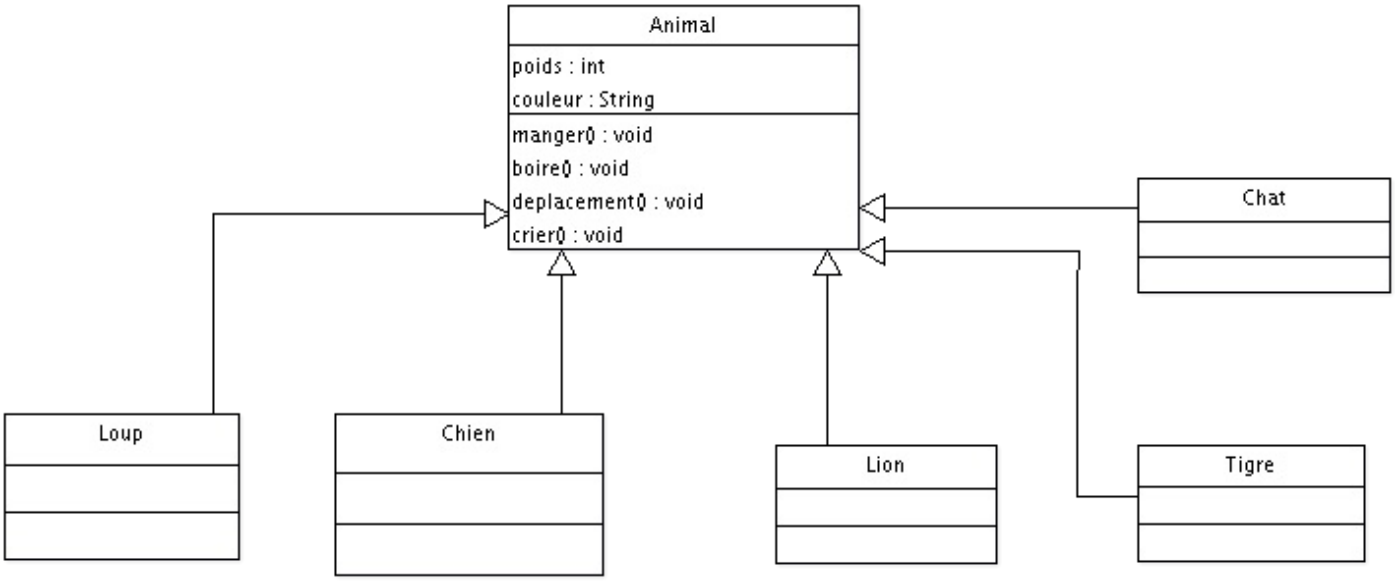
Dans ce programme, vous avez :

- des loups
- des chiens
- des chats
- des lions
- des tigres.

Je pense tout de même que vous n'allez pas faire toutes vos classes bêtement... il va de soi que tous ces animaux ont des choses en commun ! Et qui dit chose en commun... dit héritage.
Que pouvons-nous définir de commun à tous ces animaux, sinon :

- une couleur
- un poids
- qu'ils crient
- qu'ils se déplacent
- qu'ils mangent
- qu'ils boivent.

Nous pouvons donc faire une classe mère, appelons-la **`Animal`**.
Avec ce que nous avons dégagé de commun, nous pouvons lui définir des attributs et des méthodes. Voici donc à quoi pourraient ressembler nos classes pour le moment :



Nous avons bien notre classe mère **`Animal`** et nos animaux qui en héritent.
À présent, laissez-moi vous poser une question.

Vu que notre classe **`Animal`** est `public` dans notre cas, qu'est-ce qu'est censé faire un objet **`Animal`** ? Quel est son poids, sa couleur, que mange-t-il ?

Si nous avons un morceau de code qui ressemble à ceci :

Code : Java

```
public class Test{
   public static void main(String[] args){
      Animal ani = new Animal();
      ani.manger();//Que doit-il faire ? ?
   }
}
```

...personnellement, je ne sais pas comment mange un objet **`Animal`**...
Vous conviendrez que toutes les classes ne sont pas bonnes à être instanciées !

C'est là que rentrent en jeu nos classes abstraites. En fait, ces classes servent à définir une ***super classe***.

D'accord, mais comment on empêche une classe d'être instanciable puisque tu nous a dit que la JVM déclare un constructeur par défaut... On ne peut donc pas omettre le constructeur !

Tout à fait. Pour répondre à cette question : suivez le guide !

Retour en haut

Une classe Animal très abstraite

En fait, il existe une règle pour qu'une classe soit considérée comme abstraite. Elle doit être déclarée avec le mot clé **abstract**.

Voici un exemple illustrant mes dires :

Classe déclarée abstraite :

Code : Java

```
abstract class Animal{

}
```

Une telle classe peut avoir le même contenu qu'une classe normale. Ses enfants pourront utiliser tous ses éléments déclarés (attributs et méthodes) **public**. Cependant, ce type de classe permet de définir des méthodes abstraites. Ces méthodes ont une particularité ; elles n'ont pas de corps !

En voici un exemple :

Code : Java

```
abstract class Animal{
   abstract void manger();//une méthode abstraite
}
```

Vous voyez pourquoi on dit "***méthode abstraite***", difficile de voir ce que cette méthode sait faire...

Retenez bien qu'une méthode abstraite n'est composée que de l'entête de la méthode suivie d'un point-virgule : ;

Jusque-là ça va, mais concrètement, à quoi ça sert ?

Tout d'abord, vous devez savoir qu'une méthode abstraite ne peut exister que dans une classe abstraite. Si dans une classe, vous avez une méthode déclarée abstraite, vous **DEVEZ DÉCLARER CETTE CLASSE COMME ETANT ABSTRAITE**.

Maintenant voyons à quoi cela peut servir. Vous avez vu les avantages de l'héritage et du polymorphisme. Dans ce cas, nos classes enfants hériteront aussi des classes abstraites mais, vu que celles-ci n'ont pas de corps, nos classes enfants seront **OBLIGÉES de redéfinir ces méthodes !**
De ce fait, nos classes enfants auront des méthodes polymorphes en leur sein et donc, la covariance des variables repointe le bout de son nez...
La covariance appliquée aux classes abstraites donne ceci :

Code : Java

```
public class Test{

   public static void main(String args[]){

     Animal loup = new Loup();
     Animal chien = new Chien();
     loup.manger();
     chien.crier();

   }
}
```

Attends ! Tu nous as dit qu'on ne pouvait instancier de classe abstraites ?

Et je maintiens mes dires. Ici, nous n'avons pas instancié notre classe abstraite. Nous avons instancié un objet **Loup** que nous avons mis dans un objet de type **Animal** : il en va de même pour l'instanciation de la classe **Chien**.

Vous devez vous rappeler que l'instance se crée avec le mot clé **new**. En aucun cas le fait de déclarer une variable d'un type de classe donnée (ici, `Animal`) est une instanciation ! Ici nous instancions un **Loup** et un **Chien**.

Vous pouvez aussi utiliser une variable de type **Object** comme référence pour un objet **Loup**, un objet **Chien**...
Vous savez déjà que ce code fonctionne :

Code : Java

```
public class Test{
  public static void main(String[] args){
     Object obj = new Loup();
  }
}
```

Par contre, ceci pose problème :

Code : Java

```
public class Test{
  public static void main(String[] args){
     Object obj = new Loup();
     Loup l = obj;//Problème de référence
  }
}
```

Eh oui ! Vous essayez ici de mettre une référence de type **Object** dans une référence de type **Loup** . Pour avertir la JVM que la référence que vous voulez mettre dans votre objet de type **Loup** est un **Loup**, vous devez utiliser le transtypage !
Revoyons notre code :

Code : Java

```
public class Test{
  public static void main(String[] args){
     Object obj = new Loup();
     Loup l = (Loup)obj;//Vous prévenez la JVM que la référence que
vous passez est de type Loup
  }
}
```

Vous pourrez bien évidemment instancier directement un objet **Loup**, un objet **Chien** ou tout autre.

Pour le moment, nous n'avons aucun code dans aucune classe ! Les exemples que je vous ai fournis ne font rien du tout, mais ils seront censés fonctionner lorsque nous aurons mis des morceaux de code dans nos classes.

À présent, étoffons nos classes et notre diagramme avant d'avoir un code qui fonctionne bien !

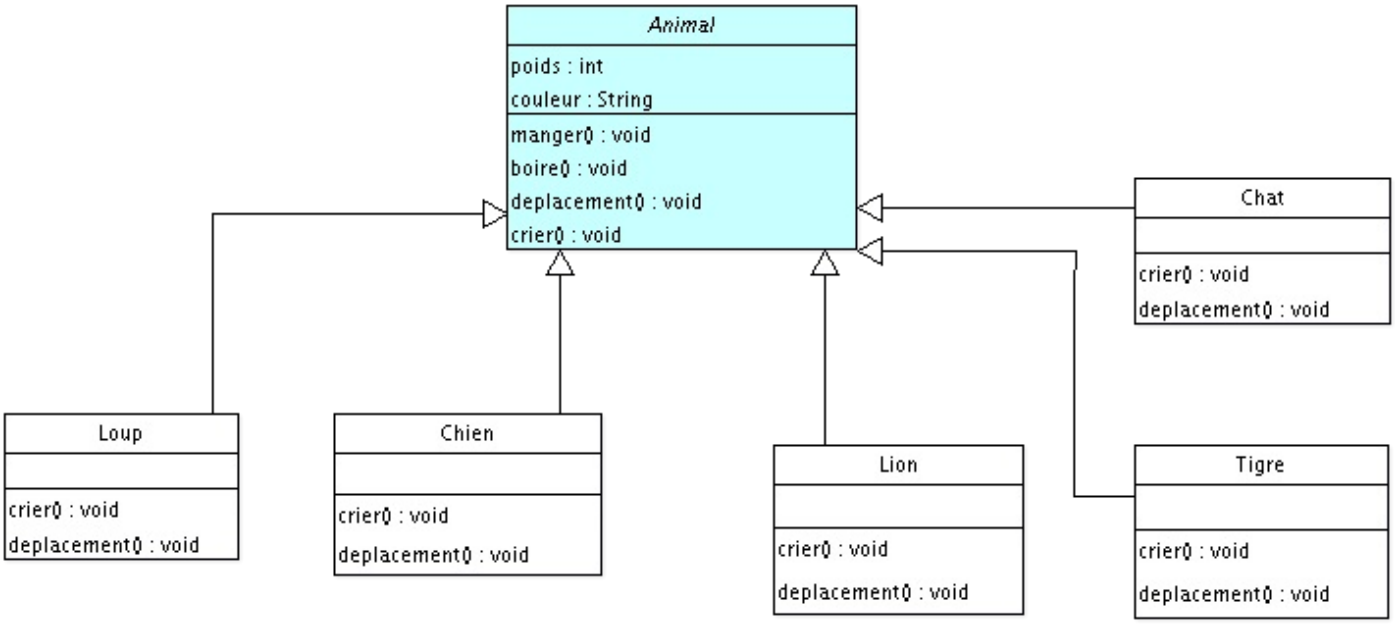
Retour en haut

Étoffons notre exemple

Nous allons donc rajouter des morceaux de code à nos classes.
Tout d'abord, voyons le topo de ce que nous savons.

- Nos objets auront tous une couleur et un poids différents. Nos classes auront donc le droit de modifier ceux-ci.
- Ici, nous partons du principe que tous nos animaux mangeront de la viande. La méthode **`manger()`** sera donc définie dans la classe Animal.
- Idem pour la méthode **`boire()`**. Ils boiront tous de l'eau .
- Par contre, ils ne crient pas et ne se déplaceront pas de la même manière. Nous ferons donc des méthodes polymorphes et déclarerons les méthodes **`crier()`** et **`deplacement()`** abstraites dans la classe `Animal`.

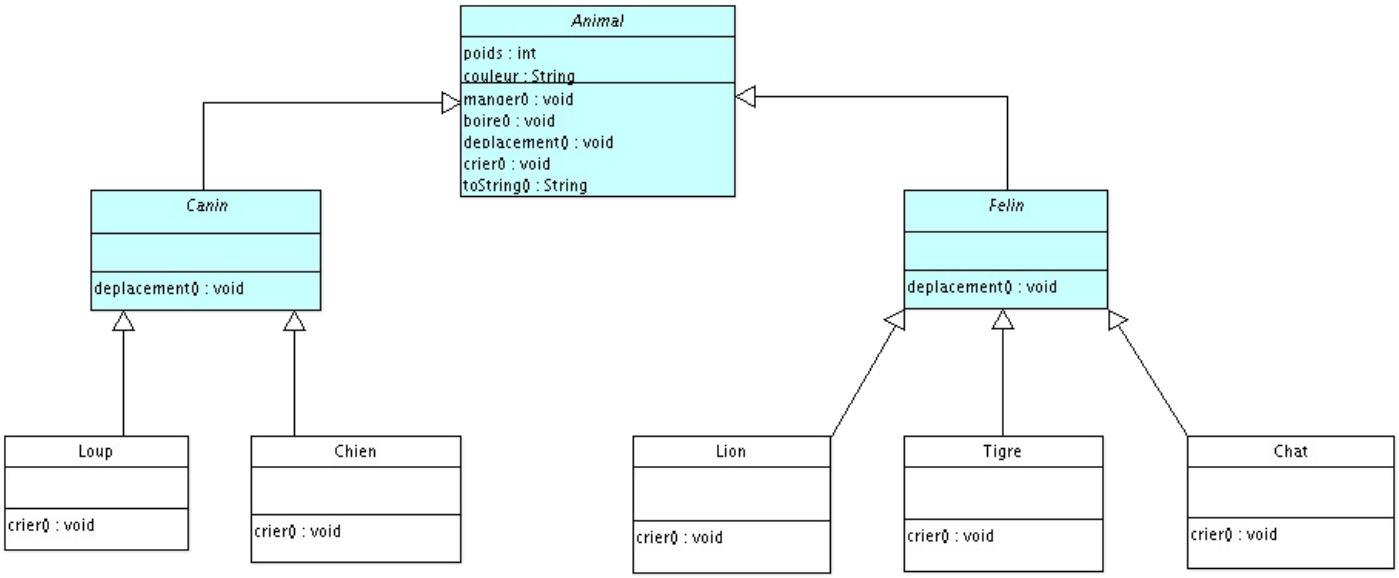
Voici ce que donneraient nos classes :



J'ai colorié la classe abstraite en bleu mais il y un autre moyen de savoir si une classe est abstraite :
le nom de celle-ci est en italique.
Je ne sais pas si c'est une convention ou non, mais argoUML la différencie de cette façon !

Nous voyons bien que notre classe `Animal` est déclarée abstraite et que nos classes filles héritent de celle-ci. De plus, nos classes filles ne redéfinissent que deux méthodes sur quatre, on en conclut ici que ces deux méthodes doivent être abstraites. Nous ajouterons deux constructeurs à nos classes filles, un par défaut, ainsi qu'un avec les deux paramètres d'initialisation. À ceci nous ajouterons aussi les accesseurs d'usage. Cependant... nous pouvons améliorer un peu cette architecture, sans pour autant rentrer dans les détails !

Vu les animaux présents, nous aurions pu faire une sous-classe **`Carnivore`**, ou encore **`AnimalDomestique`** et **`AnimalSauvage`**... Ici, nous allons nous contenter de faire deux sous-classes **`Canin`** et **`Felin`** qui hériteront d'**`Animal`** et dont nos objets hériteront !
Nous allons redéfinir la méthode **`deplacement()`** dans cette classe car nous allons partir du principe que les félins se déplacent d'une certaine façon, et les canins d'une autre. Avec cet exemple, nous réviserons le polymorphisme...
Voilà notre diagramme mis à jour :



Vous avez vu ? J'ai ajouté une méthode **`toString()`** :D.

Voici les codes Java correspondant :

Animal.java

Code : Java

```
abstract class Animal {

	/**
* La couleur de l'animal
*/
	protected String couleur;
	/**
* Le poids
*/
	protected int poids;

	/**
* La méthode manger
*/
	protected void manger(){
		System.out.println("Je mange de la viande");
	}

	/**
* La méthode boire
*/
	protected void boire(){
		System.out.println("Je bois de l'eau !");
	}

	/**
* La méthode de déplacement
*/
	abstract void deplacement();
	/**
* La méthode de cri
*/
	abstract void crier();

	public String toString(){

		String str = "Je suis un objet de la " +
```

```
		this.getClass() + ", je suis " + this.couleur + ", je pèse " +
		this.poids;
                        return str;
                }

        }
```

Felin.java

Code : Java

```
        public abstract class Felin extends Animal {

                @Override
                void deplacement() {
                        System.out.println("Je me déplace seul !");
                }

        }
```

Canin.java

Code : Java

```
        public abstract class Canin extends Animal {

                @Override
                void deplacement() {
                        System.out.println("Je me déplace en meute !");
                }

        }
```

Chien.java

Code : Java

```
        public class Chien extends Canin {

                public Chien(){

                }
                public Chien(String couleur, int poids){
                        this.couleur = couleur;
                        this.poids = poids;
                }


                void crier() {
                        System.out.println("J'aboie sans raison ! ");
                }

        }
```

Loup.java

Code : Java

```
public class Loup extends Canin {

	public Loup(){

	}
	public Loup(String couleur, int poids){
		this.couleur = couleur;
		this.poids = poids;
	}

	void crier() {
		System.out.println("J'hurle à la lune en faisant
ouhouh ! ! ");
	}
}
```

Lion.java

Code : Java

```
public class Lion extends Felin {

	public Lion(){

	}
	public Lion(String couleur, int poids){
		this.couleur = couleur;
		this.poids = poids;
	}

	void crier() {
		System.out.println("Je rugis dans la savane !");
	}

}
```

Tigre.java

Code : Java

```
public class Tigre extends Felin {

	public Tigre(){

	}
	public Tigre(String couleur, int poids){
		this.couleur = couleur;
		this.poids = poids;
	}

	void crier() {
		System.out.println("Je grogne très fort !");
	}
```

```
}
```

Chat.java

Code : Java

```
public class Chat extends Felin {


        public Chat(){

        }
        public Chat(String couleur, int poids){
                this.couleur = couleur;
                this.poids = poids;
        }

        void crier() {
                System.out.println("Je miaule sur les toits !");
        }

}
```

Dis donc ! Une classe abstraite ne doit pas avoir une méthode abstraite ?

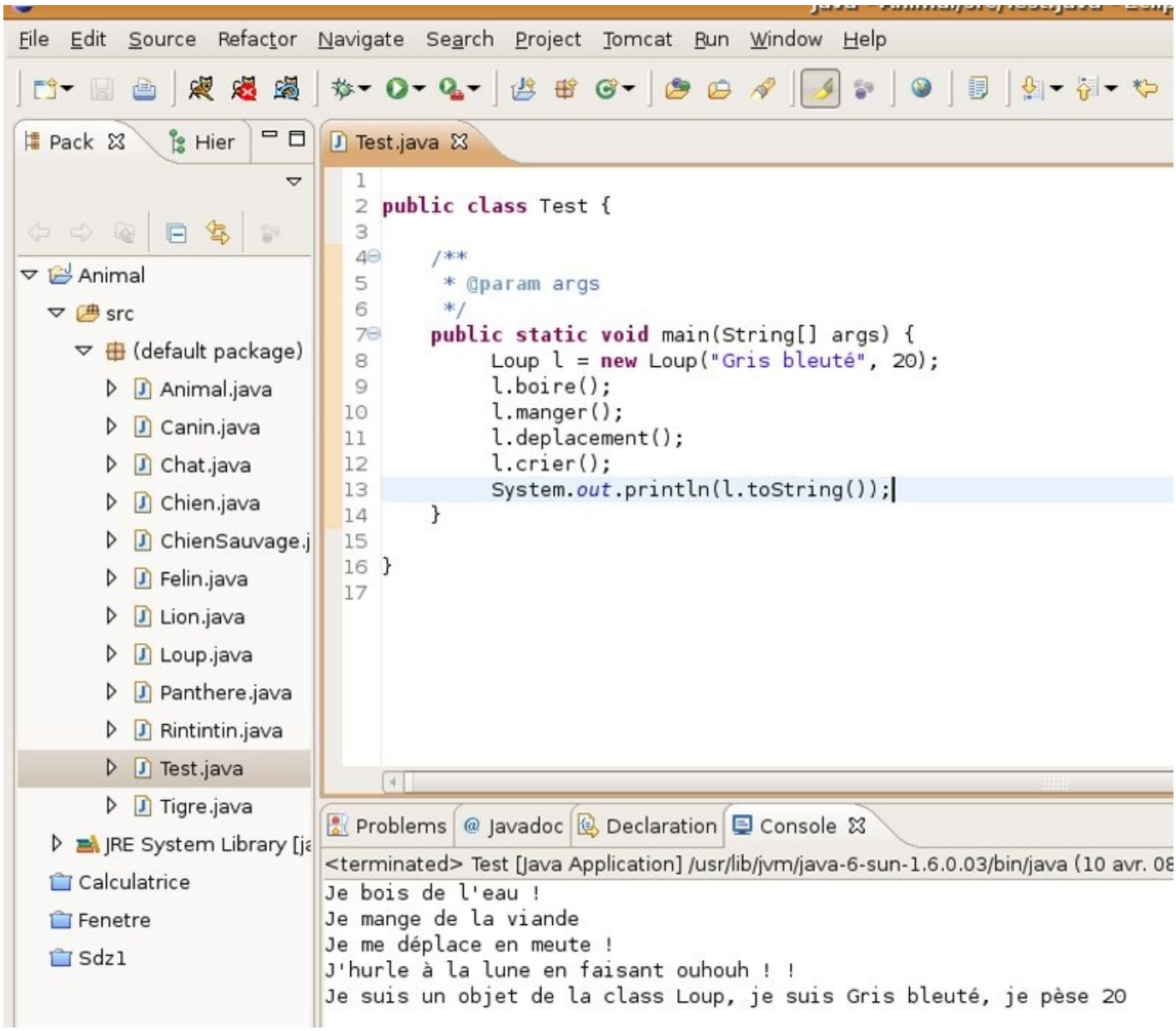
Je n'ai jamais dis ça ! Une classe déclarée abstraite n'est plus instanciable, mais elle n'est nullement obligée d'avoir des méthodes abstraites !
En revanche, une classe ayant une méthode abstraite doit être déclarée abstraite !

Maintenant que vous avez toutes vos classes,
faites des tests. Autant que vous le voulez.
Dans cet esprit :

Code : Java

```
public class Test {

        /**
* @param args
*/
        public static void main(String[] args) {
                Loup l = new Loup("Gris bleuté", 20);
                l.boire();
                l.manger();
                l.deplacement();
                l.crier();
                System.out.println(l.toString());
        }

}
```

Voilà le jeu d'essai de ce code :



Dans la méthode **`toString()`** de la classe **`Animal`**, j'ai utilisé la méthode **`getClass()`** qui *-je vous le donne en mille-* est dans la classe **`Object`**. Celle-ci retourne "`class <nom de la classe>`".

Dans cet exemple, nous pouvons voir que nous avons un objet **`Loup`**.

- À l'appel de la méthode **`boire()`** : l'objet appelle la méthode de la classe **`Animal`**.
- À l'appel de la méthode **`manger()`** : idem.
- À l'appel de la méthode **`toString()`** : idem.
- À l'appel de la methode **`deplacement()`** : c'est la méthode de la classe **`Canin`** qui est invoquée ici.
- À l'appel de la méthode **`crier()`** : c'est la méthode de la classe **`Loup`** qui est appelée.

Remplacez le type de référence (ici, `Loup`) par **`Animal`** ou **`Object`**. Essayez avec des objets **`Chien`**, etc. Et vous verrez que tout fonctionne, excepté que vous ne pourrez pas instancier d'**`Object`**, de **`Felin`** ou de **`Canin`** !

Avant de partir en quête d'un QCM, je crois qu'une nouvelle astuce d'eclispe est disponible !

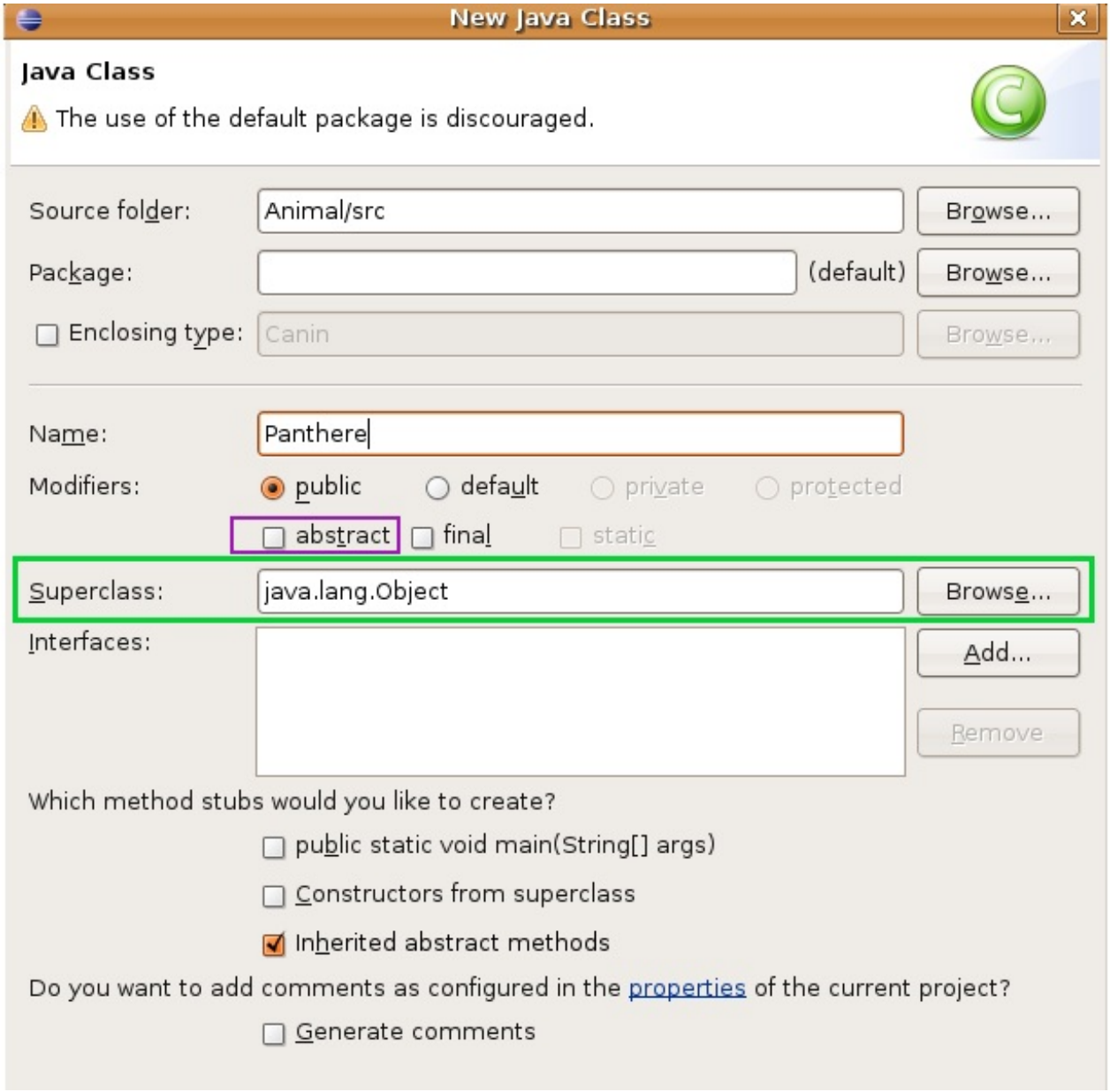
Retour en haut

Astuce d'Eclipse

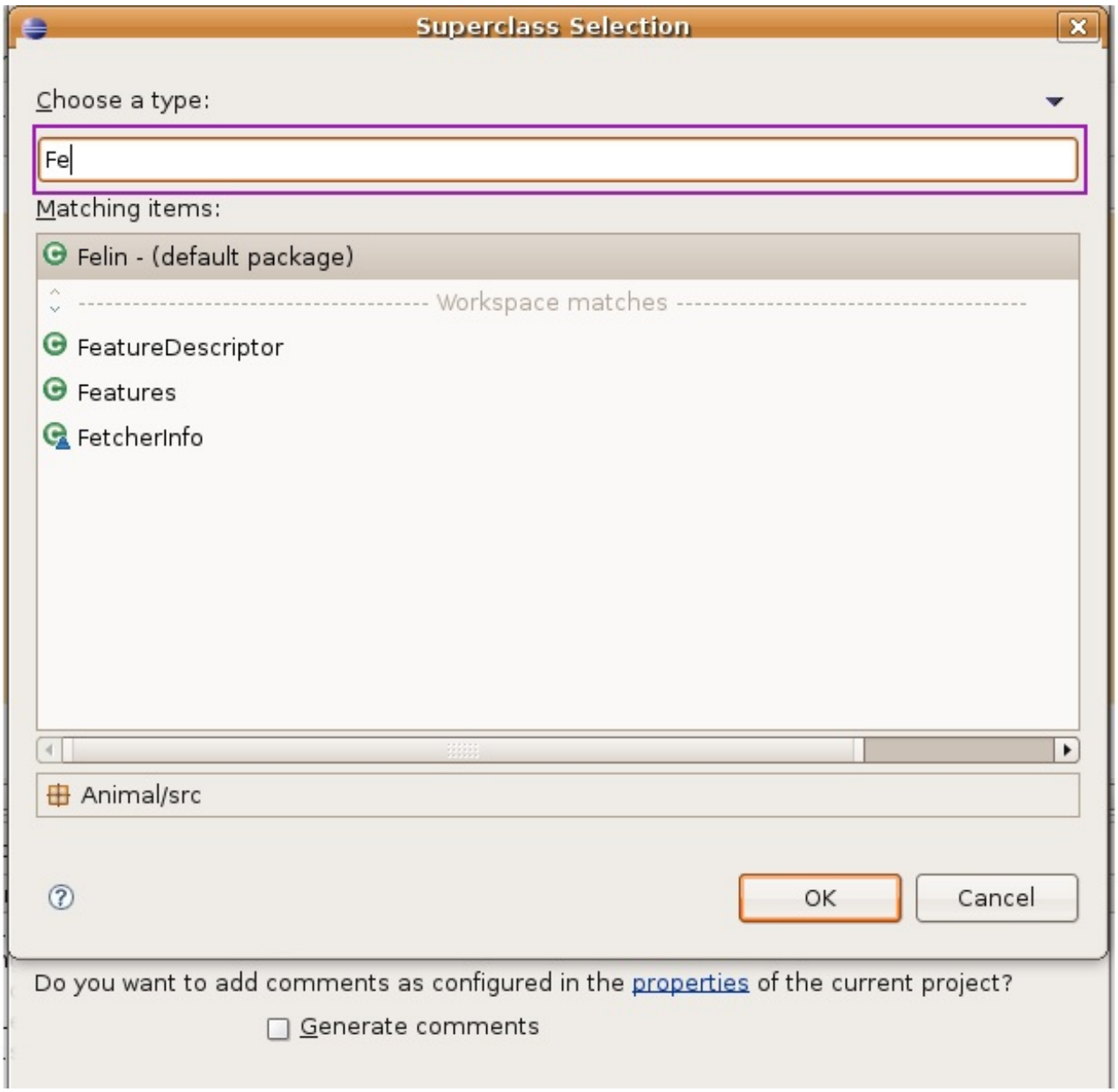
Lorsque vous créez votre classe et plus particulièrement vos classes héritées, Eclipse peut gérer la gestion des mots clé.
Il peut :

- ajouter le mot clé `extends` avec la classe mère
- ajouter les méthodes à redéfinir dans le cas d'une classe abstraite.

Voyez comment faire. Nous allons faire une classe **`Panthere`** héritée de **`Felin`**.
Une fois que vous avez inscrit le nom de votre classe, regardez plus bas : vous pouvez voir le nom de la super classe de votre nouvelle classe :



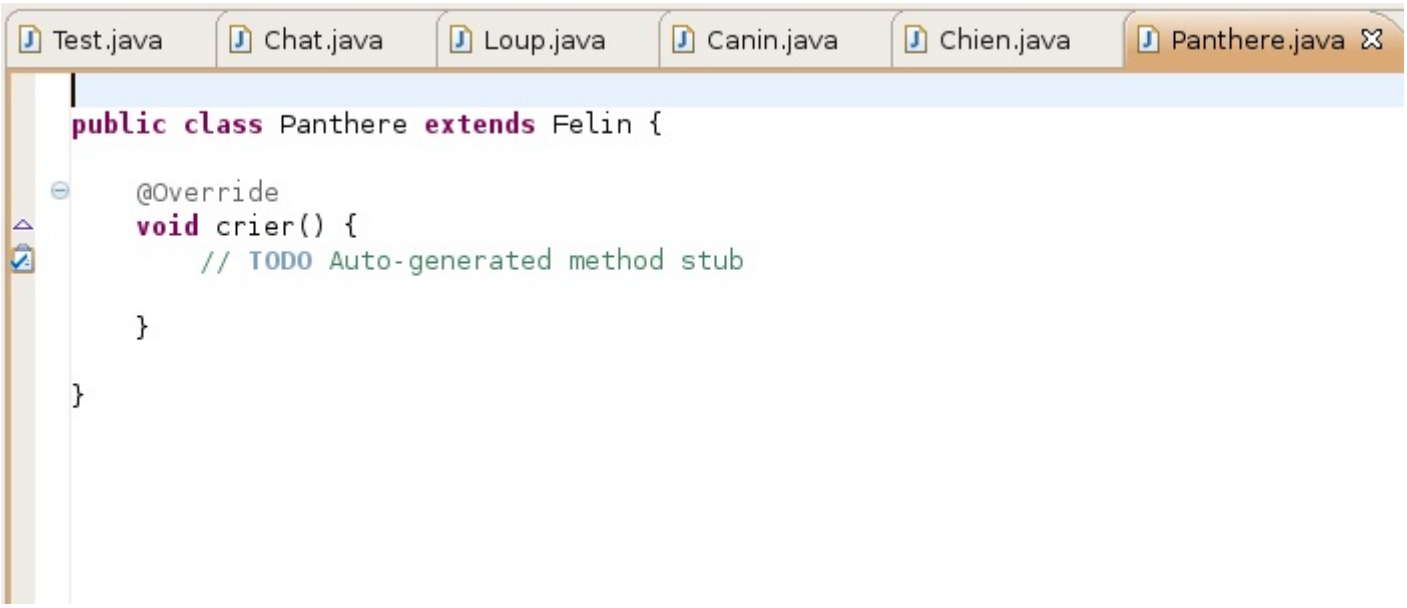
En cochant la case abstract, votre nouvelle classe sera abstraite. Et, comme vous pouvez le voir, votre classe hérite par défaut de **`Object`**. Pour changer ceci, cliquez sur Browse. Vous arrivez sur cette fenêtre ; ici, il vous suffit de mettre le nom de la classe mère, comme ceci :



Vous devez taper le nom complet de votre classe. Ici, comme nous n'avons pas encore utilisé de ***package***, il suffit de taper **`Felin`**. En plus, Eclipse gère l'auto-complétion, ce qui signifie qu'il termine tous les noms des classes en vous proposant une liste exhaustive. Il vous suffit donc de choisir la classe mère, et de valider.

Lorsque la classe mère est dans un package, vous devez saisir le nom complet de la classe. Par exemple, si vous voulez faire une classe héritée de la classe **`Scanner`**, vous devrez commencer par saisir : **`java.util.Scanner`**.

Terminez l'étape et voici le résultat :



Vous voyez que l'héritage est bon et que les méthodes de la classe abstraite sont mises automatiquement !
Pratique, n'est-ce pas ?

Et au fait, que veut dire @Override ?

L'annotation `@Override` doit être utilisée lorsqu'une méthode redéfinit la méthode de la super classe.
Celle-ci existe dans la classe mère mais on redéfinit son contenu dans la classe fille.
Cependant, dans notre exemple, c'est Eclipse qui rajoute automatiquement cette annotation, mais, si vous redéfinissez une méthode d'une classe mère manuellement, vous pouvez l'ajouter vous même, tout en sachant que **ceci n'est pas obligatoire** !

Allez, en avant pour le topo.
Retour en haut

Ce qu'il faut retenir

Avec les classes abstraites, vous devez vous rappelez ceci :

- une classe est définie comme abstraite avec le mot clé `abstract`.
- Une classe abstraite **ne peut pas être instanciée**.
- Une classe abstraite n'est pas obligée de contenir de méthode abstraite.
- Si une classe contient une méthode abstraite, cette classe doit alors être déclarée abstraite.
- Une méthode abstraite n'a pas de corps.
- Les classes abstraites sont à utiliser lorsque qu'une classe mère ne doit pas être instanciée.

Un petit topo ici... Bon... Un QCM vous attend.

Retour en haut

Maintenant, veuillez aller à la partie suivante : **Les interfaces.**

Retour en haut

Les interfaces

Voici donc l'un des autres fondements du langage : les **interfaces** !

Il ne s'agit pas ici d'interfaces graphiques...

Mais vous allez apercevoir ce que les programmeurs Java appellent le nec plus ultra du polymorphisme !
Je suis sûr que vous devez être impatients de voir ce point... Alors, qu'attendons nous ?

Une petite devinette

L'un des atouts majeurs, pour ne pas dire l'atout majeur, de la programmation orientée objet est la réutilisabilité de vos objets. Il est bien commode de pouvoir utiliser un objet, voire même une architecture que nous avons développée dans une application nouvelle.

Admettons que l'architecture que nous avons développée dans les chapitres précédents forme une bonne base. Que se passerait-il si un autre développeur vous demande la possibilité d'utiliser vos objets dans un autre type d'application ? Dans la nôtre, nous ne nous sommes occupés que de l'aspect générique des animaux que nous avons créés. Cependant, la personne qui vous a contacté, lui, développe une application pour un chenil.

La principale contrainte, c'est que vos chiens vont devoir apprendre à faire de nouvelles choses comme :

- faire le beau,
- faire des calins,
- faire une léchouille.

Je ne vois pas le problème ! Tu n'as qu'à ajouter ces méthodes dans la classe **`Animal`**.

Oula ! Vous vous rendez bien compte que vous allez avoir des lions qui vont faire le beau ?

Dans ce cas, on n'a qu'à mettre ces méthodes dans la classe **`Chien`** !

Ceci pourrait être une solution, mais j'y vois deux contre-indications :

- vous allez devoir mettre en oeuvre une convention de nommage entre vous et le programmeur qui va utiliser vos objets... Vous ne devrez pas utiliser une méthode ***faireCalin()*** alors que le programmeur utilise une méthode **`faireUnCalin()`** ;
- si vous faites ceci. **ADIEU LE POLYMORPHISME** ! Vous ne pourrez pas appeler vos objets par le biais d'un super type. Pour pouvoir accéder à ces méthodes, vous devrez obligatoirement passer par une référence à un objet **`Chien`**. Pas terrible, tout ça...

Tu nous as dit que pour utiliser au mieux le polymorphisme, nous devions définir les méthodes au plus haut niveau de la hiérarchie.

Tout à fait.

Alors du coup, il faut redéfinir un super type pour pouvoir utiliser le polymorphisme !

Oui, et je vous rappelle que l'héritage multiple est interdit en Java. Et quand je dis interdit, je veux dire que Java ne gère pas ça ! Nous sommes sur la bonne voie. Il faudrait pouvoir développer un nouveau super type et s'en servir dans nos classes **`Chien`**. Eh bien nous pouvons faire cela avec des **interfaces**.

En fait, les interfaces permettent de créer un nouveau super type ; on peut même en ajouter autant que nous le souhaitons dans une seule classe !
En plus, concernant l'utilisation de nos objets, la convention est toute trouvée... Pourquoi ? Parce qu'une interface n'est rien d'autre qu'une classe 100 % abstraite !
Allez : venons-en au fait !

Retour en haut

Votre première interface

Pour définir une interface, au lieu d'écrire :

Code : Java

```
public class A{

}
```

il vous suffit de faire :

Code : Java

```
public interface I{

}
```

Voilà, vous venez d'apprendre à déclarer une interface. Maintenant, tant qu'à y ajouter des méthodes, vu qu'une interface est une classe 100 % abstraite, vous n'avez qu'à y mettre des méthodes abstraites. Mais sans mettre le mot clé **abstract**.
Voici des exemples d'interfaces :

Code : Java

```
public interface I{

   public void A();
   public String B();

}
```

Code : Java

```
public interface I2{

   public void C();
   public String D();

}
```

Et pour faire en sorte qu'une classe utilise une interface, il suffit d'utiliser le mot clé **implements**.
Ce qui nous donnerait :

Code : Java

```
public class X implements I{
    public void A(){
      //.......
    }

    public String B(){
      //.......
    }
}
```

Voilà, c'est tout. **On dit que la classe X implémente l'interface I**.
Et donc, comme je vous le disais, vous pouvez implémenter plusieurs interfaces, voilà donc comment ça se passe :

Code : Java

```
public class X implements I, I2{
   public void A(){
      //.......
    }

    public String B(){
      //.......
    }

    public void C(){
      //.......
    }

    public String D(){
      //.......
    }
}
```

Par contre, lorsque vous implémentez une ou plusieurs interfaces, vous **DEVEZ OBLIGATOIREMENT** redéfinir leurs méthodes !

Et ainsi, avec le polymorphisme, vous pouvez faire ceci :

Code : Java

```
public class Test{

   public static void main(String[] args){
      I var = new X();//Avec cette référence, vous pouvez utiliser
de façon polymorphe les méthodes de l'interface I
      I2 var2 = new X();//Avec cette référence, vous pouvez utiliser
de façon polymorphe les méthodes de l'interface I2
      var.A();
      var2.C();

   }

}
```

Maintenant que vous savez cela, attelons-nous à notre programme.

Retour en haut

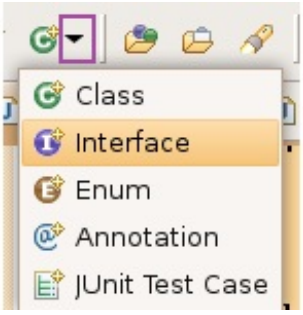
Implémentation de l'interface Rintintin

Voilà où nous en sommes

- Nous voulons que nos chiens puissent être amicaux.
- Nous voulons définir un super type pour pouvoir utiliser le polymorphisme.
- Nous voulons pouvoir continuer à utiliser nos objets comme avant.

Comme le titre de cette sous-partie le stipule, nous allons créer l'interface **Rintintin** pour ensuite l'implémenter dans notre objet **Chien**.

Sous Eclipse, vous pouvez faire File / New / Interface, ou simplement cliquer sur la flèche noire à côté du C pour la création de classe, et choisir interface.



Voici son code :

Code : Java

```
public interface Rintintin{

   public void faireCalin();
   public void faireLechouille();
   public void faireLeBeau();

}
```

À présent, il ne nous reste plus qu'à implémenter l'interface dans notre classe **Chien**. Ce qui nous donne :

Code : Java

```
public class Chien extends Canin implements Rintintin {

        public Chien(){

        }
        public Chien(String couleur, int poids){
                this.couleur = couleur;
                this.poids = poids;
        }


        void crier() {
                System.out.println("J'aboie sans raison ! ");
        }

        public void faireCalin() {
                System.out.println("Je te fais un GROS CÂLIN");

        }

        public void faireLeBeau() {
```

```
                System.out.println("Je fais le beau !");

        }

        public void faireLechouille() {
                System.out.println("Je fais de grosses
léchouilles...");

        }

}
```

L'ordre des déclarations est PRIMORDIAL. Vous DEVEZ mettre l'expression d'héritage AVANT l'expression d'implémentation, SINON votre code ne compilera pas !

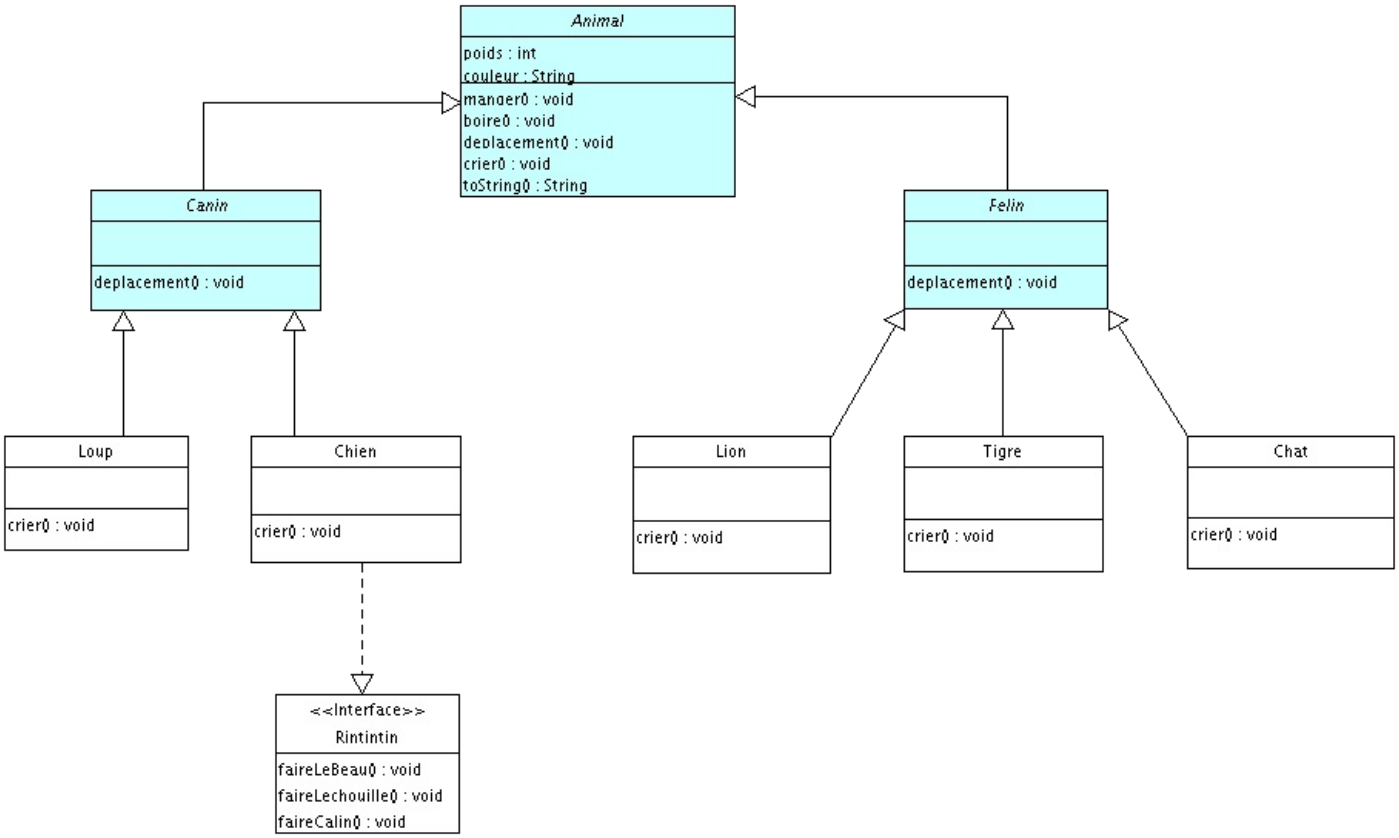
Voici un code que vous pouvez utiliser pour tester le polymorphisme de notre implémentation :

Code : Java

```
public class Test {

        /**
* @param args
*/
        public static void main(String[] args) {
                //Les méthodes d'un chien
                Chien c = new Chien("Gris bleuté", 20);
                c.boire();
                c.manger();
                c.deplacement();
                c.crier();
                System.out.println(c.toString());

                System.out.println("------------------------------
------------");
                //Les méthodes de l'interface
                c.faireCalin();
                c.faireLeBeau();
                c.faireLechouille();

                System.out.println("------------------------------
------------");
                //Utilisons le polymorphisme de notre interface
                Rintintin r = new Chien();
                r.faireLeBeau();
                r.faireCalin();
                r.faireLechouille();

        }

}
```

Objectif atteint !

Nous avons réussi à définir deux super classes afin de les utiliser comme super types et de jouir pleinement du polymorphisme ! Nous allons voir au chapitre suivant qu'il y a une façon d'utiliser les interfaces très intéressante ! Mais pour le moment, voyons un peu à quoi ressemble notre diagramme, à présent :



Nous voyons bien notre interface **Rintintin** ainsi que ses méthodes, et notre flèche pointillée qui se dirige vers notre interface signifie "**implémente**".
Nous pouvons lire **Chien implemente Rintintin**.

Bon : c'est l'heure d'une petite astuce de notre IDE préféré, il me semble...

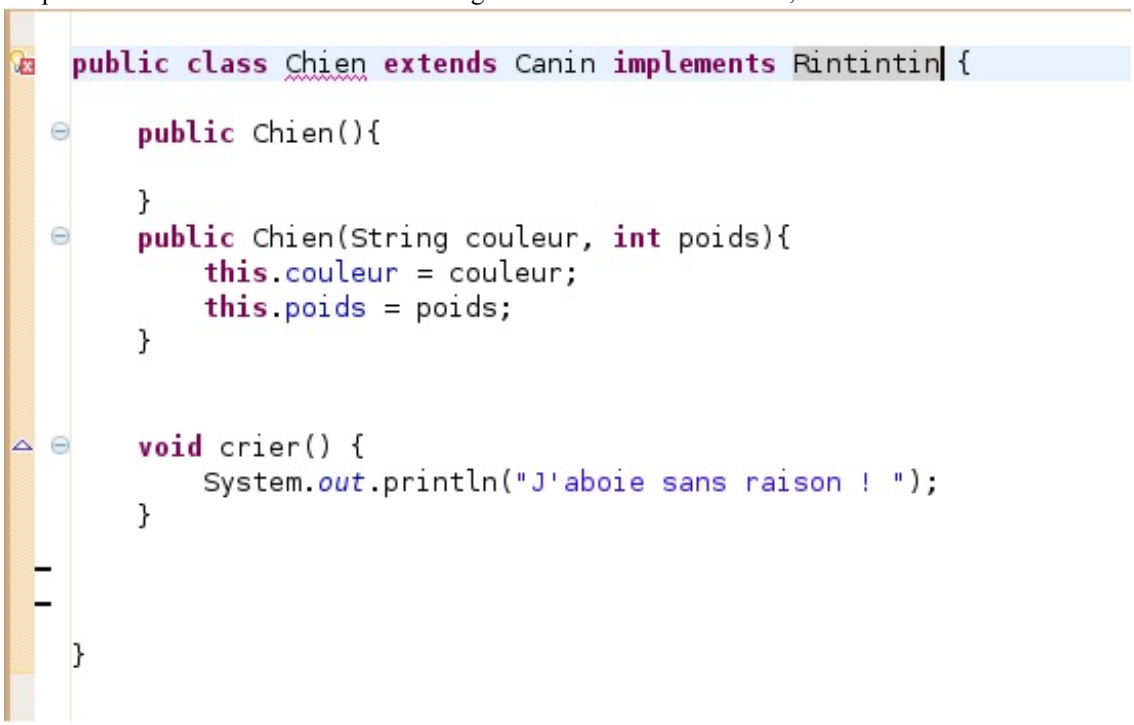
Retour en haut

Astuce d'Eclipse

Astuce 1

Je vous ai expliqué comment créer directement une interface.
Mais vous avez sans doute remarqué qu'une erreur persistante apparaît lorsque vous saisissez **`implements Rintintin`** après la déclaration de votre classe.

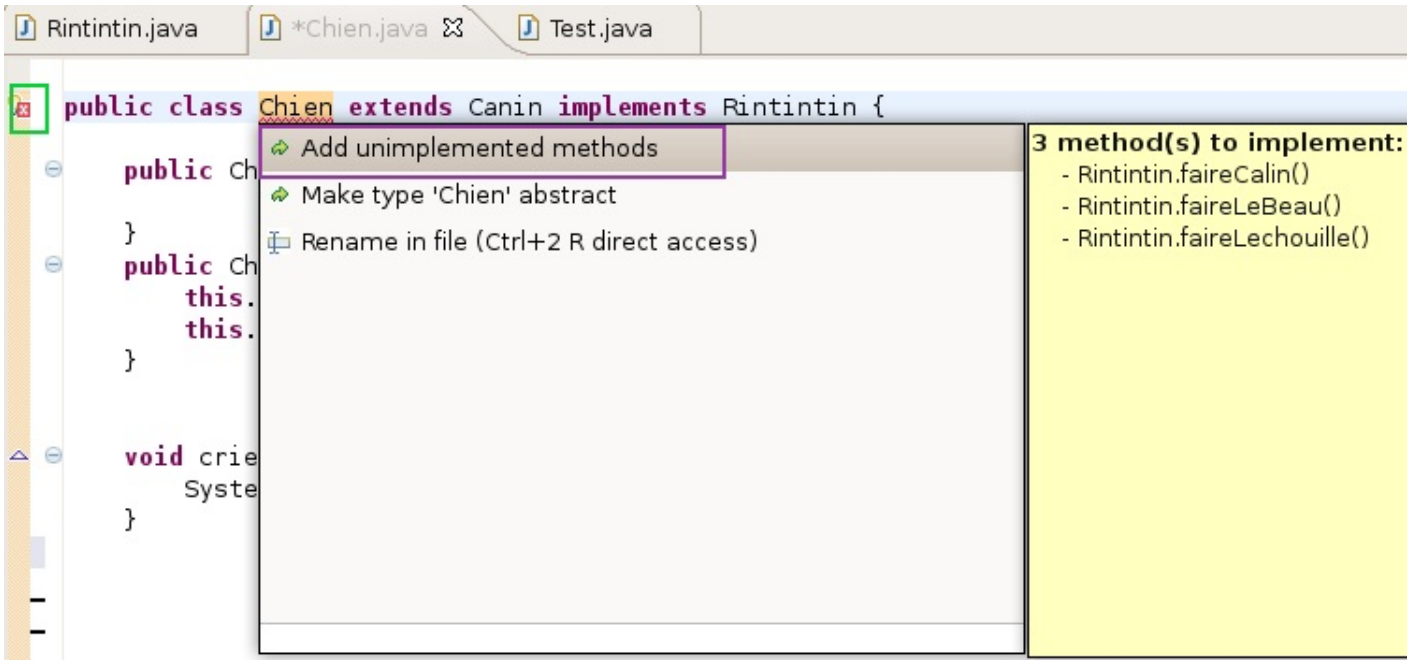
Je pense même que vous devez avoir une zolie croix rouge à côté du nom de la classe, comme celle-ci :



```
public class Chien extends Canin implements Rintintin {

    public Chien(){

    }
    public Chien(String couleur, int poids){
        this.couleur = couleur;
        this.poids = poids;
    }


    void crier() {
        System.out.println("J'aboie sans raison ! ");
    }


}
```

Cette erreur est dûe au fait que vous n'avez pas implémenté les méthodes de l'interface !

Cliquez sur cette croix rouge et choisissez l'option add unimplemented method : Eclipse ajoutera automatiquement toutes les méthodes à implémenter par la classe.

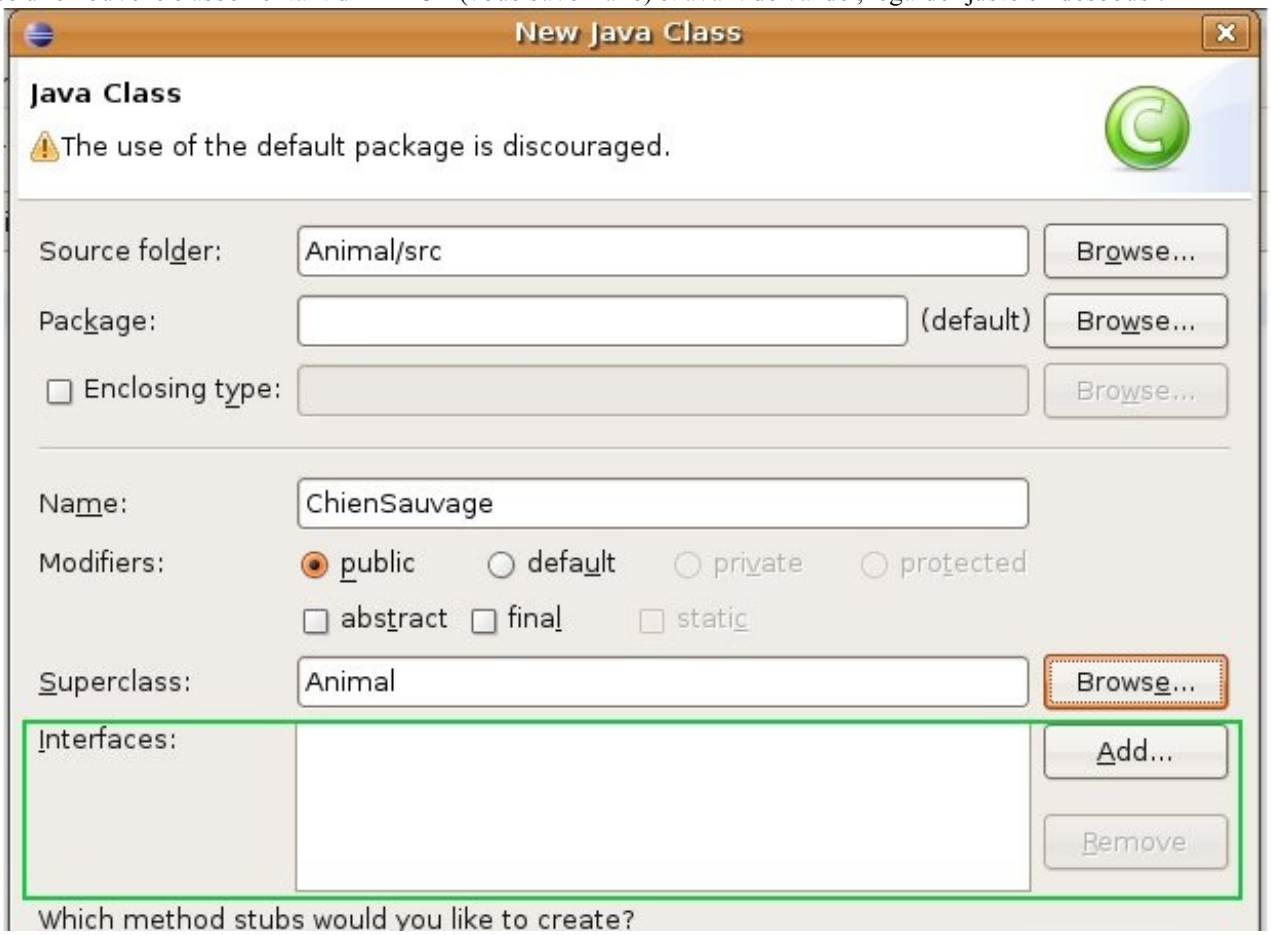


Astuce 2

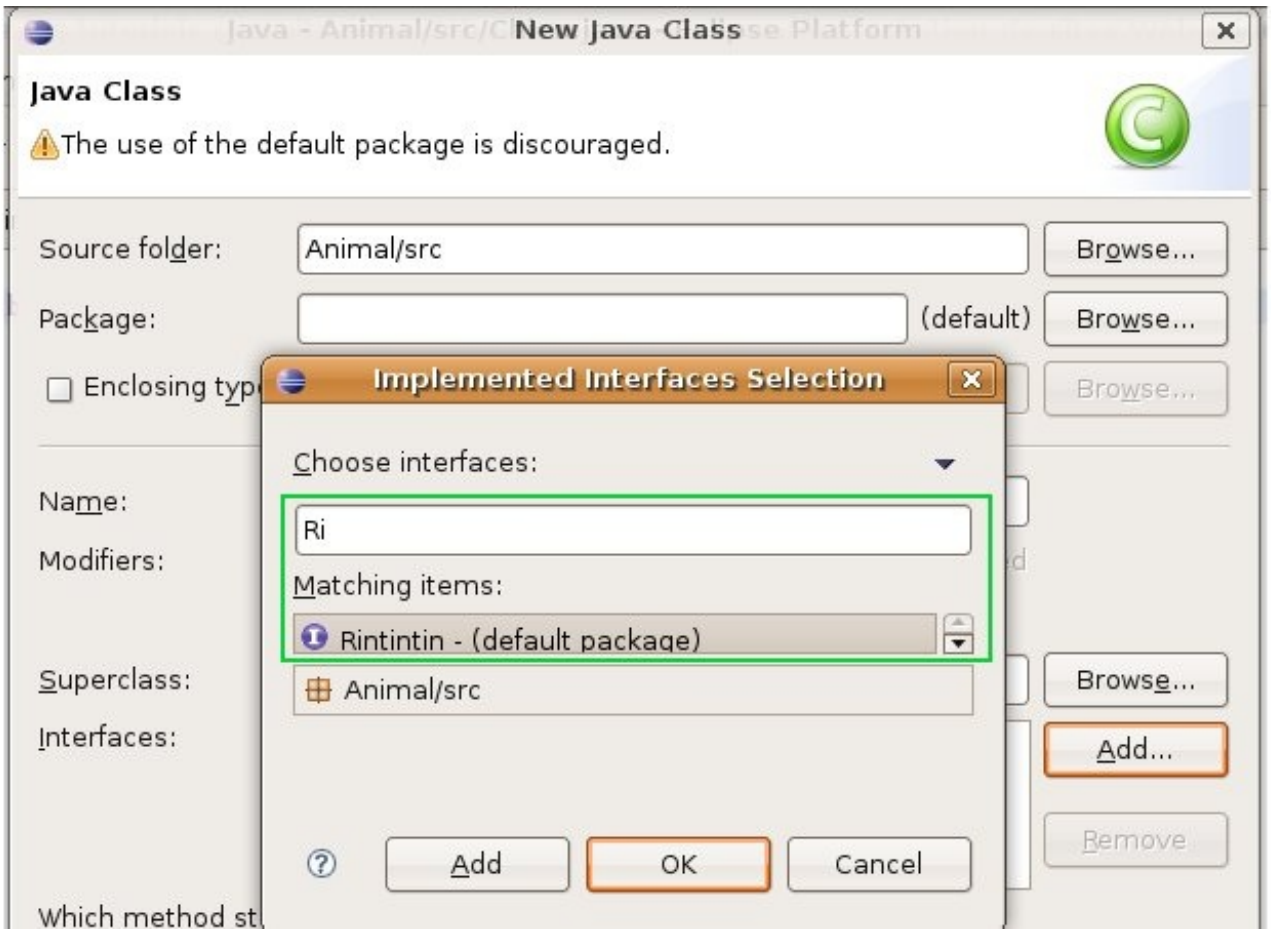
Je vais vous demander de faire appel à de vieux souvenirs provenant du chapitre précédent. Rappelez-vous l'astuce d'Eclipse afin de définir la super classe pour l'héritage. Eh bien Eclipse fait la même chose pour les interfaces...

Si, si, je vous assure. Regardez plutôt comment on crée une classe **`ChienSauvage`** héritant d' **`Animal`** et implémentant **`Rintintin`**.

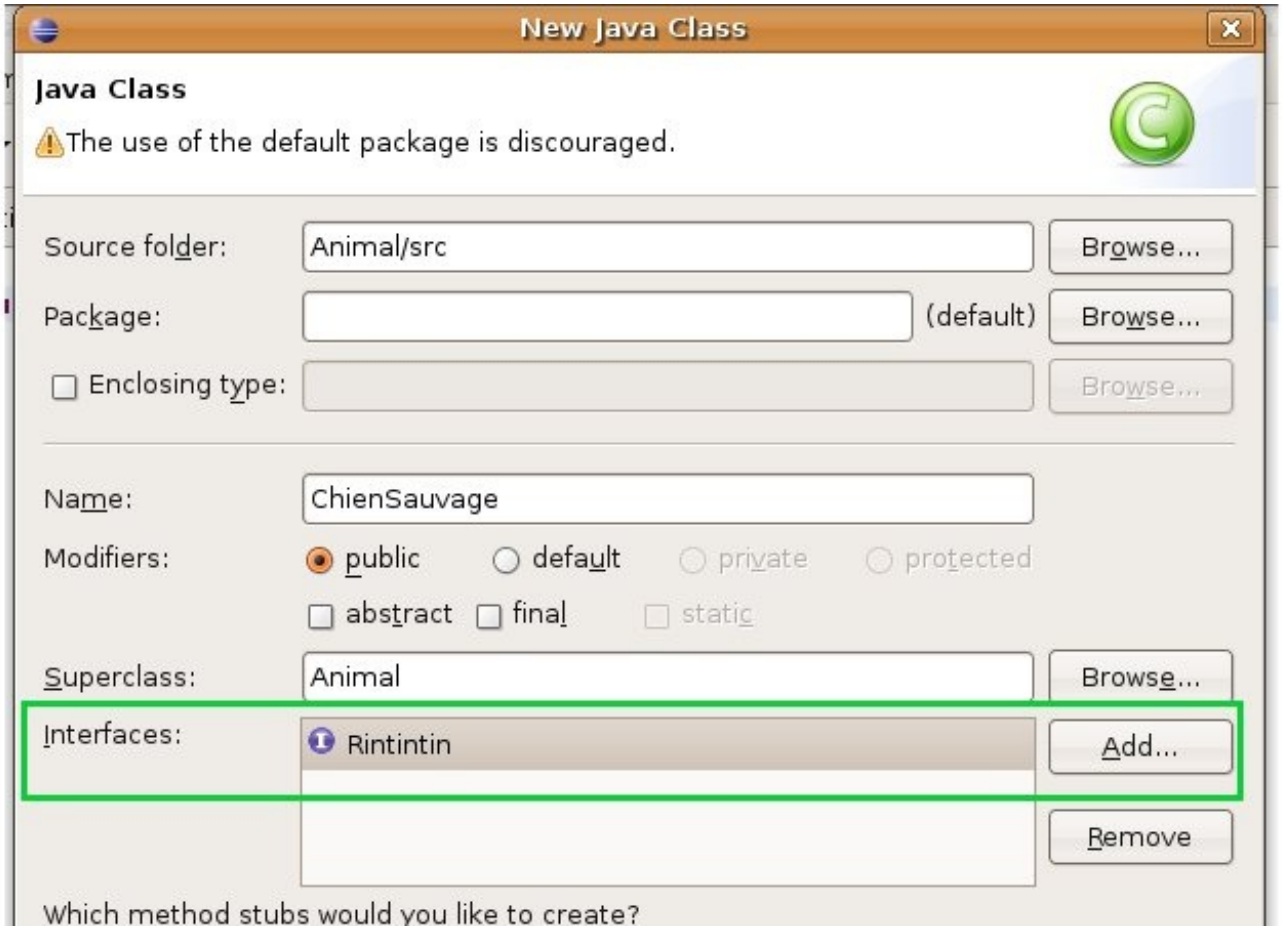
On crée une nouvelle classe héritant d'**`Animal`** (vous savez faire) et avant de valider, regardez juste en dessous :



Cliquez à présent sur Add, vous arrivez sur une fenêtre où il vous est demandé quelle interface vous voulez implémenter. Commencez à taper **`Rintintin`** et, grâce à l'autocomplétion, Eclipse vous fait une sélection des interfaces disponibles dont **`Rintintin`**. Voyez plutôt :



Il ne vous reste plus qu'à cliquer sur OK, et voyez le résulat :



Votre classe implémente l'interface **`Rintintin`**.
Terminez l'opération : vous avez une classe qui hérite d'**`Animal`** ET qui implémente **`Rintintin`**. Celle-ci redéfinit toutes les méthodes abstraites de la classe abstraite ainsi que toutes les méthodes de l'interface.

On dit merci qui ?
Bon : je crois qu'un topo est indispensable.
Retour en haut

Ce qu'il faut retenir

Alors, ici, la nouveauté vient de l'interface... Mais récapitulons un peu ce que nous avons vu.

- Une interface est une classe **100 % abstraite**.
- Toutes les méthodes d'une interface n'ont pas de corps.
- Une interface sert à définir un **super type et à utiliser le polymorphisme**.
- Une interface s'implémente dans une classe en utilisant le mot clé `implements`.
- Vous pouvez implémenter autant d'interfaces que vous voulez dans vos classes.
- **Vous devez redéfinir toutes les méthodes de l'interface (ou les interfaces) dans votre classe**.

Allez, c'est l'heure du QCM !

Retour en haut

Prenez bien le temps de comprendre les deux chapitres que nous venons de voir.
Ceux-ci forment l'un des gros blocs du langage et vous verrez, lorsque nous aborderons les interfaces graphiques, que ces deux concepts sont omniprésents !

Pour le moment, nous allons continuer de voir les fondements de la programmation orientée objet, et poursuivons tout de suite par **les exceptions** !

Retour en haut

Les exceptions

Voici encore une notion très importante en programmation Java.

Exception : erreur se produisant dans un programme conduisant le plus souvent à l'arrêt de celui-ci.

Il vous est sûrement déjà arrivé d'avoir un gros message affiché en rouge dans la console d'eclipse : eh bien ceci a été généré par une exception... qui n'a pas été **capturée**. La gestion des exceptions s'appelle aussi la capture d'exception !

Le principe consiste à repérer un morceau de code qui pourrait générer une exception (une division par zéro, par exemple), de capturer l'exception correspondante, et enfin de gérer celle-ci, c'est-à-dire d'afficher un message personnalisé et de continuer le traitement.

Bon : vous voyez maintenant ce que nous allons aborder durant ce chapitre... Donc, allons-y !

Premier exemple d'exception et le bloc try{....} catch{...}

Pour vous faire comprendre le principe des exceptions, je dois tout d'abord vous dire que Java contient une classe nommée **Exception**, où sont répertoriés différents cas d'erreurs. La division par zéro dont je vous parlais tout à l'heure en fait partie !

Toutes les classes de Java possèdent des exceptions : par exemple, la classe **`java.io`**, qui gère les entrées - sorties, a, entre autres, l'exception **`IOException`**.

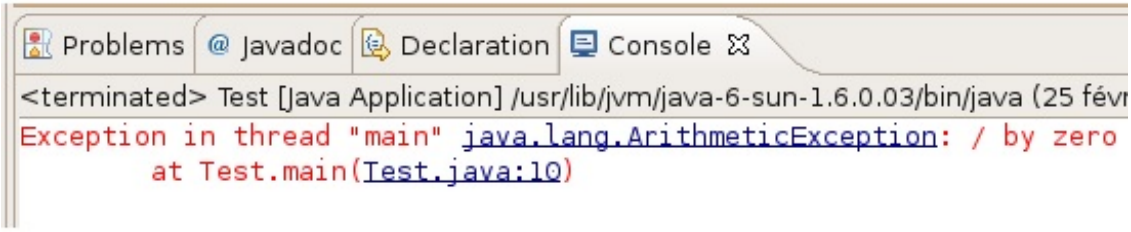
Je ne peux malheureusement pas vous en faire toute la liste, mais.... je peux vous donner une astuce pour savoir de laquelle il s'agit... Et ainsi pouvoir la capturer, et afficher un message personnalisé, sans que votre programme ne s'arrête.

Créez un nouveau projet avec seulement la classe **`main`**, et mettez-y le code suivant :

Code : Java

```
int j = 20, i = 0;
System.out.println(j/i);
System.out.println("coucou toi !");
```

Vous devriez avoir un zoli message d'erreur Java (en rouge) comme celui-ci :



Mais surtout, vous devez vous rendre compte que lorsque l'exception a été levée, **le programme s'est arrêté** !
Dans ce message, le nom de l'exception qui a été déclenchée est `ArithmeticException`. Nous savons donc maintenant qu'une division par zéro est une `ArithmeticException`. Nous allons pouvoir la capturer, et réaliser un traitement en conséquence.

Une exception se capture grâce à un bloc **`try`** {*ici, le code susceptible de générer une exception*} ; le traitement associé, lorsque le bloc `try{...}` capture une exception, se trouve dans un bloc **`catch`** {*ici, ce que le programme doit faire si une exception est capturée*}.

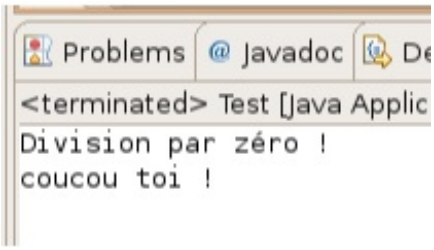
Ce que je vous propose maintenant, c'est de capturer l'exception de notre division par zéro, et d'afficher un message personnalisé. Pour ce faire, tapez le code suivant dans votre **`main`** :

Code : Java

```
public class Test {

	/**
* @param args
*/
	public static void main(String[] args) {

		int j = 20, i = 0;
		try {
			System.out.println(j/i);
		} catch (ArithmeticException e) {
			// TODO Auto-generated catch block
			System.out.println("Division par zéro !");
		}
		System.out.println("coucou toi !");
```

```
            }

        }
```

Si vous exécutez ce code, vous devez avoir ceci sous les yeux :



Je tiens tout d'abord à vous féliciter : vous venez de capturer votre première exception en Java !

Voyons un peu ce qui se passe :

- nous initialisons deux variables de type *int*, l'une à 0, et l'autre à un chiffre quelconque.
- Ensuite, nous isolons le code susceptible de générer une exception, ici : **`System.out.println(j/i);`**.
- Lorsque le programme atteint cette ligne, une exception de type **`ArithmeticException`** est levée.
- Notre bloc `catch` contient justement un objet de type **`ArithmeticException`** en paramètre. Nous l'avons appelé **e**.
- L'exception étant capturée, l'instruction du bloc `catch` s'exécute !
- Notre message d'erreur personnalisé s'affiche alors à l'écran.

Lorsque nous capturons une exception, nous pouvons aussi dire que nous levons une exception...

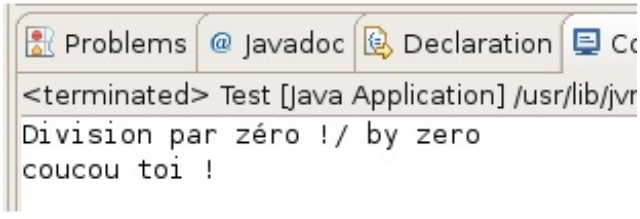
D'accord, mais le paramètre de notre bloc `catch`, il sert à quoi, au juste ?

Il sert à savoir quel type d'exception doit être capturé. Et l'objet -*ici, **e***- peut servir à agrémenter notre message, grâce à l'appel de la méthode **`getMessage()`**.
Faites à nouveau ce même test, en remplaçant l'instruction du `catch` par celle-ci :

Code : Java

```
System.out.println("Division par zéro !" + e.getMessage());
```

Vous devez maintenant avoir ceci :



Voilà ce que vous renvoie la fonction **`getMessage()`**.
Je vous disais aussi que le principe de capture d'exception permettait de ne pas arrêter l'exécution du programme. Et vous avez

pu le constater par vous-mêmes !
Lorsque nous capturons une exception, le code présent dans le bloc `catch(){...}` est exécuté mais le programme poursuit son cours !
Mais ce que vous ignorez sûrement, c'est que nous pouvons créer et intercepter nos propres exceptions !

Si nous passions tout de suite à la prochaine partie de ce chapitre ?
Retour en haut

Les exceptions personnalisées

À partir de maintenant, nous allons nous servir à nouveau de notre projet Ville (celui que vous avez utilisé dans les premiers chapitres...).
Nous allons perfectionner un peu la gestion de nos objets **Ville** et **Capitale**... Comment ? Eh bien je vois bien une exception qui pourrait être créée... Et je suis sûr que certains petits malins se sont déjà amusés à créer des villes ou des capitales avec un nombres d'habitants négatif....

Je vous propose simplement de mettre en oeuvre une exception de notre cru, ceci afin de pouvoir interdire l'instanciation d'objet **Ville** ou **Capitale** ayant un nombre négatif d'habitants.

La procédure pour faire ce tour de force est un peu particulière :

*Sdz1.java | Ville.java | Capitale.java | NombreHabitantException.java

```
1
2  public class Sdz1 {
3
4      public static void main(String[] args)
5      {
6          Ville v = null;
7          try {
8              v = new Ville("Rennes", -12000, "France");
9          } catch (NombreHabitantException e) {
10             v = new Ville();
11         }
12
13         System.out.println(v.toString());
14     }
15
16 }
17
```

Problems | Javadoc | Declaration | Console

```
<terminated> sdz1 [Java Application] C:\Program Files\Java\jre1.6.0_03\bin\javaw.exe (25 févr. 08 11:26:38)
Instanciation avec une nombre d'habitant négatif
        => -12000
       Inconnu est une ville de Inconnu, elle comporte : 0 => elle est donc de catégorie : ?
```

1. Nous devons créer une classe héritée de la classe Exception : appelons-la **NombreHabitantException**. Par convention, les exceptions ont un nom se terminant par **Exception**.
2. Nous devons renvoyer l'exception levée à notre classe **NombreHabitantException**.
3. Ensuite, gérer celle-ci dans notre classe **NombreHabitantException**.

Comment faire tout ça ?

Eh bien je vais vous apprendre encore deux autres mots clés !
Non, ne grimacez pas... Je vous assure que si vous en retenez un, vous allez retenir l'autre...

Le premier mot clé

throws

Ce mot clé permet de dire à une instruction Java (condition, déclaration de variable...) ou à une classe entière qu'une exception potentielle sera gérée par une classe -*souvent une classe personnalisée*- mais ce peut être la classe **Exception** elle-même. Ce mot clé est suivi du nom de la classe qui va gérer l'exception. Ceci a pour but de définir le type d'exception qui risque d'être générée par l'instruction, ou la classe qui précède le mot clé **throws**.

Le deuxième mot clé

throw

Celui-ci permet d'instancier un objet dans la classe suivant l'instruction **throws**. Cette instruction est suivie du mot clé new ainsi que d'un objet cité avec **throws**. En fait, il lance une exception, tout simplement.

Faites surtout bien attention à ne pas confondre ces deux mots clé.

Pour pouvoir mettre en pratique ce système, nous devons commencer par créer une classe qui va gérer nos exceptions. Celle-ci, je vous le rappelle, doit être héritée d'**Exception**. Pour commencer, inutile de créer un constructeur, ce qui nous donnerait une classe Erreur, héritée de Exception, vide.
Comme ceci :

Code : Java

```
class NombreHabitantException extends Exception{

   public NombreHabitantException(){
                System.out.println("Vous essayez d'instancier une
classe Ville avec un nombre d'habitants négatif !");
        }

}
```

Reprenez votre projet avec vos classes **Ville**, **Capitale** et créez maintenant une classe **NombreHabitantException**, comme je viens de le faire !

Maintenant, c'est dans le constructeur de nos objets que nous allons mettre une condition qui, si elle est remplie, lève une exception de type NombreHabitantException.

En gros, nous devons dire à notre constructeur de ville : "*Si l'utilisateur crée une instance ville avec un nombre d'habitants négatif, créer un objet de type* NombreHabitantException *(hérité d'*Exception).

Le constructeur d'initialisation de ville doit ressembler à ce qui suit, maintenant.

Code : Java

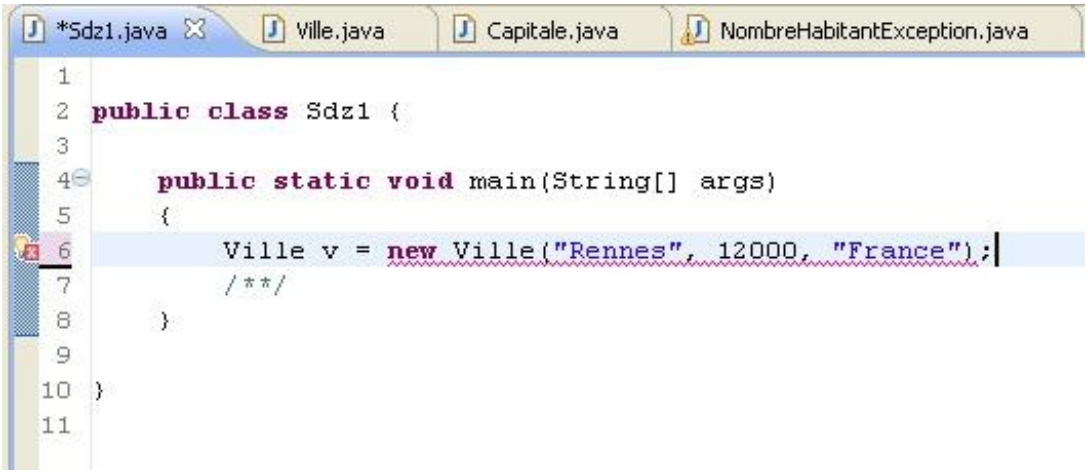
```
public Ville(String pNom, int pNbre, String pPays) throws
NombreHabitantException
  {
          if(pNbre < 0)
                  throw new NombreHabitantException();
          else
          {
                  nbreInstance++;
                  nbreInstanceBis++;

                  nomVille = pNom;
                  nomPays = pPays;
                  nbreHabitant = pNbre;
                  this.setCategorie();
          }
  }
```

throws NombreHabitantException nous indique que si une erreur est capturée, celle-ci sera traitée en tant qu'objet de la classe NombreHabitantException ! Ce qui, au final, nous renseigne sur le type de l'erreur en question.

throw new NombreHabitantException(); instancie la classe **NombreHabitantException** si la condition **if(nbre < 0)** est remplie.

Maintenant que vous avez fait cette petite modification, retournez dans votre classe **main**, effacez son contenu, puis créez un objet ville de votre choix !

Et à présent, vous devez voir une erreur persistante ; c'est tout à fait normal, ce qui doit vous donner ceci :



Ceci signifie qu'à partir de maintenant, dû aux changements dans le constructeur, vous devrez gérer les exceptions possibles sur cette instruction. Avec un bloc **try{} catch{}**.

On dit aussi que votre constructeur est devenu une **méthode à risque**, et vous avez laissé le soin au développeur de gérer l'exception potentielle !

Donc, pour que l'erreur disparaisse, il nous faut entourer notre instanciation avec un bloc **try{...}catch{...}**.
Comme ceci :

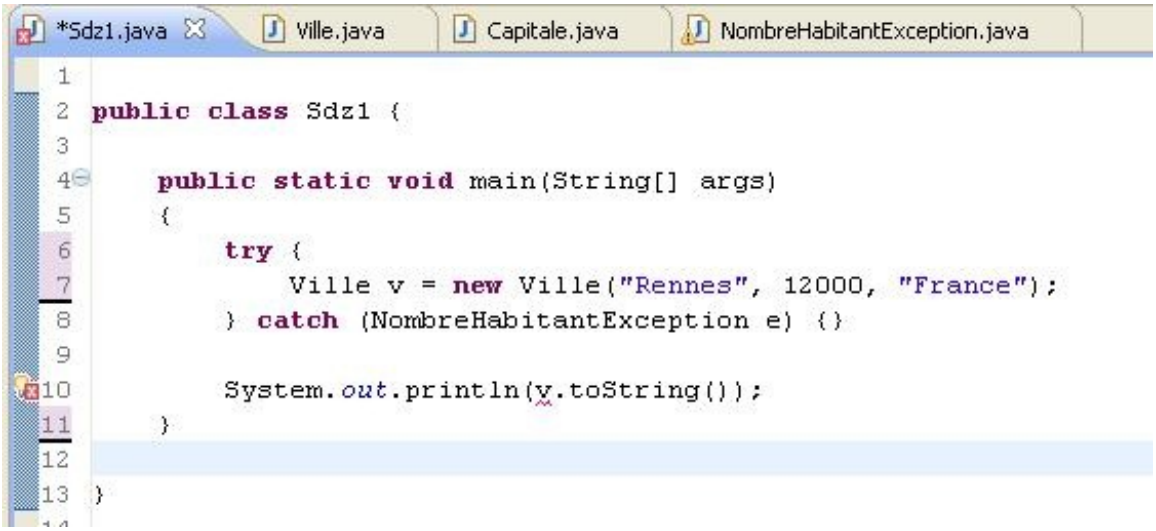
Vous pouvez constater que l'erreur a disparu et que notre code compile et s'exécute correctement.
Par contre, il faut que vous soyez préparés à une chose. Le code que j'ai utilisé ci-dessus fonctionne très bien, mais ce code :

Code : Java

```
public class Sdz1 {

        public static void main(String[] args)
        {
                try {
                        Ville v = new Ville("Rennes", 12000,
```

```
        "France");
                        } catch (NombreHabitantException e) {}

                        System.out.println(v.toString());
                }

        }
```

ne fonctionnera pas et pour appuyer mes dires, voici le témoignage de quelqu'un d'intègre :



Vous pouvez constater qu'Eclipse n'aime pas du tout notre code !

Pourquoi cela ?

Tout simplement car la déclaration de votre objet **Ville** est faite dans un sous-bloc d'instructions, celui du bloc **try{}**. Et rappelez-vous :

Une variable déclarée dans un bloc d'instructions n'existe que dans ce bloc d'instructions !

Donc, ici, notre objet **v**, n'existe pas en dehors de l'instruction **try{}**. Pour **pallier ce problème**, il nous suffit de **déclarer notre objet en dehors du bloc try{} et de l'instancier à l'intérieur** !

Nous pouvons faire ceci :

Code : Java

```
        public class Sdz1 {

                public static void main(String[] args)
                {
                        Ville v = null;
                        try {
                                v = new Ville("Rennes", 12000, "France");
                        } catch (NombreHabitantException e) {}

                        System.out.println(v.toString());
                }

        }
```

Et ce code nous donne :

Code : Console

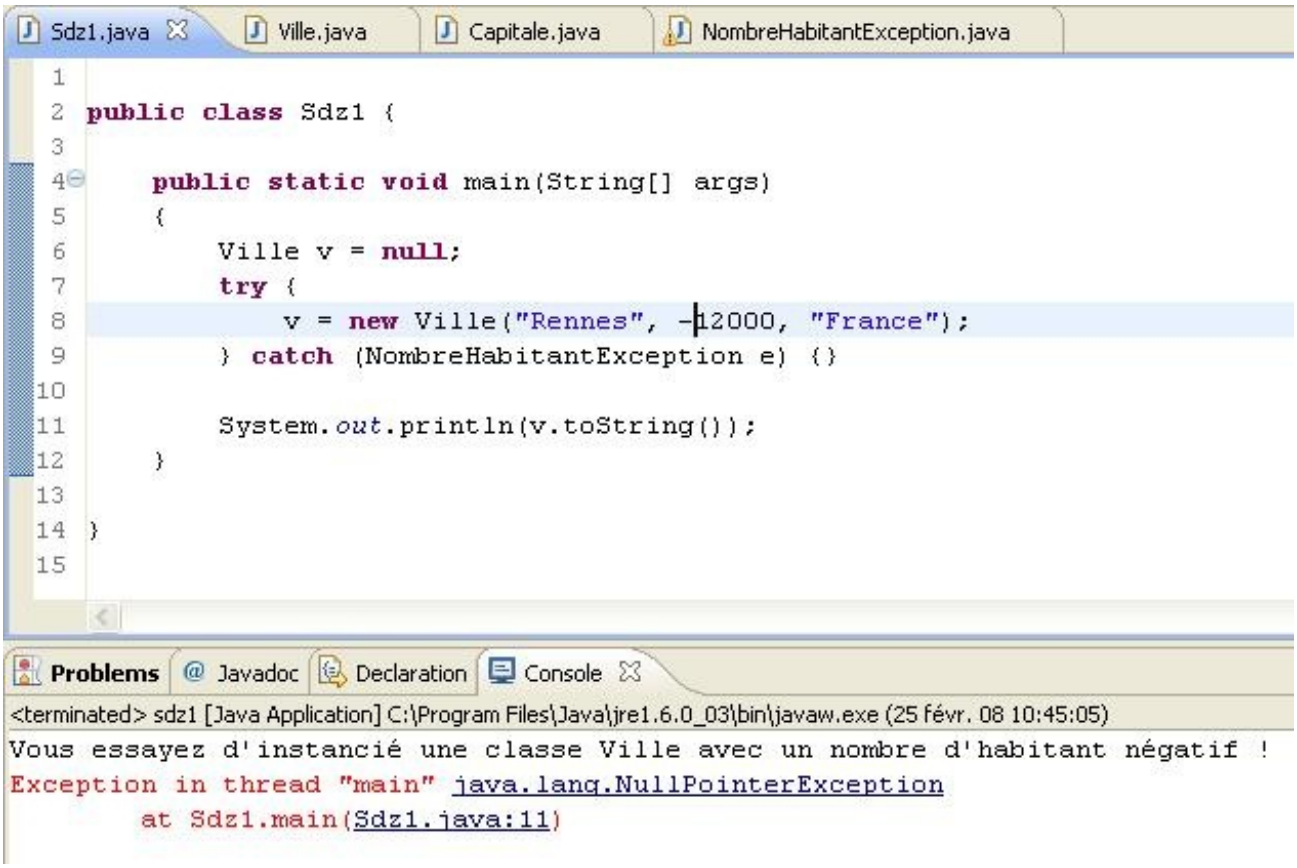
```
Rennes est une ville de France, elle comporte : 12000 => elle est donc de catégorie
```

Mais si nous déclarons une Ville avec un nombre d'habitants négatif pour tester notre exception ?
Avec ce code, par exemple :

Code : Java

```
public class Sdz1 {

        public static void main(String[] args)
        {
                Ville v = null;
                try {
                        v = new Ville("Rennes", -12000, "France");
                } catch (NombreHabitantException e) {}

                System.out.println(v.toString());
        }

}
```

Voici ce que nous obtenons :



Voyons ce qu'il s'est passé.

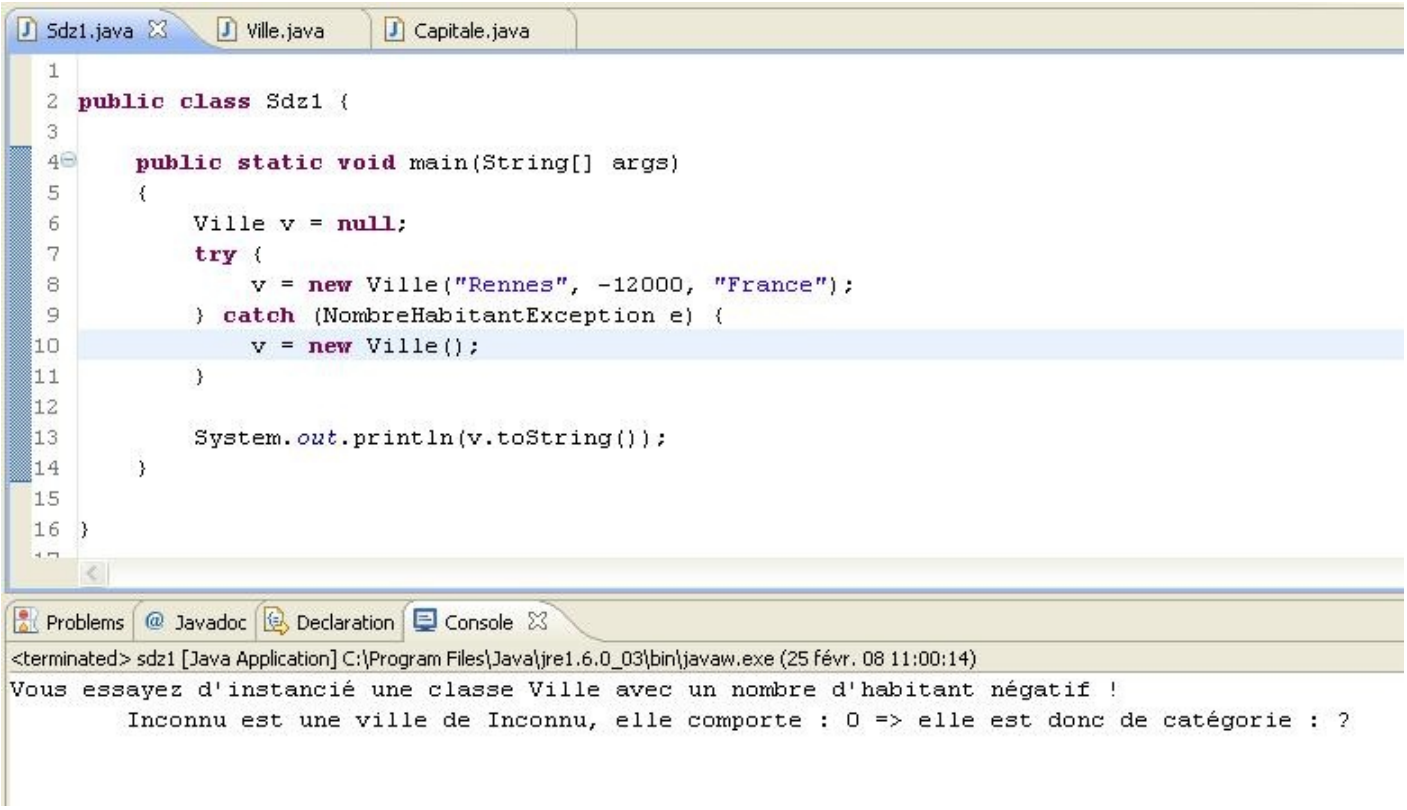
- Nous avons bien déclaré notre objet en dehors du bloc d'instructions.

- Au moment d'instancier celui-ci, une exception est levée ! L'instanciation échoue lamentablement !
- La clause **catch{}** est exécutée et notre objet **NombreHabitantException** est instancié. Écriture du message.
- Et lorsque nous arrivons sur l'instruction "**System.out.println(v.toString());**", notre objet est **null** !
- Une **NullPointerException** est levée !

Ce qui signifie que si notre instanciation a échoué dans notre bloc **try{}**, le programme plantera !

Comment empêcher cela, alors ?

Vous allez voir, c'est très simple. Il suffit d'instancier un objet Ville par défaut dans notre bloc **catch{}**. Grâce à cela, si notre instanciation avec valeur échoue, on fait une instanciation par défaut qui, elle, n'est pas une méthode à risque !
Voyez plutôt :



Vous pouvez voir que l'exception est bien levée et que notre objet est instancié mais, surtout, que notre programme ne plante plus !

Maintenant que nous avons vu la création d'exception, il serait de bon ton que nous puissions avoir de plus amples renseignements concernant celle-ci.
Par exemple, il serait peut-être bon de réafficher le nombre d'habitants que l'objet a reçu...
Pour faire ceci, nous n'avons qu'à créer un deuxième constructeur dans notre classe **NombreHabitantException**, qui prend un nombre d'habitants en paramètre. Un peu comme ça :

Code : Java

```
public NombreHabitantException(int nbre)
{
    System.out.println("Instanciation avec un nombre d'habitants
négatif");
    System.out.println("\t => " + nbre);
}
```

Il vous suffit maintenant de définir cette construction de notre objet hérité d'**Exception** dans votre classe **Ville**. Comme ça :

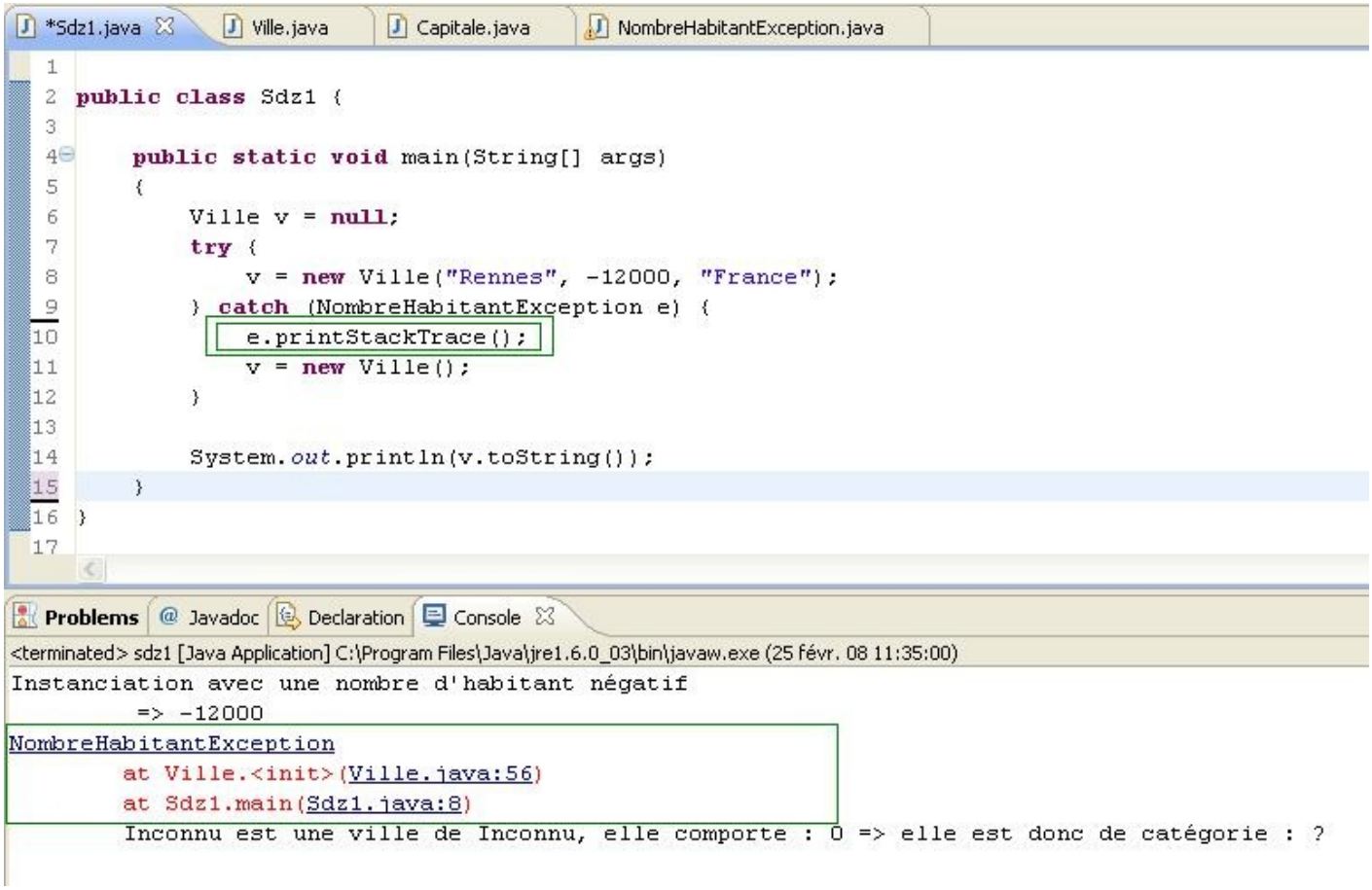
Code : Java

```
public Ville(String pNom, int pNbre, String pPays) throws
NombreHabitantException
  {
          if(pNbre < 0)
                  throw new NombreHabitantException(pNbre); // on
appelle le nouveau constructeur
          else
          {
                  nbreInstance++;
                  nbreInstanceBis++;

                  nomVille = pNom;
                  nomPays = pPays;
                  nbreHabitant = pNbre;
                  this.setCategorie();
          }

  }
```

Et si vous exécutez le même code que précédemment, vous obtiendrez ceci :

C'est pas mal, avouez-le !

Mais vous devez savoir que l'objet passé en paramètre de la clause **catch** a des méthodes héritées de la classe **Exception**. Regardez :



Vous pouvez les utiliser si vous le voulez et surtout, si vous en avez l'utilité...
Nous utiliserons certaines de ces méthodes dans les prochains chapitres...

Ici, la méthode **printStackTrace()** permet de voir ou se situe l'exception dans notre code ! Elle vous informe sur le nom de la classe levant l'exception et le numéro de ligne ou se trouve le code l'ayant levée.

Je vais vous faire peur : ici, nous avons capturé une exception mais nous pouvons en capturer plusieurs...

Retour en haut

La gestion de plusieurs exceptions

Bien entendu, ceci est valable pour toutes sortes d'exceptions, qu'elles soient personnalisées, ou faisant partie de Java !
Disons que nous voulons lever une exception, si le nom de la ville fait moins de 3 caractères.

Nous allons répéter les premières étapes vues précédemment, c'est-à-dire créer une classe **NomVilleException**:

Code : Java

```
public class NomVilleException extends Exception {

        public NomVilleException(String message){
                super(message);
        }

}
```

Vous avez remarqué que nous avons utilisé **super**. Avec cette redéfinition, nous pourrons afficher notre message d'erreur en utilisant la méthode **getMessage()**.
Vous allez voir.

Maintenant ajoutez une condition dans notre constructeur **Ville** :

Code : Java

```
public Ville(String pNom, int pNbre, String pPays) throws
NombreHabitantException, NomVilleException
  {
          if(pNbre < 0)
                  throw new NombreHabitantException(pNbre);

          if(pNom.length() < 3)
                  throw new NomVilleException("le nom de la ville
est inférieur à 3 caractères ! nom = " + pNom);
          else
          {
                  nbreInstance++;
                  nbreInstanceBis++;

                  nomVille = pNom;
                  nomPays = pPays;
                  nbreHabitant = pNbre;
                  this.setCategorie();
          }

  }
```

Vous remarquez que les différentes erreurs dans l'instruction `throws` sont séparées par une virgule !
Maintenant, nous sommes parés pour la capture de deux exceptions personnalisées. Regardez comment on gère deux exceptions sur une instruction :

Code : Java

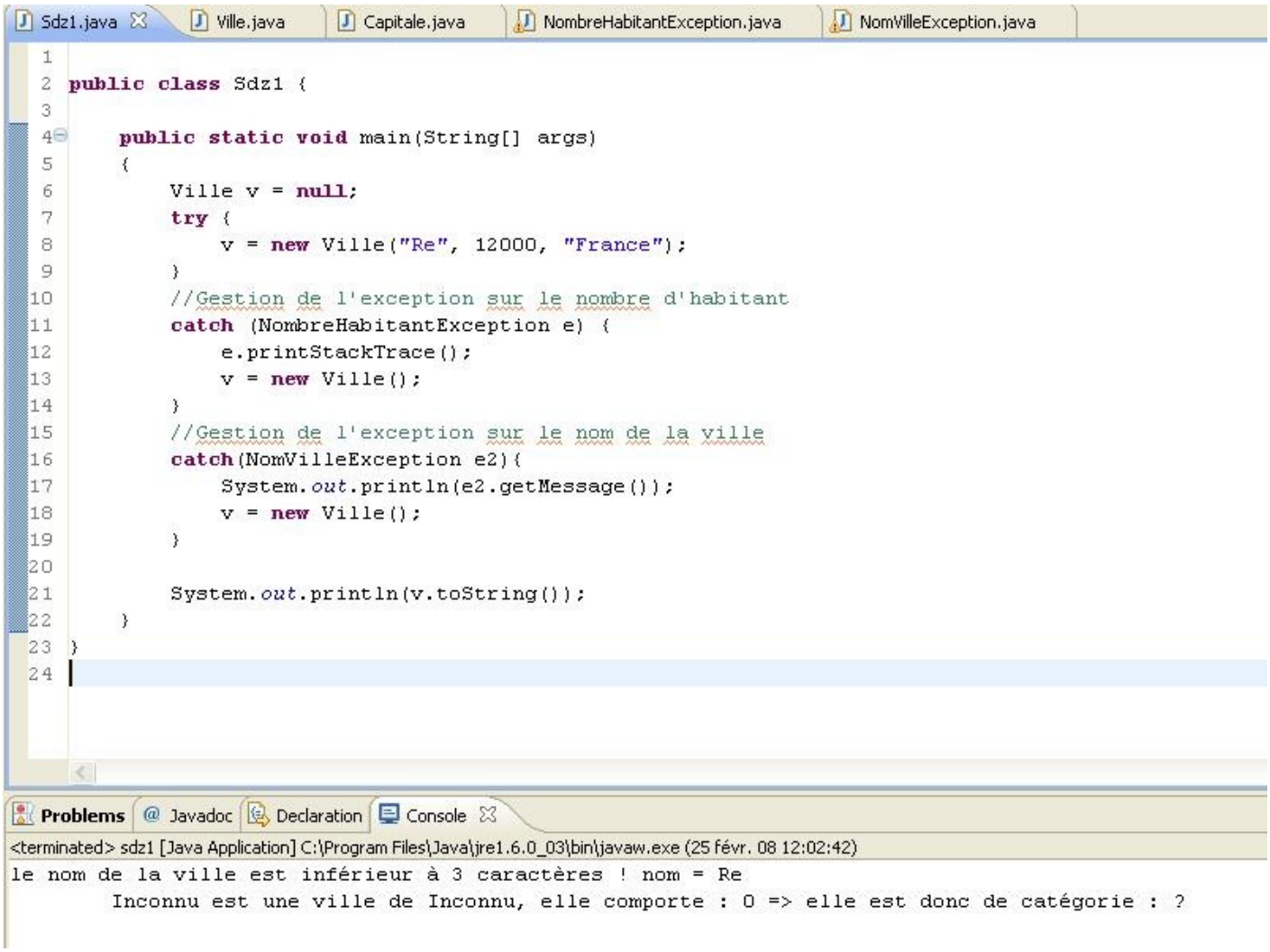
```
Ville v = null;
                try {
                        v = new Ville("Re", 12000, "France");
                }
                //Gestion de l'exception sur le nombre d'habitants
                catch (NombreHabitantException e) {
                        e.printStackTrace();
                        v = new Ville();
                }
```

```
                        //Gestion de l'exception sur le nom de la ville
                        catch(NomVilleException e2){
                                System.out.println(e2.getMessage());
                                v = new Ville();
                        }

                        System.out.println(v.toString());
```

Vous pouvez voir comment utiliser la méthode **`getMessage()`** à présent. C'est tout bête, avouez-le !

Vous pouvez constater qu'un deuxième bloc **`catch{}`** s'est glissé... Eh bien c'est comme ceci que nous gérerons plusieurs exceptions !
Vous avez aussi remarqué que j'ai aussi changé le code afin que l'exception sur le nom soit levée et non plus l'exception sur le nombre d'habitants...

Vous aurez ceci :



Si vous mettez un nom de ville de moins de 3 caractères, et un nombre d'habitants négatif, c'est l'exception du nombre d'habitants qui sera levée en premier ! Et pour cause... il s'agit de notre première condition dans notre constructeur...

Lorsque plusieurs exceptions sont gérées par une portion de code, pensez bien à mettre les blocs `catch` du plus pertinent au moins pertinent. En fait, il s'agit des erreurs capturées à mettre par ordre de pertinence. Dans notre premier exemple d'exception, sur la division par zéro, si nous avions mis un bloc **`catch(Exception ex){}`** avant le bloc

`catch(ArithmeticException e){}`, une erreur se serait produite à la compilation, car `Exception` est plus générique que `ArithmeticException`.

Il y a une instruction dont je ne vous ai pas encore parlé... Il s'agit de la clause `finally`. Celle-ci est une clause se positionnant après les clauses `catch`.
En fait, ce qui se trouve dans cette clause sera **TOUJOURS** exécuté. **Qu'une exception soit levée ou non**.

Exemple :

Code : Java

```
public class Sdz1 {

	public static void main(String[] args)
	{
		Ville v = null;
		try {
			v = new Ville("Re", 100, "France");
		}
		//Gestion de l'exception sur le nombre d'habitants
		catch (NombreHabitantException e) {
			e.printStackTrace();
			v = new Ville();
		}
		//Gestion de l'exception sur le nom de la ville
		catch(NomVilleException e2){
			System.out.println(e2.getMessage());
			v = new Ville();
		}

		//La fameuse clause finally
		finally{
			System.out.println("\n----------------------
----------------------");
			System.out.println("Voici le code qui est
toujours exécuté ! ! ! ");
			System.out.println("----------------------
----------------------\n");
		}
		System.out.println(v.toString());
	}
}
```

Quoi qu'il se passe dans la classe `try` ou `catch`, les instructions de la clause `finally` seront TOUJOURS exécutées ! Et vous pouvez faire autant de tests que vous le voulez.

Euh... d'accord. Mais à quoi ça peut bien nous servir ? On ne peut pas mettre l'instanciation de notre objet ici !

Très juste !
Par contre, vous allez apprendre dans les chapitres suivants à ouvrir des flux de données. Ce genre de code regorge d'exceptions en tout genre et vous serez ravis de pouvoir fermer votre flux, quoiqu'il arrive !

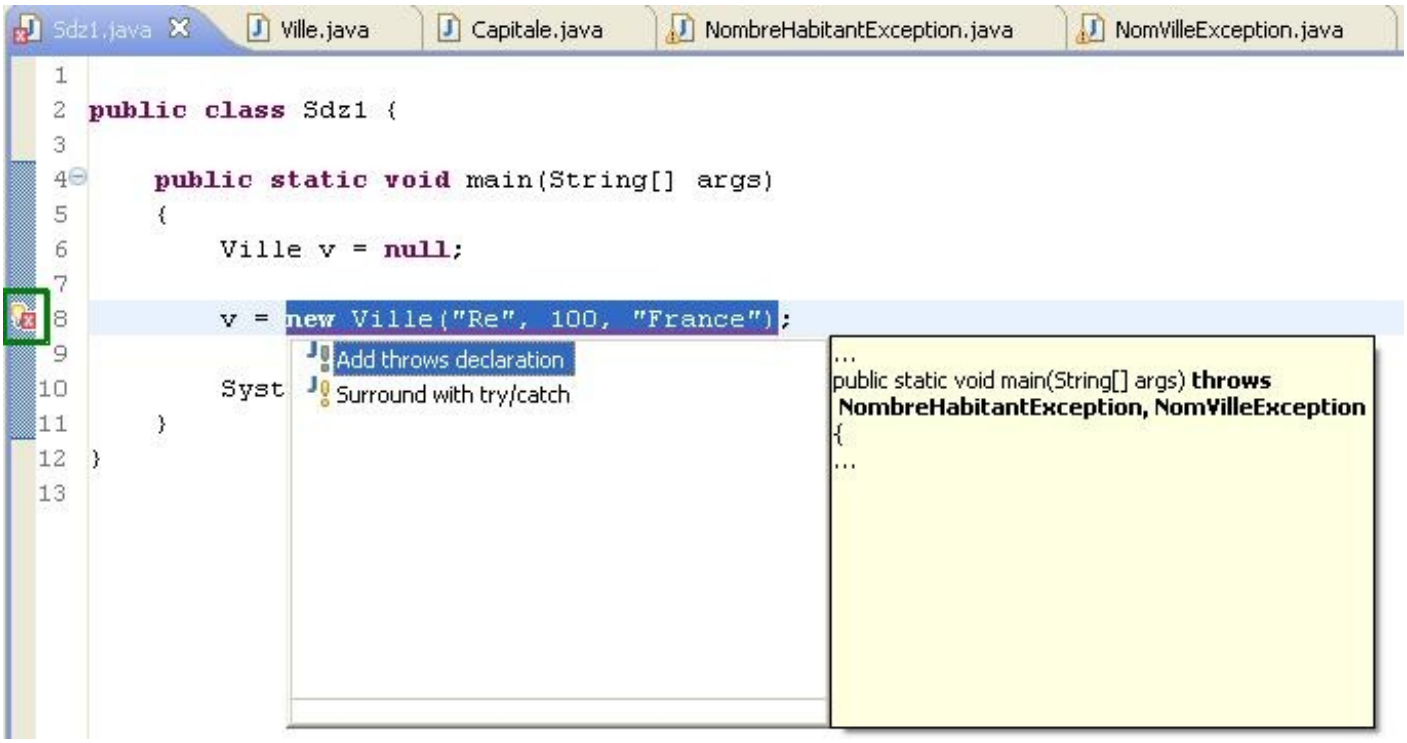
Bon : je crois qu'un récapitulatif s'impose mais avant ceci, voici notre rubrique "**Astuce d'Eclipse**".
Retour en haut

Astuce d'Eclipse

Il y a plusieurs manières de procéder, mais toutes font la même chose.
L'astuce ici réside dans le fait de générer les blocs `try{} catch{}` automatiquement. Bien sûr, il faut que les clauses de déclenchement soient définies au préalable !

Si vous reprenez le code de votre méthode **`main`**, si vous effacez le contenu et ajoutez une instanciation de l'objet `Ville` sans les clauses `try{} catch{}`, vous avez l'erreur persistante dont je vous parlais au début du chapitre.

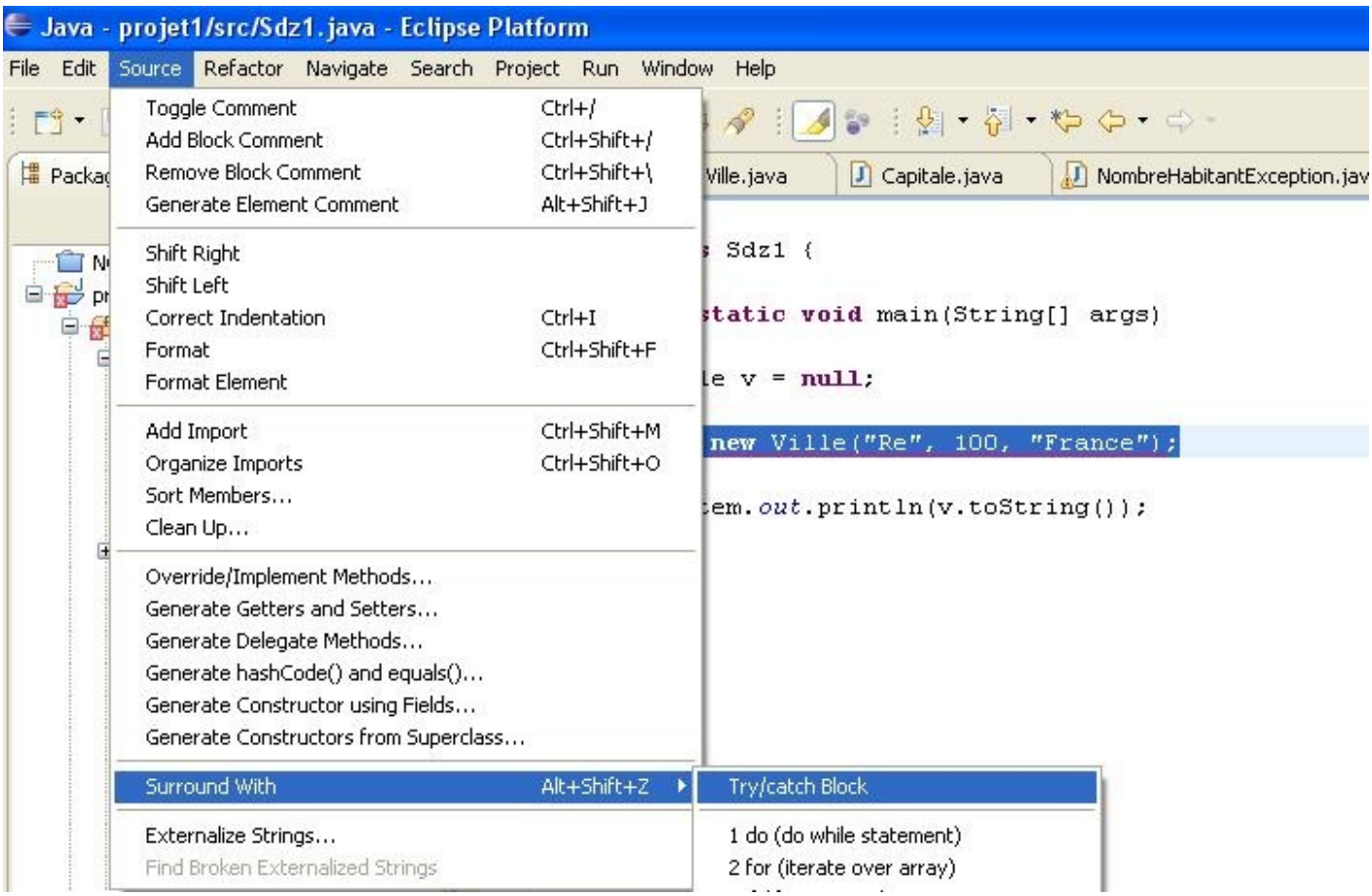
Si vous cliquez sur la croix rouge, située à gauche de votre zone de saisie, vous obtenez ceci :



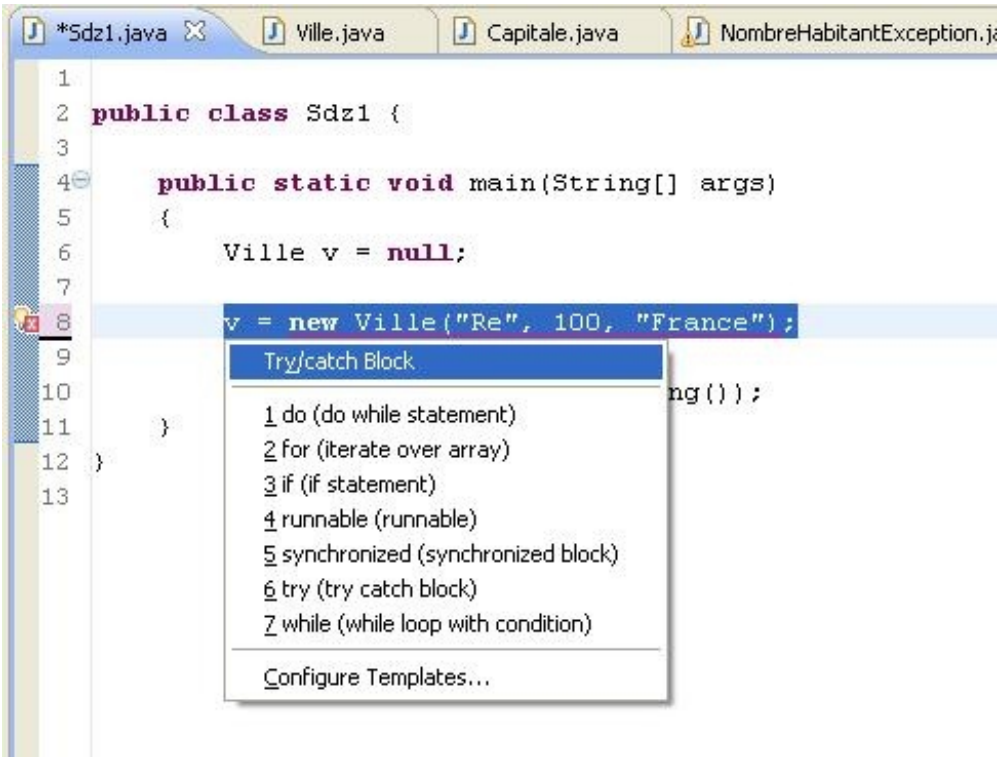
Choisissez l'option Surround with try/catch et vous avez votre code, tout beau tout propre !

La deuxième méthode consiste à sélectionner votre (ou vos) ligne(s) de code à risque et de faire : Source / Surround with / try/catch block ou d'utiliser le raccourci clavier ***Shift*** + ***Alt*** + ***Z*** :

Voici l'image en utilisant le menu :



Voici l'image en utilisant le raccourci clavier :



Voilà, maintenant, rendez-vous au topo habituel !

Retour en haut

Ce qu'il faut retenir

- La super classe qui gère les exceptions s'appelle : **`Exception`**.
- Lorsqu'un événement que la JVM ne sait pas gérer apparaît, une exception est levée (ex : division par zéro) !
- Vous pouvez créer une classe d'exception personnalisée en créant une classe héritant de la classe **`Exception`**.
- L'instruction qui permet de capturer des exceptions est le bloc **`try{} catch{}`**.
- Si une exception est levée dans le bloc **`try`**, les instructions figurant dans le bloc **`catch`** seront exécutées si celui-ci capture l'exception levée !
- Vous pouvez ajouter autant de blocs **`catch`** que vous le voulez à la suite d'un bloc **`try`**. Mais respectez l'ordre de pertinence. **Du plus pertinent au moins pertinent** !
- Dans une classe objet, vous pouvez prévenir la JVM qu'une méthode est dite "***à risque***", ceci grâce au mot clé **`throws`**.
- Vous pouvez définir plusieurs risques d'exceptions sur une même méthode. Il suffit de séparer les déclarations par une virgule !
- Dans cette méthode, vous pouvez définir les conditions d'instanciation d'une exception et lancer cette dernière grâce au mot clé **`throw`** suivie de l'instanciation.
- Une instanciation lancée par le biais de l'instruction **`throw`** **DOIT ÊTRE DÉCLARÉE** avec **`throws`** au préalable ! !

Retour en haut

Je me doute que vous avez dû avoir pas mal de sueurs froides à la lecture de ce chapitre...

Vous devez savoir aussi que vous pouvez imbriquer des blocs de captures d'exceptions comme ceci :

Code : Java

```
try{
   // Le code sensible pouvant lever une Exception1

   try{
       //Code sensible pouvant lever une Exception2
   }catch(Exception2 e2){}


}
catch(Exception1 e){
}
```

Je vous propose maintenant de voir **les collections** !

Faites tout de même une pause pour bien comprendre le fonctionnement des exceptions ; pour les avides de connaissances, rendez-vous au chapitre suivant !

Retour en haut

Les collections d'objets

Voici une partie qui va sans doute plaire...
Ici nous allons voir qu'il est possible de stocker des données autrement qu'avec des tableaux !
Et je dirais même plus, ces fameux objets collections sont dynamiques... en gros, ils n'ont pas de taille à pré-définir. On ne peut donc pas dépasser leur capacité !

Je ne passerai pas en revue tous les types et tous les objets `collection`... J'ajouterai aussi qu'il s'agit d'une présentation succinte des collections d'objets. Je préfère vous prévenir à l'avance !

Je sens que vous êtes impatients. Allons-y, alors !

L'objet LinkedList

Une liste chaînée est une liste dont chaque élément est relié au suivant par une référence à ce dernier, sa taille n'est pas fixe : on peut ajouter et enlever des éléments selon nos besoins.
Les `LinkedList` acceptent tout type d'objet.
Chaque élément contient une référence sur l'élément suivant sauf pour le dernier : son suivant est en fait **`null`**.
Cette classe se trouve dans le package **`java.util`**.

Voici un petit code pour appuyer mes dires :

Code : Java

```
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;

public class Test {

	public static void main(String[] args) {

		List l = new LinkedList();
		l.add(12);
		l.add("toto ! !");
		l.add(12.20f);


		for(int i = 0; i < l.size(); i++)
			System.out.println("Élément à l'index " + i
+ " = " + l.get(i));

	}
}
```

Vous pourrez constater que tous les éléments s'affichent !
Maintenant, vous devez savoir autre chose sur ce genre d'objet. Ceux-ci implémentent l'interface **`Iterator`**. Ceci signifie que nous pouvons utiliser cette interface pour lister notre **`LinkedList`**.
Dans le code suivant, j'ai rajouté le parcours avec un itérateur :

Code : Java

```
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;

public class Test {

	public static void main(String[] args) {

		List l = new LinkedList();
		l.add(12);
		l.add("toto ! !");
		l.add(12.20f);


		for(int i = 0; i < l.size(); i++)
			System.out.println("Élément à l'index " + i
+ " = " + l.get(i));


		System.out.println("\n \tParcours avec un itérateur
");
		System.out.println("-------------------------------
---");
		ListIterator li = l.listIterator();

		while(li.hasNext())
```

```
                System.out.println(li.next());
            }
        }
```

Vous pouvez constater que les deux manières de procéder sont analogues !
Cependant, je dois vous dire quelques choses sur les listes chaînées. Vu que les éléments ont une référence à leur élément suivant, ce type de listes peut être particulièrement lourd lorsqu'elles deviennent volumineuses !
À utiliser avec précaution. Si vous voulez en savoir plus, c'est par là.
Nous allons voir un autre objet de la même famille : les **`ArrayList`**.
Retour en haut

L'objet ArrayList

Voici un objet bien pratique.
Un **ArrayList** est donc un de ces objets qui n'ont pas de taille limite, et en plus, ils acceptent n'importe quel type de données ! **null** y compris !

Dans un **ArrayList**, nous pouvons mettre tout ce que nous voulons. Vous devez par contre importer la classe **ArrayList**. Elle se trouve dans le package **java.util**. Mais vous connaissez une bonne technique pour importer vos classes !

Pour preuve, voici un morceau de code qui le met en œuvre :

Code : Java

```
import java.util.ArrayList;

public class Test {

	public static void main(String[] args) {

		ArrayList al = new ArrayList();
		al.add(12);
		al.add("Une chaîne de caractères !");
		al.add(12.20f);
		al.add('d');

		for(int i = 0; i < al.size(); i++)
		{
			System.out.println("donnée à l'indice " + i
+ " = " + al.get(i));
		}
	}
}
```

Si vous exécutez ce code, vous aurez :

```
Problems  Javadoc  Declaration  Console
<terminated> Test [Java Application] C:\Program Files\Java\jre1.6.0_03\bin\javaw.exe (25 f
donnée à l'indice 0 = 12
donnée à l'indice 1 = Une chaîne de caractère !
donnée à l'indice 2 = 12.2
donnée à l'indice 3 = d
```

Je pense que vous voyez déjà les avantages des **ArrayList**.
Vous devez savoir aussi qu'il existe tout un panel de méthodes dans cet objet :

- **add()** : permet d'ajouter un élément.
- **get(int index)** : retourne l'élément à l'index demandé.
- **remove(int index)** : efface l'entrée à l'indice demandé.
- **isEmpty()** : renvoie "vrai" si l'objet est vide.
- **removeAll()** : efface tout le contenu de l'objet.
- **contains(Object element)** : retourne "vrai" si l'élément passé en paramètre est dans l'objet.
- ...

Si vous voulez en savoir plus, c'est par là.

Un autre objet pratique, les **Hashtable**.
Retour en haut

L'objet Hashtable

Table de hachage, si vous traduisez mot à mot...
Ce type d'objet rentre dans la catégorie des **Map**. Entendez par là qu'ils rangent leurs éléments avec un système "***clé-valeur***". La clé est unique, mais la valeur, elle, peut être associée à plusieurs clés !

On parcourt ces objets grâce aux clés qu'ils contiennent. Vu que notre mémoire peut être défaillante, il existe un moyen de parcours très simple. En utilisant la classe **Enumeration**. L'objet obtient tout le contenu de notre **Hashtable** et permet de le parcourir très simplement. Regardez, le code suivant insère les saisons avec des clés qui ne se suivent pas, et notre énumération récupère seulement les valeurs :

Code : Java

```
import java.util.Enumeration;
import java.util.Hashtable;

public class Test {

        public static void main(String[] args) {

                Hashtable ht = new Hashtable();
                ht.put(1, "printemps");
                ht.put(10, "été");
                ht.put(12, "automne");
                ht.put(45, "hiver");

                Enumeration e = ht.elements();

                while(e.hasMoreElements())
                        System.out.println(e.nextElement());

        }
}
```

Vous constaterez que le code marche très bien et qu'il est très simple d'utilisation.
Cet objet nous offre lui aussi tout un panel de méthodes utiles :

- **`isEmpty()`** : retourne "vrai" si l'objet est vide.
- **`contains(Object value)`** : retourne "vrai" si la valeur est présente. Identique à **`containsValue(Object value)`**.
- **`containsKey(Object key)`** : retourne "vrai" si la clé passée en paramètre est présente.
- **`put(Object key, Object value)`** : ajoute le couple `key/value` dans l'objet.
- **`elements()`** : retoune une énumération des éléments de l'objet.
- **`keys()`** : retourne la liste des clés sous forme d'énumération.
- ...

Vous devez savoir qu'il existe un autre objet de type **Map** : il s'agit du **HashMap** ; son utilisation ne diffère que très peu du **Hashtable**. Je n'en parlerai donc pas...

Si vous voulez en savoir plus sur les **Hashtable**, c'est par là.
Retour en haut

L'objet HashSet

Un `Set` est une collection qui n'accepte pas les doublons. Elle n'accepte qu'une seule fois la valeur **`null`**, car deux fois cette valeur est considérée comme un doublon.
On peut dire que cet objet n'a que des éléments différents en son sein !
Certains `Set` sont plus restrictifs que d'autres, n'acceptent pas `null` ou un certain type d'objet.
On peut parcourir ce type de collection avec un objet **`Iterator`** ou, cet objet peut retourner un tableau d'**`Object`**.
Voici un code simple :

Code : Java

```
import java.util.HashSet;
import java.util.Iterator;

public class Test {

        public static void main(String[] args) {

                HashSet hs = new HashSet();
                hs.add("toto");
                hs.add(12);
                hs.add('d');

                Iterator it = hs.iterator();
                while(it.hasNext())
                        System.out.println(it.next());

                System.out.println("\nParcours avec un tableau
d'objet");
                System.out.println("--------------------------------
---");

                Object[] obj = hs.toArray();
                for(Object o : obj)
                        System.out.println(o);

        }
}
```

Voici une liste des méthodes que l'on peut trouver dans cet objet :

- **`add()`** : ajoute un élément.
- **`contains(Object value)`** : retourne "vrai" si l'objet contient `value`.
- **`isEmpty()`** : retourne "vrai" si l'objet est vide.
- **`iterator()`** : renvoie un objet de type `Iterator`.
- **`remove(Object o)`** : retire l'objet o de la collection.
- **`toArray()`** : retourne un tableau d'`Object`.
- ...

Pour en savoir plus sur les HashSet, c'est par là.

Voilà : nous avons vu quelque chose d'assez intéressant. Je ne pense pas qu'un QCM soit de mise pour cette partie...
Profitez-en !
Allez ! Zou...
Retour en haut
Voici encore un chapitre important !
Surtout prenez bien le temps de faire des tests, de voir comment se comporte chaque objet...

Le principal problème qui va se poser maintenant, c'est :

Quelle collection utiliser ?

Pour ceci, je ne peux pas vous aider... cela dépendra du type de besoin que vous aurez...
Personnellement, je me sers le plus souvent d'`ArrayList` et de `Hashtable`.

Dès que vous vous sentez prêts, en avant pour la **généricité en Java**.
Retour en haut

La généricité en Java

Pour ce concept, ajouté au JDK depuis sa version 1.5, nous allons surtout travailler avec des exemples tout au long de ce chapitre.
Le principe de la généricité est de faire des classes qui n'acceptent qu'un certain type d'objet ou de donnée, mais de façon dynamique ! :D.

Avec ce que nous avons vu au chapitre précédent, vous avez sûrement poussé un soupir de soulagement lorsque vous avez vu que ces objets acceptent tous types de données !
Par contre, un problème de taille se pose : lorsque vous voudrez travailler avec ces données, vous allez devoir faire un `cast` ! Et peut-être même un `cast` de `cast`, voire même un `cast` de `cast` de `cast`...

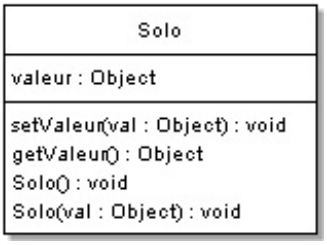
C'est là que se situe le problème... Mais comme je vous le disais, depuis la version 1.5 du JDK, la généricité est là pour vous aider !

Notion de base

Bon, pour vous montrer la puissance de la généricité, nous allons tout de suite voir un exemple de classe qui ne l'utilise pas !
Ne vous en faites pas... Ayez confiance en moi...

Il y a un exemple très simple, que vous pourrez retrouver aisément sur le net, car il s'agit d'un des cas les plus simples pour expliquer les bases de la généricité. Nous allons coder une classe **`Solo`**. Celle-ci va travailler avec des références de type **`String`**.
Voici le diagramme de classe :

Vous pouvez voir que le code de cette classe est très rudimentaire ! On affecte une valeur, on peut la mettre à jour et la récupérer...
Maintenant, si je vous demande de me faire une classe qui permette de travailler avec n'importe quel type de données, j'ai une vague idée de ce que vous allez faire... Ce ne serait pas un truc comme ça :



J'en étais sûr... :D. Créez-la et créez-vous aussi une classe avec une méthode **`main`** !
Mais si vous voulez utiliser les données de l'objet **`Solo`**, vous allez devoir faire un **`cast`**.
Testez ce code dans votre `main` :

Code : Java

```
public class Test {

        public static void main(String[] args) {

                Solo val = new Solo(12);
                int nbre = val.getValeur();
        }

}
```

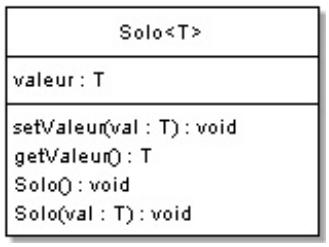
Vous constatez que vous tentez vainement de mettre un objet de type **`Object`** dans un objet de type **`Integer`**. Ceci est interdit ! !
La classe **`Object`** est plus globale que la classe **`Integer`**, vous ne pouvez donc pas faire cette manipulation, sauf si vous "castez" votre objet en **`Integer`**, comme ceci :

Code : Java

```
public class Test {

        public static void main(String[] args) {
```

```
                Solo val = new Solo(12);
                int nbre = (Integer)val.getValeur();
        }

}
```

Pour le moment, on peut dire que votre classe peut travailler avec tous les types de données, mais les choses se corsent un peu à l'utilisation... Vous serez peut-être tentés de faire une classe par type de donnée (`SoloInt`, `SoloString`).
Et c'est là que la généricité est pratique. Car avec ceci, vous allez pouvoir savoir ce que contient votre objet **`Solo`**, et vous n'aurez qu'une seule classe à développer !
Voilà le diagramme de classe de cet objet :



Et voici son code :

Code : Java

```
public class Solo<T> {

        /**
* Variable d'instance
*/
        private T valeur;

        /**
* Constructeur par défaut
*/
        public Solo(){
                this.valeur = null;
        }

        /**
* Constructeur avec paramètre
* Inconnu pour l'instant
* @param val
*/
        public Solo(T val){
                this.valeur = val;
        }


        /**
* Définit la valeur avec le paramètre
* @param val
*/
        public void setValeur(T val){
                this.valeur = val;
        }

        /**
* retourne la valeur déjà "castée" par la signature de la méthode !
* @return
*/
        public T getValeur(){
```

```
                return this.valeur;
        }
}
```

Impressionnant, n'est-ce pas ?
Dans cette classe, le **T** n'est pas encore défini. Vous le ferez à l'instanciation de cette classe. Par contre, une fois instancié avec un type, l'objet ne pourra travailler qu'avec le type de données que vous lui avez spécifié ! Exemple de code :

Code : Java

```
public class Test {

        public static void main(String[] args) {
                // TODO Auto-generated method stub
                Solo<Integer> val = new Solo<Integer>(12);
                int nbre = val.getValeur();
        }
}
```

Ce code fonctionne très bien, mais si vous essayez de faire ceci :

Code : Java

```
public class Test {

        public static void main(String[] args) {
                // TODO Auto-generated method stub
                Solo<Integer> val = new Solo<Integer>("toto"); //Ici
on essaie de mettre une chaîne de caractère à la place d'un entier
                int nbre = val.getValeur();
        }
}
```

...ou encore ceci :

Code : Java

```
public class Test {

        public static void main(String[] args) {
                // TODO Auto-generated method stub
                Solo<Integer> val = new Solo<Integer>(12);
                val.setValeur(12.2f);   //Ici on essaie de mettre
un float à la place d'un entier
        }
}
```

...vous verrez une erreur dans votre zone de saisie. Ceci vous indique que votre objet ne reçoit pas le bon type d'argument, ou que votre réceptacle n'a pas le bon type de données ! Dans tous les cas de figure, il y a conflit entre le type de données que vous avez passé à votre instance lors de sa création et le type de données que vous essayez d'utiliser avec celle-ci !

Par contre, vous devez savoir que cette classe ne fonctionne pas seulement avec des **Integer**. Vous pouvez utiliser tous les types que vous souhaitez !

Voici une démonstration de mes dires :

Code : Java

```
public class Test {
```

```
        public static void main(String[] args) {
                // TODO Auto-generated method stub
                Solo<Integer> val = new Solo<Integer>();
                Solo<String> valS = new Solo<String>("TOTOTOTO");
                Solo<Float> valF = new Solo<Float>(12.2f);
                Solo<Double> valD = new Solo<Double>(12.202568);
        }
}
```

Vous devez avoir remarqué que je n'ai pas utilisé ici les types de données que vous utilisez pour déclarer des variables de type primitif ! Ce sont les classes de ces types primitifs !

En effet, lorsque vous déclarez une variable de type primitif, vous pouvez utiliser leurs classes de définition, mais c'est rarement utilisé car très lourd à la lecture. Par exemple :

Code : Java

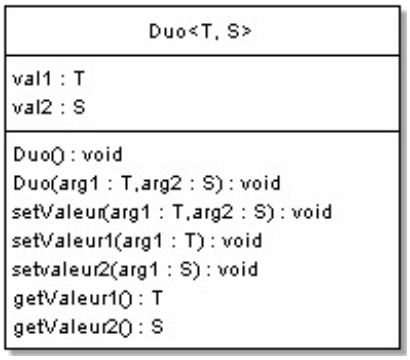
```
public class Test{

    public static void main(String[] args){

        int i = new Integer(12); // est équivalent à int i = 12;
        double d = new Double(12.2586); // est équivalent à double d
= 12.2586;
        //...
    }
}
```

Bon ! Maintenant que vous avez un bel exemple de généricité, nous allons complexifier un peu les choses !

Retour en haut

Plus loin dans la généricité !

Vous devez savoir que la généricité peut être multiple !
Nous avons créé une classe **Solo**, mais rien ne vous empêche de créer une classe **Duo**, qui elle, prend deux paramètres génériques ! Voici la modélisation de cette classe :



Vous pouvez voir que cette classe prend deux types de références non encore définies !
Voilà le code source de cette classe :

Code : Java

```
public class Duo<T, S> {

	/**
* Variable d'instance de type T
*/
	private T valeur1;
	/**
* Variable d'instance de type S
*/
	private S valeur2;

	/**
* Constructeur par défaut
*/
	public Duo(){
		this.valeur1 = null;
		this.valeur2 = null;
	}

	/**
* Constructeur avec paramètres
* @param val1
* @param val2
*/
	public Duo(T val1, S val2){
		this.valeur1 = val1;
		this.valeur2 = val2;
	}

	/**
* Méthodes d'initialisation des deux valeurs
* @param val1
* @param val2
*/
	public void setValeur(T val1, S val2){
		this.valeur1 = val1;
		this.valeur2 = val2;
	}

	/**
* Retourne la valeur T
* @return
*/
	public T getValeur1() {
```

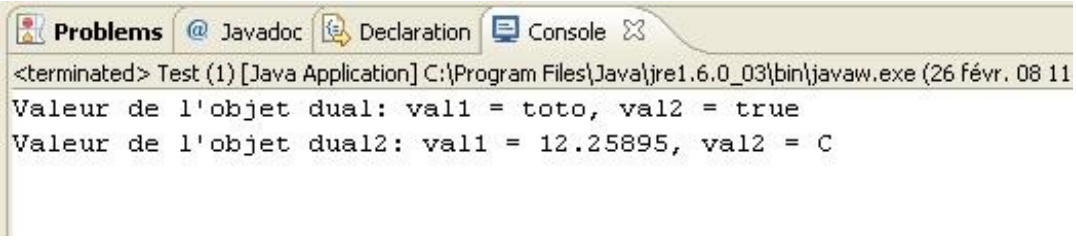
```
                return valeur1;
        }

        /**
* Définit la valeur T
* @param valeur1
*/
        public void setValeur1(T valeur1) {
                this.valeur1 = valeur1;
        }

        /**
* retourne la valeur S
* @return
*/
        public S getValeur2() {
                return valeur2;
        }

        /**
* définit la valeur S
* @param valeur2
*/
        public void setValeur2(S valeur2) {
                this.valeur2 = valeur2;
        }

}
```

Voici un code que vous pouvez tester :

Code : Java

```
public class Test {

        public static void main(String[] args) {

                Duo<String, Boolean> dual = new Duo<String,
Boolean>("toto", true);
                System.out.println("Valeur de l'objet dual: val1 = "
+ dual.getValeur1() + ", val2 = " + dual.getValeur2());

                Duo<Double, Character> dual2 = new Duo<Double,
Character>(12.25895, 'C');
                System.out.println("Valeur de l'objet dual2: val1 =
" + dual2.getValeur1() + ", val2 = " + dual2.getValeur2());

        }
}
```

Et voici le résultat :



Vous voyez qu'il n'y a rien de bien méchant ici. Ce principe fonctionne exactement comme l'exemple précédent. La seule différence est dans le fait qu'il n'y a pas un, mais deux paramètres génériques !

Attends une minute... Lorsque je déclare une référence de type **`Duo<String, Boolean>`**, je ne peux plus la changer en un autre type !

En fait, avec ce que je vous ai fait voir, non.
Pour le moment, si vous faites :

Code : Java

```
public class Test {

        public static void main(String[] args) {

                Duo<String, Boolean> dual = new Duo<String,
Boolean>("toto", true);
                System.out.println("Valeur de l'objet dual: val1 = "
+ dual.getValeur1() + ", val2 = " + dual.getValeur2());
                dual = new Duo<Double, Character>();
        }
}
```

vous violez la contrainte que vous avez émise lors de la déclaration du type de référence ! Mais il existe un moyen de contourner ça.
Tout simplement en disant, à la déclaration, que votre objet va accepter tous types de références ! Comment en utilisant ce qu'on appelle le **wildcard** : **?**.
Comme ceci :

Code : Java

```
public class Test {

        public static void main(String[] args) {

                Duo<?, ?> dual = new Duo<String, Boolean>("toto",
true);

                System.out.println("Valeur de l'objet dual: val1 = "
+ dual.getValeur1() + ", val2 = " + dual.getValeur2());
                dual = new Duo<Double, Character>();
                dual = new Duo<Integer, Float>();
                dual = new Duo<Solo, Solo>();
        }
}
```

Avec ce type de déclaration, votre objet accepte bien n'importe quel type de référence !
Intéressant, non ?
Donc si vous suivez bien, on va pouvoir encore corser la chose !

Retour en haut

Généricité et collection

Vous pouvez aussi utiliser la généricité sur les objets servant à gérer des collections.
C'est même l'un des points les plus utiles de la généricité !

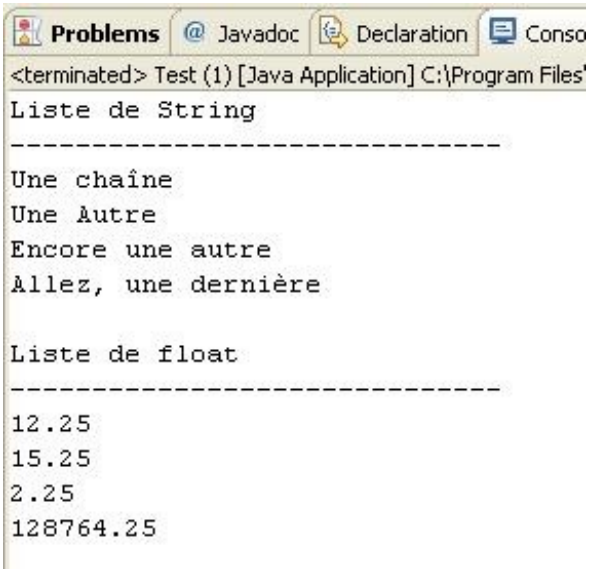
En effet, lorsque vous listiez le contenu d'un **ArrayList** par exemple, vous n'étiez JAMAIS sûrs à 100 % de savoir sur quel type de référence vous alliez tomber... Eh bien ce calvaire est terminé et le polymorphisme va pouvoir réapparaître, plus puissant que jamais !

Voyez comment utiliser (même si vous l'aviez deviné) la généricité avec les collections :

Code : Java

```
import java.util.ArrayList;


public class Test {

        public static void main(String[] args) {

                System.out.println("Liste de String");
                System.out.println("------------------------------
");
                ArrayList<String> listeString= new
ArrayList<String>();
                listeString.add("Une chaîne");
                listeString.add("Une Autre");
                listeString.add("Encore une autre");
                listeString.add("Allez, une dernière");

                for(String str : listeString)
                        System.out.println(str);



                System.out.println("\nListe de float");
                System.out.println("------------------------------
");

                ArrayList<Float> listeFloat = new
ArrayList<Float>();
                listeFloat.add(12.25f);
                listeFloat.add(15.25f);
                listeFloat.add(2.25f);
                listeFloat.add(128764.25f);

                for(float f : listeFloat)
                        System.out.println(f);
        }
}
```

Voici le résultat de ce code :



La généricité sur les listes est régie par les mêmes lois vues précédemment !
Pas de type `float` dans un `ArrayList<String>`.

Vu qu'on y va *crescendo*, on pimente à nouveau le tout !
Retour en haut

Héritage et généricité

Là où les choses sont pernicieuses, c'est quand vous utilisez des classes usant de la généricité avec des objets usant de la notion d'héritage !
L'héritage dans la généricité est une des choses les plus complexes à comprendre en Java. Pourquoi ? Tout simplement parce qu'elle va à l'encontre de ce que vous avez appris jusqu'à présent...

Acceptons le postulat suivant

Nous avons une classe **Voiture** dont hérite une autre classe **VoitureSansPermis**, ce qui nous donnerait le diagramme suivant :

Jusque-là, c'est simplissime.
Maintenant, ça se complique :

Code : Java

```
import java.util.ArrayList;


public class Test {

        public static void main(String[] args) {

                ArrayList<Voiture> listVoiture = new
ArrayList<Voiture>();
                ArrayList<VoitureSansPermis> listVoitureSP = new
ArrayList<VoitureSansPermis>();

                listVoiture = listVoitureSP;//Interdit ! ! ! !
        }
}
```

Je sais que même si vous aviez l'habitude de la covariance des variables, ceci n'existe pas sous cette forme avec la généricité !

Pourquoi cela ?

Imaginez deux secondes que l'instruction interdite soit permise !
Dans **listVoiture**, vous avez le contenu de la liste des voitures sans permis, et rien ne vous empêche d'ajouter une voiture... Là où le problème prend toute son envergure, c'est lorsque vous allez vouloir sortir toutes les voitures sans permis de votre variable **listVoiture**, eh oui ! Vous y avez rajouté une voiture !
Lors du balayage de la liste vous aurez, à un moment, une référence de type **VoitureSansPermis** à qui vous tentez d'affecter une référence de type **Voiture**. Voilà pourquoi ceci est INTERDIT ! !

L'une des solutions consiste à utiliser le **wildcard**.

Je vais maintenant vous indiquer quelque chose d'important !
Avec la généricité, vous pouvez aller encore plus loin... Nous avons vu comment restreindre le contenu d'une de nos listes. Mais nous pouvons aussi élargir son contenu ! Si je veux par exemple qu'un **ArrayList** puisse avoir toutes les instances de **Voiture** et de ses classes filles. **Comment faire ?**

Ce qui suit s'applique aussi aux interfaces susceptibles d'être implémentées par une classe !

Attention les yeux, ça pique :

Code : Java

```
import java.util.ArrayList;
public class Test {

	public static void main(String[] args) {
		//Voici un ArrayList n'acceptant que des instances
de Voiture ou de ses sous-classes
		ArrayList<? extends Voiture> listVoitureSP = new
ArrayList<VoitureSansPermis>();
	}
}
```

Et une application de ceci consiste à faire des méthodes génériques, comme par exemple avoir une méthode qui permette de lister toutes les valeurs de notre **ArrayList** citée précédemment. Voici :

Code : Java

```
import java.util.ArrayList;
public class Test {

	public static void main(String[] args) {

		ArrayList<? extends Voiture> listVoitureSP = new
ArrayList<VoitureSansPermis>();
		afficher(listVoitureSP);
	}

	/**
* Méthode générique !
* @param <T>
* @param list
*/
	static void afficher(ArrayList<? extends Voiture> list){
		for(Voiture v : list)
			System.out.println(v.toString());
	}

}
```

Eh ! Attends, on a voulu ajouter des objets dans notre collection et le programme ne compile plus !

Oui, alors, ce que je ne vous avait pas dis, c'est que, dès que vous utilisez le wildcard combiné avec le mot clé **extends**, vos listes seront verrouillées en insertion : **Elles se transforment en collections en lecture seule...**

Pourquoi ça ?

En fait, il faut déjà savoir que c'est à la compilation du programme que Java ne vous laisse pas faire.
Le verrou vient du fait que, vu que le wildcard signifie "***tout objet***" combiné avec **extends** signifiant "***héritant***", au moment de la compilation, Java n'a aucune idée de l'objet qu'on vient d'assigner à notre collection : les concepteurs ont donc préféré bloquer ce mode d'utilisation.

Par contre, ce type d'utilisation fonctionne à merveille pour la lecture :

Code : Java

```
import java.util.ArrayList;
import java.util.List;


public class Main {

 public static void main(String[] args){
```

```
        //Liste de voiture
        ArrayList<Voiture> listVoiture = new ArrayList<Voiture>();
        listVoiture.add(new Voiture());
        listVoiture.add(new Voiture());

        ArrayList<VoitureSansPermis> listVoitureSP = new
      ArrayList<VoitureSansPermis>();
        listVoitureSP.add(new VoitureSansPermis());
        listVoitureSP.add(new VoitureSansPermis());

        affiche(listVoiture);
        affiche(listVoitureSP);
       }

       /**
      * Avec cette méthode, on accepte aussi bien les collections de
      Voiture
      * que les collection de VoitureSansPermis
      * @param list
      */
       static void affiche(List<? extends Voiture> list){

        for(Voiture v : list)
         System.out.print(v.toString());
       }

      }
```

Avant que vous ne posiez la question : NON ! Déclarer la méthode comme ceci `affiche(List<Voiture> list)` ne vous permet pas de parcourir des listes de **VoitureSansPermis**, même si celle-ci hérite de la classe **Voiture**.
Les méthodes déclarées avec un type générique sont verrouillées afin de n'être utilisées qu'avec ce type bien précis, toujours pour les mêmes raisons que ci-dessus !

Pfiou ! C'est bien compliqué tout ça...

Attendez, ce n'est pas encore fini. Nous avons vu comment élargir le contenu de nos collections (pour la lecture), nous allons voir comment restreindre les collections acceptées par nos méthodes.
La méthode :

Code : Java

```
      static void affiche(List<? extends Voiture> list){

        for(Voiture v : list)
       System.out.print(v.toString());
      }
```

Autorise un objet de type **List** de n'importe quel type dont **Voiture** est la super classe.

L'instruction suivante signifie :
La méthode autorise un objet de type **List** de n'importe quel super classe de la classe **Voiture**, **Voiture** y compris.

Code : Java

```
      static void affiche(List<? super Voiture> list){

        for(Object v : list)
       System.out.print(v.toString());
      }
```

Ce code fonctionne donc parfaitement :

Code : Java

```
public static void main(String[] args){

  //Liste de voiture
  List<Voiture> listVoiture = new ArrayList<Voiture>();
  listVoiture.add(new Voiture());
  listVoiture.add(new Voiture());

  ArrayList<Object> listVoitureSP = new ArrayList<Object>();
  listVoitureSP.add(new Object());
  listVoitureSP.add(new Object());

  affiche(listVoiture);
 }

 /**
* Avec cette méthode, on accepte aussi bien les collections de
Voiture
* que les collection d' Object : super classe de toutes les classes
* @param list
*/
 static void affiche(List<? super Voiture> list){

  for(Object v : list)
   System.out.print(v.toString());
 }
```

Je conçois bien que ceci est un peu ardu à comprendre... Mais vous en aurez sûrement besoin dans une de vos prochaines applications !

Bon : je crois que nous avons fait un bon tour du sujet même si nous n'avons pas tout abordé... Allez, le topo classique.

Retour en haut

Ce qu'il faut retenir

- La généricité est apparue depuis le JDK 1.5.
- Ce concept est très utile pour développer des objets pouvant travailler avec plusieurs types de données.
- Vous passerez donc **moins de temps à développer** des classes pour traiter de façon identique des données différentes.
- La généricité permet de **ré-utiliser le polymorphisme** sans risque avec les collections.
- Cela vous permet une meilleure **robustesse du code**.
- Vous pouvez coupler les collections avec la généricité !
- Le `wildcard (?)` permet de dire que ***n'importe quel type peut être traité*** et donc accepté !
- Dès que le `wildcard (?)` est utilisé sur une collection, **cela revient à rendre la dite collection en lecture seule !**
- Vous pouvez élargir le champ d'acceptation d'une collection générique grâce au mot clé `extends`.
- L'instruction `<? extends MaClasse>` autorise toutes les collections de classes ayant pour super type `MaClasse`.
- L'instruction `<? super MaClasse>` autorise toutes les collections de classes ayant pour type `MaClasse` et tous ses supers types !
- Pour ce genre de cas, les **méthodes génériques** sont particulièrement adaptées et **permettent d'utiliser le polymorphisme dans toute sa splendeur** !

Retour en haut

J'espère que ce chapitre n'a été trop... lourd...
En attendant, nous avons presque terminé cette seconde partie... La programmation d'interface graphique se rapproche !
Mais il nous reste une dernière chose à aborder qui peut s'avérer importante ! **La réflexivité**.

Retour en haut

Java et la réflexivité

Voici un chapitre qui, je pense, ne vous servira pas tout de suite.
Cependant, il me semble important d'en parler...

La réflexivité n'est en fait que le moyen de connaître toutes les informations concernant une classe donnée. Vous pourrez même créer des instances de classe de façon dynamique grâce à ceci.
Je pense faire une partie sur les ***design pattern*** après celle sur les interfaces graphiques ! Et, à ce moment, vous aurez sans doute besoin des connaissances de ce chapitre, surtout lorsque nous aborderons le ***pattern factory***.

En attendant, je pense que ce chapitre va tout de même vous intéresser !
Alors, allons-y...

Commençons par le commencement

La **réflexivité**, aussi appelée **introspection**, consiste à découvrir de façon dynamique des informations propres à une classe Java ou à un objet. Ceci est notamment utilisé au niveau de la machine virtuelle Java lors de l'exécution de votre programme. En gros, votre machine virtuelle stocke les informations relatives à une classe dans un objet.

Concrètement, que se passe-t-il ?

Au chargement d'une classe Java, votre JVM crée automatiquement un objet. Celui-ci récupère toutes les caractéristiques de votre classe ! Il s'agit d'un objet **Class**.
Exemple: si vous avez créé trois nouvelles classes Java, à l'exécution de votre programme, la JVM va créer un objet **Class** pour chacune d'elles.

Comme vous devez vous en douter, cet objet possède une multitude de méthodes qui permettent d'avoir tous les renseignements possibles et imaginables sur une classe.

Dans ce chapitre, nous allons visiter la classe **String**.
Créez un nouveau projet ainsi qu'une classe contenant la méthode **main**.
Voici deux façons de récupérer un objet **Class** :

Code : Java

```
public class Test {

	public static void main(String[] args) {

		Class c = String.class;
		Class c2 = new String().getClass();
		/*La fameuse méthode finale dont je vous parlais dans
le chapitre sur l'héritage
Cette méthode vient de la classe Object
*/
	}
}
```

Maintenant que vous savez récupérer un objet **Class**, nous allons tout de suite voir ce qu'il sait faire !

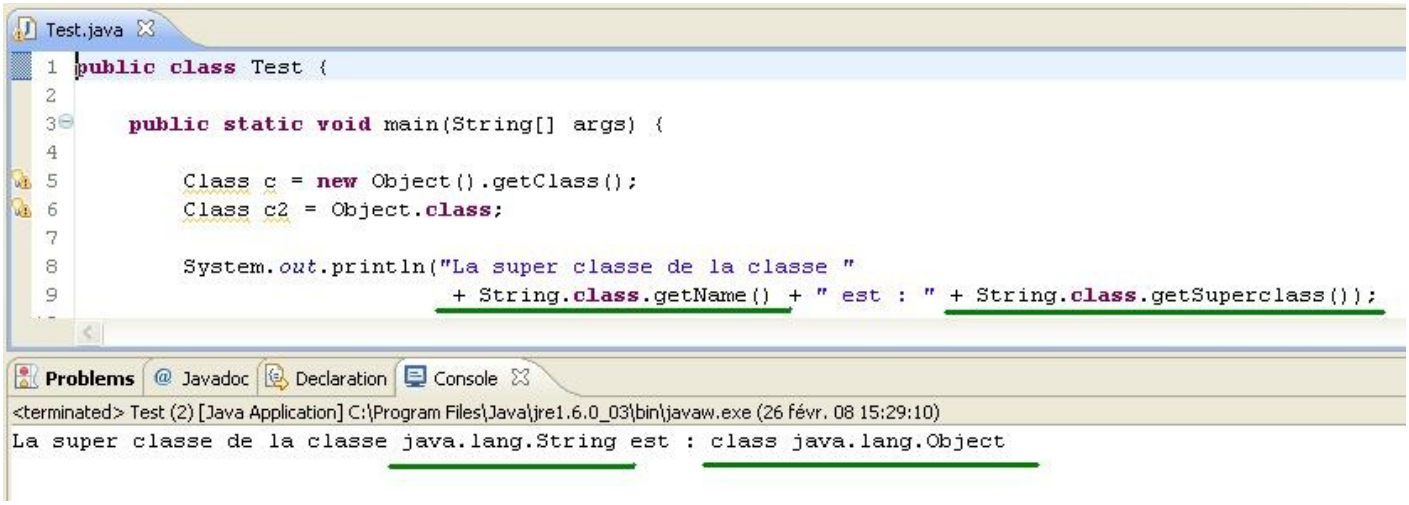
Retour en haut

Interroger un objet Class

Dans ce sous-chapitre, nous allons voir une partie des choses que sait faire un objet **Class**. Je ne vais pas tout vous montrer... De toute façon, je pense que vous êtes à même de chercher et de trouver tous seul maintenant. Vous avez l'habitude de manipuler des objets, à présent...

Connaître la super classe d'une classe

Ce qui nous donne :



La classe **Object** n'a pas de super-classe...

Voyez plutôt :

Connaître la liste des interfaces

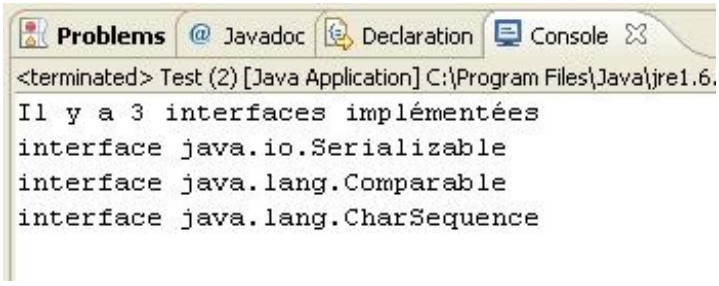
Vous pouvez tester ce code :

Code : Java

```
public class Test {

	public static void main(String[] args) {

		//On récupère un objet Class
		Class c = new String().getClass();
		//La méthode getInterfaces retourne un tableau de
Class
		Class[] faces = c.getInterfaces();
```

```
                //Pour voir le nombre d'interfaces
                System.out.println("Il y a " + faces.length + "
interfaces implémentées");
                //On parcourt le tableau d'interfaces
                for(int i = 0; i < faces.length; i++)
                        System.out.println(faces[i]);
        }
}
```

Ce qui nous donne :



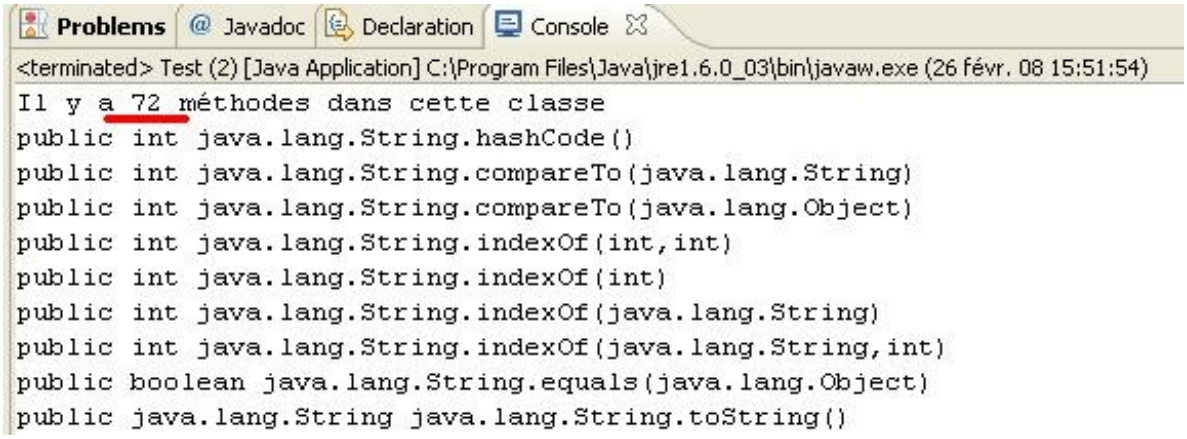
Connaître la liste des méthodes de la classe

La méthode **getMethods()** de l'objet **Class** nous retourne un tableau d'objets **Method** présents dans le **package java.lang.reflect**.
Vous pouvez soit faire l'import à la main, soit déclarer un tableau d'objets **Method** et utiliser le raccourci ***Ctrl + Shift + O***.
Voici un code qui retourne la liste des méthodes de la classe **String** :

Code : Java

```
import java.lang.reflect.Method;

public class Test {

        public static void main(String[] args) {

                Class c = new String().getClass();
                Method[] m = c.getMethods();

                System.out.println("Il y a " + m.length + " méthodes
dans cette classe");
                //On parcourt le tableau de méthodes
                for(int i = 0; i < m.length; i++)
                        System.out.println(m[i]);

        }
}
```

Voici un morceau du résultat car, comme vous pourrez le voir, il y a beaucoup de méthodes dans la classe **String**.

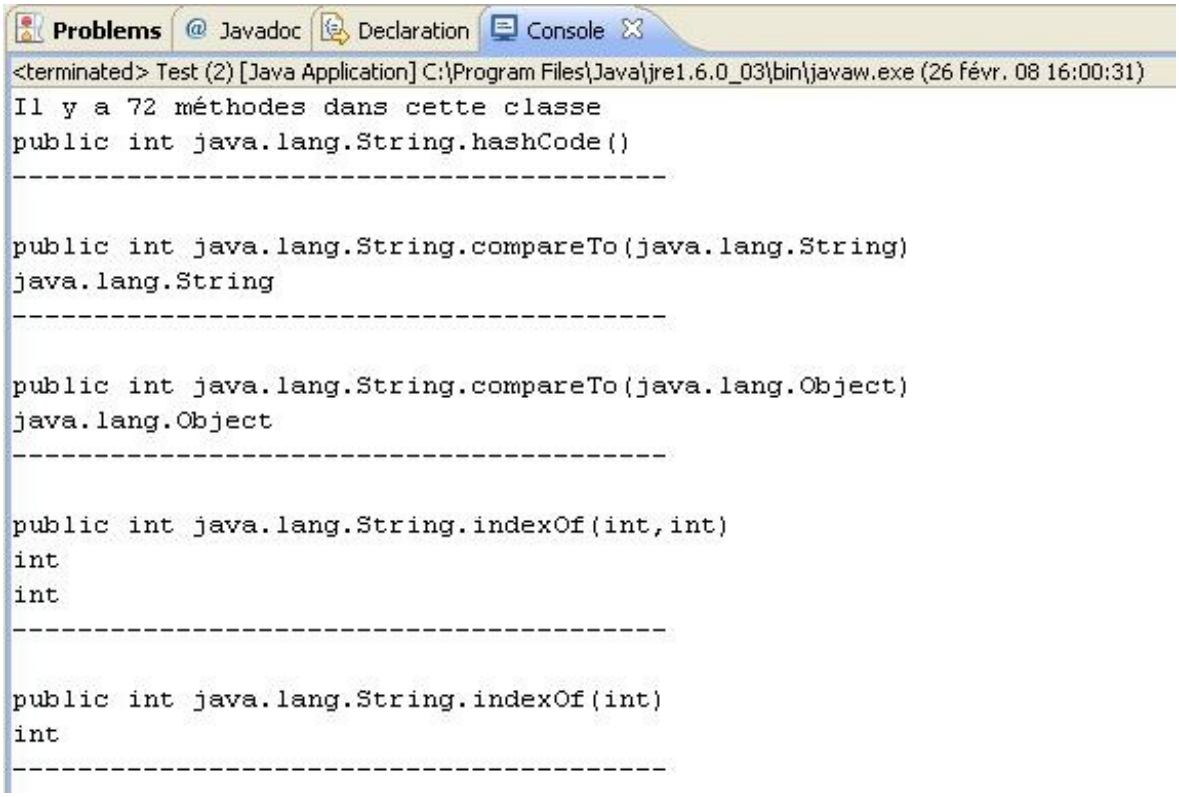


Vous pouvez constater que l'objet Method regorge lui aussi de méthodes intéressantes. Voici un code qui affiche la liste des méthodes ainsi que la liste des attributs qu'elles prennent :

Code : Java

```
import java.lang.reflect.Method;

public class Test {

	public static void main(String[] args) {

		Class c = new String().getClass();
		Method[] m = c.getMethods();

		System.out.println("Il y a " + m.length + " méthodes
dans cette classe");
		//On parcourt le tableau de méthodes
		for(int i = 0; i < m.length; i++)
		{
			System.out.println(m[i]);

			Class[] p = m[i].getParameterTypes();
			for(int j = 0; j < p.length; j++)
				System.out.println(p[j].getName());

			System.out.println("------------------------
----------------\n");
		}

	}
}
```

Et voilà :



Connaître la liste des champs (variable de classe ou d'instance) de la classe

Ici, nous allons appliquer la même méthodologie que pour la liste des méthodes sauf que cette fois, la méthode invoquée retourne un tableau d'objets **Field**. Voici un code qui affiche la liste des champs de la classe **String**.

Code : Java

```
import java.lang.reflect.Field;

public class Test {

        public static void main(String[] args) {

                Class c = new String().getClass();
                Field[] m = c.getFields();

                System.out.println("Il y a " + m.length + " champs
dans cette classe");
                //On parcourt le tableau de méthodes
                for(int i = 0; i < m.length; i++)
                        System.out.println(m[i].getName());

        }
}
```

Connaître la liste des constructeurs de la classe

Ici, nous utiliserons un objet **Constructor** pour lister les constructeurs de la classe :

Code : Java

```
import java.lang.reflect.Constructor;

public class Test {
```

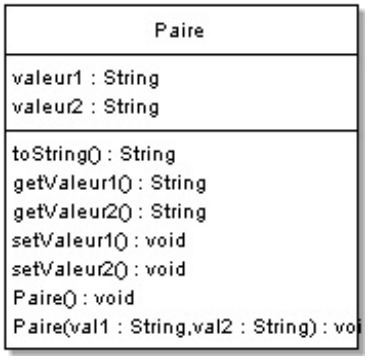
```
	public static void main(String[] args) {

		Class c = new String().getClass();
		Constructor[] construc = c.getConstructors();
		System.out.println("Il y a " + construc.length + "
constructeurs dans cette classe");
		//On parcourt le tableau des constructeur
		for(int i = 0; i < construc.length; i++){
			System.out.println(construc[i].getName());

			Class[] param =
construc[i].getParameterTypes();

			for(int j = 0; j < param.length; j++)
				System.out.println(param[j]);

			System.out.println("-------------------------
------------\n");
		}

	}
}
```

Vous pouvez donc constater que l'objet **Class** regorge de méthodes en tout genre !
Maintenant, si nous essayons d'exploiter un peu plus celles-ci...

Retour en haut

Instanciation dynamique

Nous allons voir une petite partie de la puissance de cette classe (pour l'instant).
Dans un premier temps, créez un nouveau projet avec une méthode **main**, ainsi qu'une classe correspondant à ceci :



Voici son code Java :

Code : Java

```
public class Paire {

        private String valeur1, valeur2;

        public Paire(){
                this.valeur1 = null;
                this.valeur2 = null;
                System.out.println("Instanciation ! !");
        }

        public Paire(String val1, String val2){
                this.valeur1 = val1;
                this.valeur2 = val2;

                System.out.println("Instanciation avec des
paramètres ! !");

        }

        public String toString(){
                return  "Je suis un objet qui a pour valeur : " +
this.valeur1 + " - " + this.valeur2;
        }

        public String getValeur1() {
                return valeur1;
        }

        public void setValeur1(String valeur1) {
                this.valeur1 = valeur1;
        }

        public String getValeur2() {
                return valeur2;
        }

        public void setValeur2(String valeur2) {
                this.valeur2 = valeur2;
        }
}
```

Le but du jeu maintenant consiste à créer un objet **Paire** sans utiliser l'opérateur **new**.

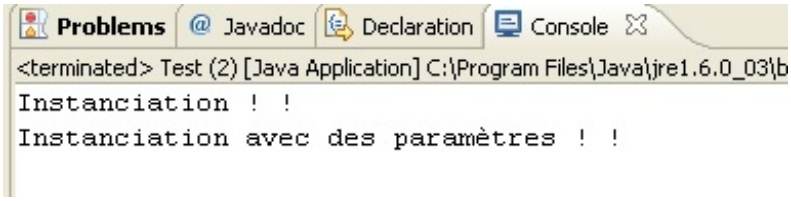
Pour instancier un nouvel objet **`Paire`**, nous allons tout d'abord récupérer ses constructeurs. Ensuite, nous allons préparer un tableau contenant les données à insérer. Puis nous invoquerons la méthode **`toString()`**.

Regardez comment procéder ; par contre, il y a moultes exceptions :

Code : Java

```
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

public class Test {

	public static void main(String[] args) {

		String nom = Paire.class.getName();


			try {
				//On crée un objet Class
				Class cl = Class.forName(nom);
				//Nouvelle instance de la classe
Paire
				Object o = cl.newInstance();

				//On crée les paramètres du
constructeur
				Class[] types = new
Class[]{String.class, String.class};
				//On récupère le constructeur avec
les deux paramètres
				Constructor ct =
cl.getConstructor(types);
				//On instancie l'objet avec le
constructeur surchargé !
				Object o2 = ct.newInstance(new
String[]{"valeur 1 ", "valeur 2"} );

			} catch (SecurityException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			} catch (IllegalArgumentException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			} catch (ClassNotFoundException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			} catch (InstantiationException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			} catch (IllegalAccessException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			} catch (NoSuchMethodException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			} catch (InvocationTargetException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}

	}
}
```

Et le résultat donne :



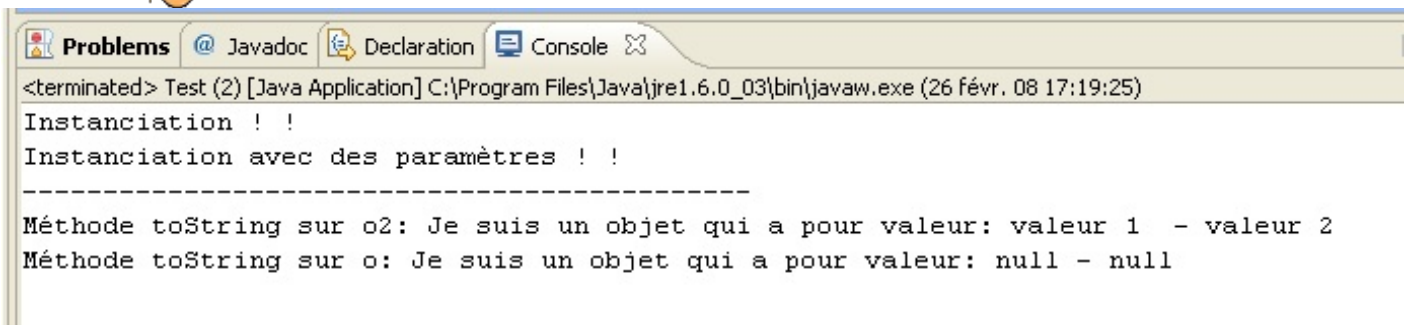
Nous pouvons maintenant appeler la méthode **`toString()`** du deuxième objet... oh et soyons fous, sur les deux :

Code : Java

```
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class Test {

	public static void main(String[] args) {

		String nom = Paire.class.getName();


			try {
				//On crée un objet Class
				Class cl = Class.forName(nom);
				//Nouvelle instance de la classe
Paire
				Object o = cl.newInstance();

				//On crée les paramètres du
constructeur
				Class[] types = new
Class[]{String.class, String.class};
				//On récupère le constructeur avec
les deux paramètres
				Constructor ct =
cl.getConstructor(types);
				//On instancie l'objet avec le
constructeur surchargé !
				Object o2 = ct.newInstance(new
String[]{"valeur 1 ", "valeur 2"} );

				//On va chercher la méthode
toString, elle n'a aucun paramètre
				Method m = cl.getMethod("toString",
null);
				//La méthode invoke exécute la
méthode sur l'objet passé en paramètre,
				// pas de paramètre, donc null en
deuxième paramètre de la méthode invoke !

				System.out.println("----------------
------------------------------");
				System.out.println("Méthode " +
m.getName() + " sur o2: " +m.invoke(o2, null));
				System.out.println("Méthode " +
m.getName() + " sur o: " +m.invoke(o, null));

			} catch (SecurityException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			} catch (IllegalArgumentException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			} catch (ClassNotFoundException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			} catch (InstantiationException e) {
```

```
                                        // TODO Auto-generated catch block
                                        e.printStackTrace();
                                } catch (IllegalAccessException e) {
                                        // TODO Auto-generated catch block
                                        e.printStackTrace();
                                } catch (NoSuchMethodException e) {
                                        // TODO Auto-generated catch block
                                        e.printStackTrace();
                                } catch (InvocationTargetException e) {
                                        // TODO Auto-generated catch block
                                        e.printStackTrace();
                                }

                        }
                }
```

Et le résultat :



Voilà : nous venons de créer deux instances d'une classe sans passer par l'opérateur **new**. Mieux encore ! Car nous avons même pu appeler une méthode de nos instances !

Je ne vais pas m'attarder trop longtemps sur ce sujet... Mais gardez en tête que cette façon de faire, même si elle est très lourde, pourrait vous être utile. Et là, je repense à mon ***pattern factory***. En quelques mots, il s'agit d'une classe Java qui ne fait que créer des instances !

Bon. Je crois que vous avez bien mérité une pause. Les deux derniers chapitres ont été assez éprouvants...
Un petit topo et en route pour la partie 3 !

Retour en haut

Ce qu'il faut retenir

- Lorsque votre JVM interprète votre programme, elle crée automatiquement un objet `Class` pour chaque classe chargée.
- Avec un tel objet, vous pouvez connaître absolument tout sur votre classe.
- L'objet `Class` utilise des sous-objets comme `Method`, `Field`, `Constructor`... qui permettent de travailler avec vos différents objets, mais aussi avec ceux présents dans Java.
- Avec cet objet, vous pouvez créer des instances de vos classes Java sans utiliser `new`.
- **Par contre, vous devez savoir que les performances restent médiocres.**

Allez : je vous fais grâce d'un QCM... Une fois n'est pas coutume !

Retour en haut

Voilà, c'était le dernier chapitre de cette partie !
Il y a eu pas mal de choses vues ici...

Je ne vais faire de long discours maintenant que vous êtes si près de la programmation événementielle...
Alors... Rendez-vous dans la troisième partie.

Retour en haut

J'espère que cette partie vous a plu et que vous avez appris plein de bonne choses !
J'ai volontairement omis de parler des ***flux*** et des ***threads*** dans cette partie. Je préfère avoir des cas bien concrets à vous soumettre pour ça...

Bon : je sais que beaucoup d'entre vous l'attendent avec impatience, alors voici la partie sur la **programmation événementielle** !

Retour en haut

Partie 3 : Java et la programmation événementielle

Dans cette partie, nous aborderons la programmation événementielle comme le stipule le titre.
Par là, entendez ***programmation d'interface graphique***, ou IHM, ou encore GUI.

Nous utiliserons essentiellement les bibliothèques **`Swing`** et **`AWT`** présentes d'office dans Java.

Nous verrons ce qui forme, je pense, les fondements de base ! Nous n'entrerons pas dans les détails, enfin pas trop...
Je ne vais pas faire de long discours maintenant, je sais que vous êtes impatients... alors go !
Retour en haut

Votre première fenêtre

Dans ce chapitre, nous allons apprendre à nous servir de l'objet **`JFrame`**, présent dans le package **`Swing`**.

À la fin du chapitre, vous serez à même de créer une fenêtre, de choisir sa taille...
Trêve de bavardage inutile, commençons tout de suite !

L'objet JFrame

Nous y voilà... Avant de nous lancer à corps perdu, vous devez savoir ce que nous allons utiliser...
Vu que nous allons développer des interfaces avec `swing`, vous devez savoir que toutes les classes `swing` se trouvent dans le package **`javax.swing`**. Mais ne vous y trompez pas, nous allons utiliser aussi des objets **`awt (java.awt)`**, mais pas de composants !
Pour faire simple, un composant peut être un bouton, une zône de texte, une case à cocher... Bref, tout ce qui peut interagir avec vous !

Il est très fortement recommandé de ne pas mélanger les composants `swing` et `awt` ! ! Ceci pour cause de conflit ! Si vous faites ceci vous aurez de très grandes difficultés à faire une IHM stable et valide ! En effet, pour faire simple, `swing` et `awt` se basent sur les mêmes fondements mais diffèrent sur l'utilisation de ces fondements...

Pourquoi ?
Tout simplement car les objets de ces deux packages ne sont pas construits de la même façon et que des conflits peuvent survenir (superposition de composants...).

Je ne vous demande pas de créer un projet avec une classe **`main`**, celui-ci doit être prêt depuis des lustres, facile !

Pour utiliser une fenêtre de type **`JFrame`**, vous devez instancier celui-ci. Comme ceci :

Code : Java

```
import javax.swing.JFrame;

public class Test {

        public static void main(String[] args){

                JFrame fenetre = new JFrame();

        }
}
```

Eh ! Lorsque j'exécute mon code, rien ne s'affiche !

Oui, parce que par défaut, votre **`JFrame`** n'est pas visible...
Pour pouvoir l'afficher à l'écran, vous devez lui dire "***sois visible***", comme ça :

Code : Java

```
import javax.swing.JFrame;

public class Test {

        public static void main(String[] args){

                JFrame fenetre = new JFrame();
                fenetre.setVisible(true);
        }
}
```

Et lorsque vous exécuterez votre code, vous pourrez voir ceci :

Eh, mais tu te moques de nous ! Elle est minuscule !

Bienvenue dans le monde de la programmation événementielle ! Il faut que vous vous y fassiez... Vos composants ne sont pas intelligents : il va falloir leur dire tout ce qu'ils doivent faire !

Bon, pour avoir une fenêtre plus conséquente, il faudrait :

- qu'elle soit plus grande ; en effet, c'est mieux ;
- qu'elle ait un titre (c'est pas du luxe !) ;
- si elle pouvait être centrée au milieu de mon écran, ce serait parfait !

Par contre, vous ne l'avez peut-être pas remarqué mais, lorsque vous cliquez sur la croix rouge (pour fermer votre fenêtre), cette action ne termine pas le processus dans Eclipse !

Pour réellement terminer le processus de l'application, vous devrez ajouter une autre instruction.

Pour chacune des choses que je viens d'énumérer, il y a aura une méthode à appeler pour que votre **`JFrame`** sache à quoi s'en tenir !
Voici un code qui reprend toutes nos doléances :

Code : Java

```
import javax.swing.JFrame;

public class Test {

        public static void main(String[] args){

                JFrame fenetre = new JFrame();

                //Définit un titre pour votre fenêtre
                fenetre.setTitle("Ma première fenêtre java");
                //Définit une taille pour celle-ci ; ici, 400 px de
large et 500 px de haut
                fenetre.setSize(400, 500);
                //Nous allons maintenant dire à notre objet de se
positionner au centre
                fenetre.setLocationRelativeTo(null);
                //Terminer le processus lorsqu'on clique sur
"Fermer"

fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                fenetre.setVisible(true);
        }
}
```

Et voici le résultat :

Par contre, je pense qu'il vaudrait mieux que nous ayons notre propre objet. Comme ça, on n'aura pas à redéfinir les attributs à chaque fois... Donc créons notre propre classe !

Retour en haut

Votre fenêtre héritée

Pour commencer, effacez tout le code que vous avez écrit dans votre méthode **main**. Ensuite, créez une classe que nous appellerons "**Fenetre**", et faites-la hériter de **JFrame**.
Voilà le code de cette classe pour le moment :

Code : Java

```
import javax.swing.JFrame;

public class Fenetre extends JFrame{

}
```

Nous allons maintenant créer notre constructeur et, dans celui-ci, nous mettrons nos instructions à l'intérieur.
Ce qui nous donne :

Code : Java

```
import javax.swing.JFrame;

public class Fenetre extends JFrame{

     public Fenetre(){
          //Définit un titre pour votre fenêtre
          this.setTitle("Ma première fenêtre java");
          //Définit une taille pour celle-ci ; ici, 400 px de large
et 500 px de haut
          this.setSize(400, 500);
          //Nous allons maintenant dire à notre objet de se
positionner au centre
          this.setLocationRelativeTo(null);
          //Ferme-toi lorsqu'on clique sur "Fermer" !
          this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

          this.setVisible(true);
     }
}
```

Ensuite, vous avez le choix, soit :

- vous conservez votre classe contenant la méthode `main` et vous créez une instance de **Fenetre**
- vous effacez cette classe, et vous mettez votre méthode `main` dans votre classe **Fenetre**. Vous devez tout de même créer une instance de votre **Fenetre**...

Personnellement, je préfère sortir ma méthode `main` dans une classe à part... Mais je ne vous force à rien !
Quelque soit l'emplacement de votre **main**, la ligne de code qui suit doit y figurer :

Code : Java

```
Fenetre fen = new Fenetre();
```

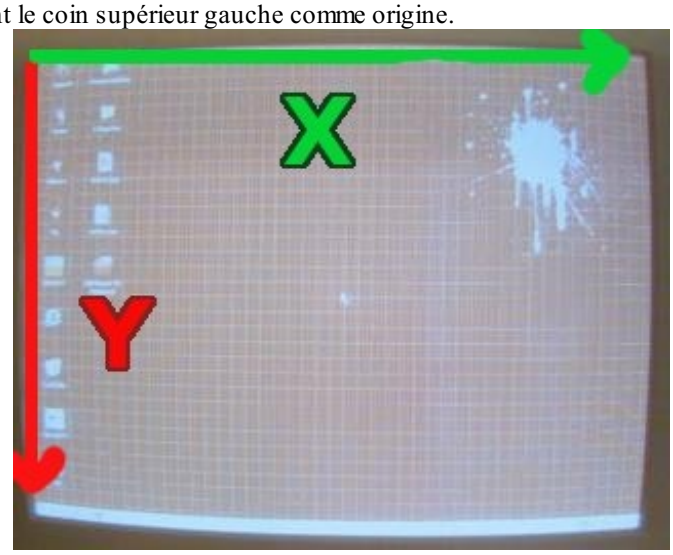
Exécutez votre nouveau code et... vous avez exactement la même chose que précédemment !
Vous conviendrez que c'est tout de même plus pratique de ne plus écrire les même choses à chaque fois... Comme ça, vous avez une classe qui va se charger de l'affichage de votre programme futur !
Faisons un léger tour d'horizon de cette classe.
Retour en haut

Des méthodes et encore des méthodes

Je vais vous faire une petite liste des méthodes que vous serez susceptibles d'utiliser.

Positionner sa fenêtre à l'écran

Déjà, nous avons centré notre fenêtre, mais vous auriez peut-être voulu la positionner ailleurs. Pour faire ceci, vous avez la méthode **`setLocation(int x, int y)`**.
Avec cette méthode, vous pouvez spécifier où doit se situer votre fenêtre sur votre écran. Les coordonnées, exprimées en pixels, sont basées sur un repère prenant le coin supérieur gauche comme origine.



La première valeur de la méthode vous positionne sur l'axe X, 0 correspondant à l'origine ; les valeurs positives déplacent la fenêtre vers la droite, et les valeurs négatives vous font sortir de l'écran vers la gauche. La même règle s'applique pour les valeurs Y, excepté que les valeurs positives font descendre la fenêtre en commençant par l'origine, et les valeurs négatives font sortir la fenêtre par le haut !

Empêcher le redimensionnement de la fenêtre

Pour faire ceci, il vous suffit d'invoquer la méthode **`setResizable(false);`**, et de le repasser à **`setResizable(true)`** pour le rendre actif !

Faire que votre fenêtre soit toujours au premier plan

Il s'agit là encore d'une méthode qui prend un booléen en paramètre. Passer `true` mettra votre fenêtre au premier plan quoi qu'il advienne, et passer `false` annulera le statut. Cette méthode est **`setAlwaysOnTop(boolean b)`**.

Retirer les contours et les boutons de contrôles

Pour ce faire, il vous suffit d'utiliser la méthode **`setUndecorated(Boolean b)`**.

Je ne vais pas faire le tour de toutes les méthodes maintenant... De toute façon, nous allons nous servir de pas mal d'autres dans un futur très proche...

C'est bien joli tout ça, mais on aimerait bien pouvoir mettre des trucs dans notre fenêtre !

Bien sûr : mais avant, il vous faut encore apprendre une bricole... En fait votre fenêtre, telle qu'elle apparaît, vous cache quelques petites choses...
Retour en haut

Ce que vous cache votre fenêtre

Vous pensez, et c'est légitime, que votre fenêtre est toute simple, dépourvue de tout composant (hormis les contours).
Eh bien vous vous trompez !
Une **`JFrame`** est découpée en plusieurs parties :

- **la fenêtre**,
- le **`RootPane`**, le ***container*** (*conteneur*) principal qui contient les autres composants,
- le **`LayeredPane`**, qui forme juste un panneau composé du `ContentPane` et de la barre de menu (`MenuBar`),
- la **`MenuBar`**, la barre de menu quand il y en a une...
- le **`ContentPane`** ; c'est dans celui-ci que nous mettrons nos composants,
- et le **`GlassPane`**, couche utilisée pour intercepter les actions de l'utilisateur avant qu'elles ne parviennent aux composants.

Pas de panique... Nous n'allons nous servir que du `contentPane` et, pour le récupérer, nous n'avons qu'à utiliser la méthode **`getContentPane()`** de la classe **`JFrame`**.

Cependant, nous allons utiliser un composant autre que le `contentPane`. Nous utiliserons un **`JPanel`**.

Il existe d'autres types de fenêtres. La **`JWindow`**, une **`JFrame`** sans bord et non *draggable* (déplaçable), et la **`JDialog`**, une fenêtre non redimensionnable. Mais nous n'en parlerons pas ici...

Bon : qu'est-ce qu'on attend ?
Ah oui ! Le topo et le QCM...

Retour en haut

Ce qu'il faut retenir

- Les composants `swing` se trouvent dans `javax.swing`, et les `awt` dans `java.awt`.
- Il ne faut pas mélanger les composants `swing` et `awt` ! !
- Une `JFrame` est constituée de plusieurs composants.
- Par défaut, une fenêtre a une taille minimale, et n'est pas visible.
- Un composant doit être bien paramétré pour qu'il fonctionne à notre convenance.

Retour en haut

J'ose espérer que ce premier chapitre était à votre goût...
Il n'était pas trop difficile pour commencer. Mais les choses vont vite se compliquer...

Continuons avec **les conteneurs**.

Retour en haut

Une histoire de conteneur

Dans cette partie, nous allons aborder la notion de conteneur, et plus particulièrement le `JPanel` (pour l'instant...)
Vous verrez pas mal de choses qui vous seront très utiles plus tard, enfin j'espère.
Mais surtout, à la fin de cette partie, nous pourrons presque commencer à mettre des composants sur nos fenêtres...

Allez, je vous sens impatients...

Créez un conteneur pour votre fenêtre

Comme je vous l'ai dit auparavant, nous allons utiliser un **`JPanel`**, composant de type *container*, dont la vocation est d'accueillir d'autres objets de même type, ou des objets de type composant (bouton, case à cocher...).

Mais dans l'absolu, n'importe quel composant peut accueillir un autre composant ! Nous verrons ça...

Voici la marche à suivre

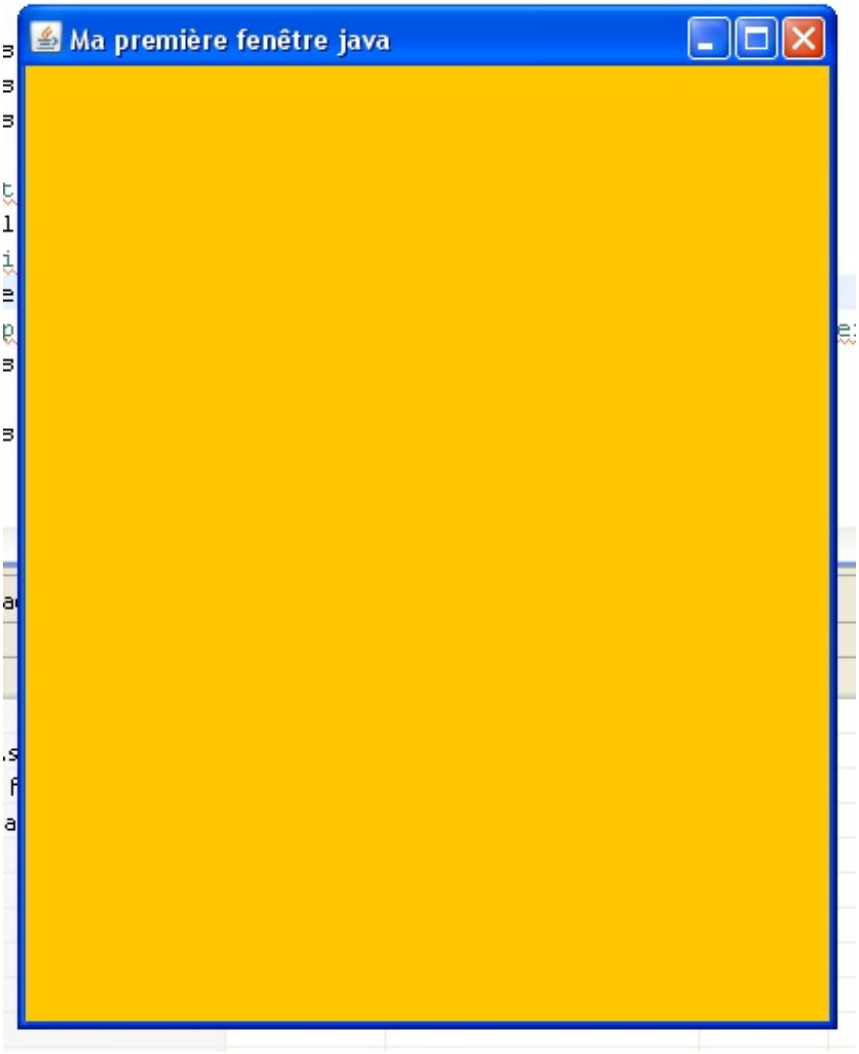
- Nous allons importer la classe **`javax.swing.JPanel`** dans notre classe héritée de **`JFrame`**.
- Nous allons instancier un **`JPanel`**, lui spécifier une couleur de fond pour pouvoir mieux le voir !
- Dire à notre **`JFrame`** que ce sera notre **`JPanel`** qui sera son **`contentPane`**.

Rien de bien sorcier en somme. Qu'attendons-nous ?

Code : Java

```
import java.awt.Color;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class Fenetre extends JFrame {

        public Fenetre(){

                this.setTitle("Ma première fenêtre java");
                this.setSize(400, 500);
                this.setLocationRelativeTo(null);

                //Instanciation d'un objet JPanel
                JPanel pan = new JPanel();
                //Définition de sa couleur de fond
                pan.setBackground(Color.ORANGE);
                //On prévient notre JFrame que ce sera notre JPanel
qui sera son contentPane
                this.setContentPane(pan);

                this.setVisible(true);
        }
}
```

Et voici le résultat :



C'est un bon début, mais je vois que vous êtes frustrés car il n'y a pas beaucoup de changement par rapport à la dernière fois...

Eh bien c'est maintenant que les choses deviennent intéressantes !
Avant de vous faire utiliser des composants comme des boutons ou autre chose, nous allons nous amuser avec notre panneau.
Plus particulièrement avec un objet qui a pour rôle de dessiner et de peindre notre composant... Ça vous tente ? Alors, Go !

Retour en haut

L'objet Graphics

Cet objet a une particularité de taille ! **Vous ne pouvez l'utiliser que si, et seulement si le système vous l'a donné !**
Et pour bien comprendre le mode de fonctionnement de vos futurs containers (ou composants), nous allons faire une classe héritée de **JPanel** : appelons-la **Panneau** ; nous allons faire petit tour d'horizon du fonctionnement de celle-ci !

Voici le code de cette classe :

Code : Java

```
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {

        public void paintComponent(Graphics g){
              //Vous pourrez voir cette phrase à chaque fois que la
méthode est invoquée !
               System.out.println("Je suis exécutée ! ! !");
               g.fillOval(20, 20, 75, 75);
        }
}
```

Hop là ! Qu'est-ce que c'est que cette méthode ?

Cette méthode est celle que l'objet appelle pour se peindre sur notre fenêtre, et si vous réduisez cette dernière et que vous l'affichez de nouveau, c'est encore cette méthode qui est appelée pour afficher notre composant ! Idem si vous redimensionnez votre fenêtre... En plus, on n'a même pas à redéfinir de constructeur, cette méthode est appelée automatiquement !

Pour personnaliser des composants, ceci est très pratique car vous n'aurez **JAMAIS** à l'appeler de vous-mêmes, **ceci est automatique** ! Tout ce que vous pourrez faire, c'est forcer l'objet à se repeindre, mais ce n'est pas cette méthode que vous invoquerez... Nous y reviendrons !

Vous aurez constaté que cette méthode possède un argument et qu'il s'agit du fameux objet **Graphics** tant convoité. Nous reviendrons sur l'instruction **`g.fillOval(20, 20, 75, 75);`** mais vous verrez ce qu'elle fait lorsque vous exécuterez votre programme...

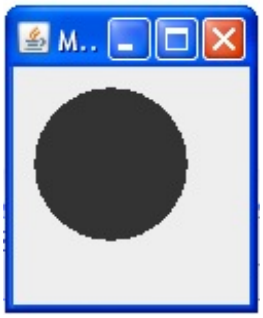
Et maintenant, dans notre classe **Fenetre** :

Code : Java

```
import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class Fenetre extends JFrame {

        public Fenetre(){

                this.setTitle("Ma première fenêtre java");
                this.setSize(100, 150);
                this.setLocationRelativeTo(null);
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setContentPane(new Panneau());

                this.setVisible(true);
        }

}
```

J'ai réduit la taille de ma fenêtre car mes *screenshots* devenaient vraiment encombrants...

Exécutez votre **`main`**, et vous devriez avoir ceci :



Une fois votre fenêtre affichée, étirez-la, réduisez-la...
À présent, vous pouvez voir ce qu'il se passe lorsque vous interagissez avec votre fenêtre ! **Celle-ci met à jour ses composants à chaque changement d'état ou de statut !** Et l'intérêt d'avoir une classe héritée d'un container ou d'un composant, c'est que nous pouvons redéfinir la façon dont est peint ce composant sur la fenêtre !

Donc, après cette mise en bouche... Si nous explorions un peu plus les capacités de notre objet **`Graphics`** ?

Retour en haut

Plus loin dans le Graphics

Comme vous avez pu le voir, l'objet Graphics permet, entre autres, de tracer des ronds... Mais cet objet possède tout un tas de méthodes plus pratiques et amusantes les unes que les autres...
Nous ne les verrons pas toutes mais vous aurez déjà de quoi faire...

Pour commencer, reprenons la méthode que j'ai utilisée précédemment : `g.fillOval(20, 20, 75, 75);`.
Si nous avions à traduire cette instruction en français, ça donnerait :
"***Trace un rond plein en commençant à dessiner sur l'axe x à 20 pixels, sur l'axe y à 20 pixels, et fais en sorte que mon rond fasse 75 pixels de large et 75 pixels de haut.***"

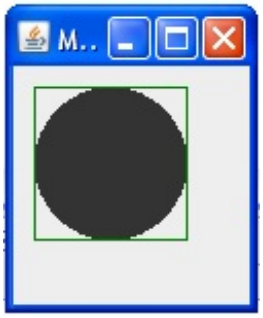
C'est simple à comprendre, n'est-ce pas ?

Oui, mais si je veux que mon rond soit centré et qu'il y reste ?

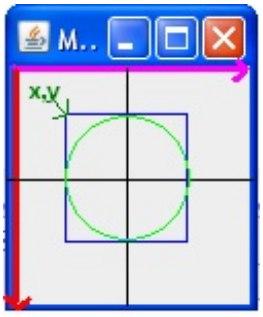
C'est dans ce genre de cas qu'il est intéressant d'avoir une classe héritée !
Vu que nous sommes dans notre objet **`JPanel`**, nous avons accès à ses données et j'ajouterais, pile au bon moment : lorsque nous allons le dessiner !

En effet, il y a des méthodes dans les objets composants qui nous retournent sa largeur (**`getWidth()`**) et sa hauteur (**`getHeight()`**) !
Par contre, réussir à centrer un rond dans un `JPanel` en toute circonstance demande un peu de calcul mathématique de base, une pincée de connaissances et un soupçon de logique !

Reprenons notre fenêtre telle qu'elle est en ce moment. Vous pourrez constater que les coordonnées de départ ne correspondent pas au départ du cercle en lui-même, mais au point de départ du carré qui entoure ce cercle !



Ceci signifie que, si nous voulons que notre cercle soit centré à tout moment, il faut que notre carré soit centré et donc, que le centre de celui-ci corresponde au centre de notre fenêtre ! J'ai essayé de faire un schéma représentant ce que nous devons obtenir.



Ainsi, le principe est de prendre la largeur et la longueur de notre composant ainsi que la largeur et la longueur du carré qui englobe notre rond ! Facile, jusqu'à présent...
Maintenant, pour trouver où se situe le point où doit commencer le dessin, il faut prendre la moitié de la largeur de notre composant, moins la moitié de la largeur de notre rond, tout ceci pour l'axe x et y.

Pour que notre rond soit le plus optimisé, nous allons prendre pour taille de notre carré la moitié de notre fenêtre !

Donc, pour simplifier le tout, nous nous retrouvons à calculer la moitié de la moitié de la largeur et de la hauteur... Ce qui revient,

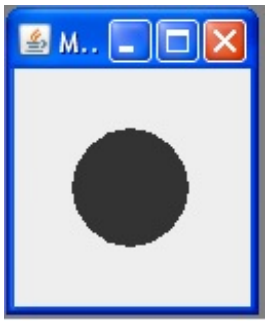
au final, à diviser la largeur et la hauteur par 4...

Voici le code qui fait ceci :

Code : Java

```
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {

        public void paintComponent(Graphics g){

                int x1 = this.getWidth()/4;
                int y1 = this.getHeight()/4;

                System.out.println("largeur = " + this.getWidth() +
", longueur = " + this.getHeight());
                System.out.println(" coordonnée de début d'affichage
x1 = " + x1 + " y1 = " + y1);

                g.fillOval(x1, y1, this.getWidth()/2,
this.getHeight()/2);
        }

}
```

Ce qui nous donne :



Bon, l'objet **Graphics** sait plein d'autres choses : peindre des ronds vides, par exemple.
Sans rire... Maintenant que vous avez vu un peu comment fonctionne cet objet, nous allons utiliser ses méthodes...

La méthode `drawOval(int x1, int y1, int width, int height)`

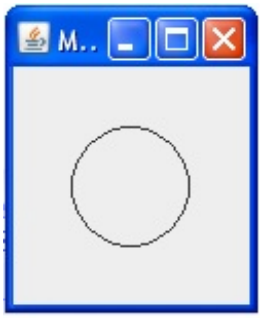
Il s'agit de la méthode qui permet de dessiner un rond vide. Celle-ci fonctionne exactement de la même manière que la méthode **fillOval**.
Voici un code mettant en oeuvre cette méthode :

Code : Java

```
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {

        public void paintComponent(Graphics g){

                int x1 = this.getWidth()/4;
                int y1 = this.getHeight()/4;

                g.drawOval(x1, y1, this.getWidth()/2,
this.getHeight()/2);
        }
```

```
    }
```

Résultat :



Si vous spécifiez une largeur différente de la hauteur, ces méthodes dessineront une forme ovale !

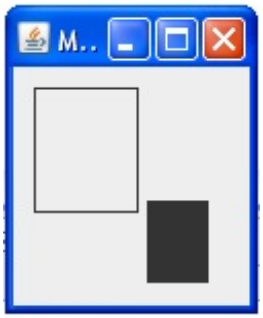
La méthode `drawRect(int x1, int y1, int width, int height)`

Cette méthode permet de dessiner des rectangles vides. Bien sûr, son homologue **`fillRect`** existe. Ces deux méthodes fonctionnent aussi comme les précédentes, voyez plutôt ce code :

Code : Java

```
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {

        public void paintComponent(Graphics g){

                g.drawRect(10, 10, 50, 60);
                g.fillRect(65, 65, 30, 40);
        }
}
```

Et le résultat :



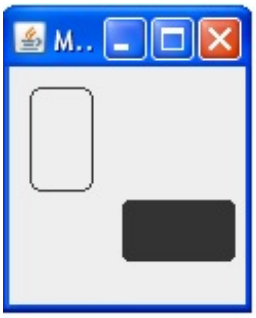
La méthode `drawRoundRect(int x1, int y1, int width, int height, int arcWidth, int arcHeight)`

Il s'agit de la même chose que précédemment, mis à part que le rectangle sera arrondi. Arrondi défini par les valeurs passées dans les deux derniers paramètres.

Code : Java

```
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {

        public void paintComponent(Graphics g){

                g.drawRoundRect(10, 10, 30, 50, 10, 10);
                g.fillRoundRect(55, 65, 55, 30, 5, 5);
        }
}
```

Résultat :



La méthode `drawLine(int x1, int y1, int x2, int y2)`

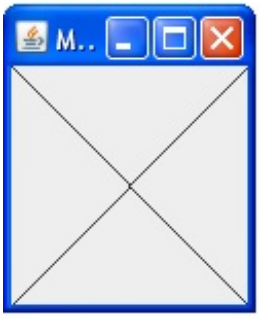
Celle-ci vous permet de tracer des lignes droites ! Il vous suffit de lui spécifier les coordonnées de départ et d'arrivée de la ligne... Simple aussi, n'est-ce pas ?

Dans ce code, je trace les diagonales de notre conteneur :

Code : Java

```
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {

        public void paintComponent(Graphics g){

                g.drawLine(0, 0, this.getWidth(), this.getHeight());
                g.drawLine(0, this.getHeight(), this.getWidth(), 0);
        }
}
```

Résultat :



La méthode

drawPolygon(int[] x, int[] y, int nbrePoints)

Avec cette méthode, vous pourrez dessiner des polygones de votre composition. Eh oui... C'est à vous de définir les coordonnées de tous les points qui forment votre polygone !
Le dernier paramètre de cette méthode est le nombre de points formant votre polygone. Ainsi, vous ne serez pas obligés de créer deux fois le point d'origine pour boucler votre figure. Java fermera celle-ci automatiquement en reliant le dernier point de votre tableau au premier...

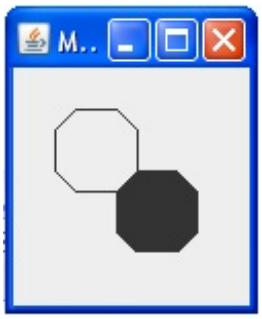
Je vous conseille vivement de faire un schéma pour vous aider... Cette méthode a aussi son homologue pour dessiner les polygones remplis : **fillPolygon**.

Code :

Code : Java

```
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {

        public void paintComponent(Graphics g){

                int x[] = {20, 30, 50, 60, 60, 50, 30, 20};
                int y[] = {30, 20, 20, 30, 50, 60, 60, 50};
                g.drawPolygon(x, y, 8);

                int x2[] = {50, 60, 80, 90, 90, 80, 60, 50};
                int y2[] = {60, 50, 50, 60, 80, 90, 90, 80};
                g.fillPolygon(x2, y2, 8);
        }
}
```

Résultat :



Vous avez aussi une méthode qui prend exactement les mêmes arguments et qui, elle, trace plusieurs lignes ! Cette méthode s'appelle : **drawPolyline(int[] x, int[]y, int nbrePoints)**.

Cette méthode va dessiner les lignes correspondant aux coordonnées que vous lui passerez dans les tableaux, sachant que lorsque vous passez à l'indice supérieur dans vos tableaux, la méthode prend automatiquement les valeurs de l'indice précédent comme point d'origine.

Cette dernière ne fait pas le lien entre la première et la dernière valeur de vos tableaux... Vous pouvez essayer le code précédent, en remplaçant **drawPolygon** par cette méthode et vous verrez...

La méthode drawString(String str, int x, int y)

Voici la méthode qui vous permet d'écrire du texte... Elle est très simple à comprendre puisqu'il vous suffit de lui passer la phrase à écrire et de lui spécifier à quelles coordonnées commencer !

Code :

Code : Java

```
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {

        public void paintComponent(Graphics g){

                g.drawString("Tiens ! le Site du Zér0 ! ! !", 10,
20);

        }
}
```

Résultat :

Vous pouvez modifier la couleur (et ça s'applique aussi pour les autres méthodes) et la police d'écriture... Pour redéfinir la police d'écriture, vous devez créer un objet **Font**. Regardez comment faire :

Code : Java

```
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;

import javax.swing.JPanel;

public class Panneau extends JPanel {

        public void paintComponent(Graphics g){

                Font font = new Font("Courier", Font.BOLD, 20);
                g.setFont(font);

                g.setColor(Color.red);
                g.drawString("Tiens ! le Site du Zér0 ! ! !", 10,
20);

        }
}
```

Et le résultat :

La méthode `drawImage(Image img,int x, int y, Observer obs);`

Ici, vous devrez charger votre image grâce à trois objets :

- un objet **`Image`**
- un objet **`ImageIO`**
- un objet **`File`**.

Vous allez voir que l'utilisation de ces objets est très simple... Nous déclarons un objet de type **`Image`**, nous allons l'initialiser en utilisant une méthode statique de l'objet **`ImageIO`**, qui, elle, prend un objet **`File`** en paramètre. Ça paraît compliqué comme ça, mais vous allez voir... **Par contre, notre image sera stockée à la racine de notre projet !**

Et en ce qui concerne le dernier paramètre de notre méthode **`drawImage`**, il s'agit de l'objet qui est censé observer l'image. Ici, nous allons mettre notre objet **`Panneau`**, donc **`this`**.

Avec cette méthode, l'image sera dessinée avec ses propres dimensions... Si vous voulez que l'image prenne l'intégralité de votre container, il faut utiliser le constructeur suivant : `drawImage(Image img, int x, int y, int width, int height, Observer obs).`

Code :

Code : Java

```
import java.awt.Graphics;
import java.awt.Image;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.JPanel;

public class Panneau extends JPanel {

        public void paintComponent(Graphics g){
                try {
                        Image img = ImageIO.read(new
File("images.jpg"));
                        g.drawImage(img, 0, 0, this);
                        //Pour une image de fond
                        //g.drawImage(img, 0, 0, this.getWidth(),
this.getHeight(), this);
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }
}
```

Pour bien vous montrer la différence, j'ai ajouté une couleur de fond rouge. Et, si vous vous demandez comment j'ai fait... j'ai mis un rectangle de couleur rouge de la taille de mon composant...

Voici les résultats selon ce que vous avez choisi :

Maintenant, je pense qu'il est temps de vous présenter le petit cousin de notre objet `Graphics` !

Retour en haut

Le cousin caché : l'objet Graphics2D

Voici une amélioration de l'objet **Graphics**, et vous allez vite comprendre pourquoi...

Pour utiliser cet objet, il nous suffit de caster l'objet **Graphics** en **Graphics2D**, et surtout de ne pas oublier d'importer sa classe !
Celle-ci se trouve dans le package **java.awt**.

Voilà notre début de code :

Code : Java

```
import javax.swing.JPanel;
import java.awt.Graphics;
import java.awt.Graphics2D;

public class Panneau extends JPanel{

     public void paintComponent(Graphics g){

        Graphics2D g2d = (Graphics2D) g;
     }
}
```

L'une des possibilités qu'offre cet objet n'est autre que la possibilité de peindre des objets avec des dégradés de couleurs... Cette opération n'est pas du tout difficile à effectuer. Pour ce faire, il vous suffit d'utiliser un objet **GradientPaint** et une méthode de l'objet **Graphics2D**.

Nous n'allons pas reprendre tous les cas que nous avons vus jusqu'à présent... Juste deux ou trois pour que vous voyez bien la différence.

Commençons par notre objet **GradientPaint**, voici comme l'initialiser :

Code : Java

```
GradientPaint gp = new GradientPaint(0, 0, Color.RED, 30, 30,
Color.cyan, true);
```

Vous devrez mettre à jour vos imports... Vous devez ajouter ici : import **java.awt.GradientPaint;**

Alors, que veut dire tout ceci ? Voici le détail du constructeur utilisé ici :

- paramètre 1 : la coordonnée x où doit commencer la couleur 1 ;
- paramètre 2 : la coordonnée y où doit commencer la couleur 1 ;
- paramètre 3 : couleur 1 ;
- paramètre 4 : la coordonnée x2 où doit commencer la couleur 2 ;
- paramètre 5 : la coordonnée y2 où doit commencer la couleur 2 ;
- paramètre 6 : couleur 2 ;
- paramètre 7 : booléen pour définir si le dégradé doit se répéter.

Entre ces deux points se créera le dégradé des deux couleurs spécifiées !

Ensuite, pour utiliser ce dégradé dans une forme, il suffit de mettre à jour votre objet **Graphics2D**, comme ceci :

Code : Java

```
GradientPaint gp = new GradientPaint(0, 0, Color.RED, 30, 30,
```

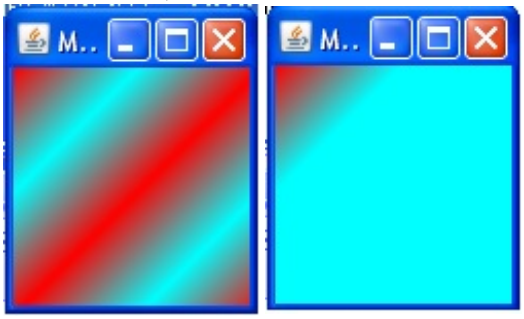
```
        Color.cyan, true);
        g2d.setPaint(gp);
```

Le code entier de cet exemple est :

Code : Java

```
import java.awt.Color;
import java.awt.Font;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;
import javax.swing.JPanel;

public class Panneau extends JPanel {

        public void paintComponent(Graphics g){

                Graphics2D g2d = (Graphics2D)g;
                GradientPaint gp = new GradientPaint(0, 0,
Color.RED, 30, 30, Color.cyan, true);
                g2d.setPaint(gp);
                g2d.fillRect(0, 0, this.getWidth(),
this.getHeight());

        }
}
```

Voici les résultats obtenus, l'un avec le booléen à **true**, et l'autre à **false** :



Votre dégradé est oblique (rien ne m'échappe, à moi...). Ce sont les coordonnées choisies qui influent sur la direction du dégradé. Dans notre exemple, nous partons du point de coordonnées (0, 0) vers le point de coordonnées (30, 30). Si vous vouliez un dégradé vertical, il faut juste mettre la valeur de x2 à 0 et voilà :

Un petit cadeau :

Code : Java

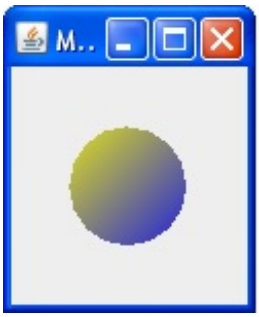
```
import java.awt.Color;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;

import javax.imageio.ImageIO;
import javax.swing.JPanel;

public class Panneau extends JPanel {

        public void paintComponent(Graphics g){

                Graphics2D g2d = (Graphics2D)g;

                GradientPaint gp = new GradientPaint(0, 0,
Color.RED, 20, 0, Color.magenta, true);
                GradientPaint gp2 = new GradientPaint(20, 0,
Color.magenta, 40, 0, Color.blue, true);
                GradientPaint gp3 = new GradientPaint(40, 0,
Color.blue, 60, 0, Color.green, true);
                GradientPaint gp4 = new GradientPaint(60, 0,
Color.green, 80, 0, Color.yellow, true);
                GradientPaint gp5 = new GradientPaint(80, 0,
Color.yellow, 100, 0, Color.orange, true);
                GradientPaint gp6 = new GradientPaint(100, 0,
Color.orange, 120, 0, Color.red, true);

                g2d.setPaint(gp);
                g2d.fillRect(0, 0, 20, this.getHeight());
                g2d.setPaint(gp2);
                g2d.fillRect(20, 0, 20, this.getHeight());
                g2d.setPaint(gp3);
                g2d.fillRect(40, 0, 20, this.getHeight());
                g2d.setPaint(gp4);
                g2d.fillRect(60, 0, 20, this.getHeight());
                g2d.setPaint(gp5);
                g2d.fillRect(80, 0, 20, this.getHeight());
                g2d.setPaint(gp6);
                g2d.fillRect(100, 0, 40, this.getHeight());

        }
}
```

Quelques exemples de dégradés avec d'autre formes :

Avec un cercle

Code : Java

```
import java.awt.Color;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;

import javax.swing.JPanel;

public class Panneau extends JPanel {

        public void paintComponent(Graphics g){

                Graphics2D g2d = (Graphics2D)g;

                GradientPaint gp = new GradientPaint(20, 20,
Color.yellow, 95, 95, Color.blue, true);
                g2d.setPaint(gp);
                g2d.fillOval(this.getWidth()/4, this.getHeight()/4,
this.getWidth()/2, this.getHeight()/2);

        }
}
```



Avec un rectangle arrondi

Code : Java

```
import java.awt.Color;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;

import javax.swing.JPanel;

public class Panneau extends JPanel {

        public void paintComponent(Graphics g){

                Graphics2D g2d = (Graphics2D)g;

                GradientPaint gp = new GradientPaint(20, 20,
Color.green, 55, 75, Color.orange, true);
                g2d.setPaint(gp);
                g2d.fillRoundRect(30, 30, 75, 75, 10, 10);

        }
}
```

Un peu de texte pour finir

Code : Java

```
import java.awt.Color;
import java.awt.Font;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;

import javax.swing.JPanel;

public class Panneau extends JPanel {

        public void paintComponent(Graphics g){
                g.setColor(Color.BLACK);
                g.fillRect(0, 0, this.getWidth(), this.getHeight());

                Graphics2D g2d = (Graphics2D)g;
                GradientPaint gp = new GradientPaint(0, 0,
Color.cyan, 30, 30, Color.pink, true);
                g2d.setPaint(gp);

                Font font = new Font("Comics Sans MS", Font.BOLD,
14);
                g2d.setFont(font);
                g2d.drawString("On s'amuse les ZérOs !", 10, 15);

                gp = new GradientPaint(-40, -40, Color.red, 100,
100, Color.white, false);
                g2d.setPaint(gp);

                font = new Font("Arial Black", Font.PLAIN, 16);
                g2d.setFont(font);
                g2d.drawString("Moi j'aime bien...", 10, 65);
        }
}
```

Nous avons fait un bon petit tour, là...
Pour les curieux, je vous conseille d'aller voir la page des tutos de Sun Microsystems sur l'objet `Graphics2D`.

Bon, je pense qu'un topo serait le bienvenu...

Retour en haut

Ce qu'il faut retenir

- L'objet `JPanel` se trouve dans le package `javax.swing`.
- **Un `JPanel` peux contenir des composants ou d'autres containers ! !**
- Lorsque vous ajoutez un `JPanel` principal à votre fenêtre, n'oubliez pas de dire à votre fenêtre que ce dernier sera son `contentPane`.
- Pour redéfinir la façon dont l'objet est dessiné sur notre fenêtre, nous devons utiliser la méthode `paintComponent` en créant une classe héritée.
- Cette méthode prend un paramètre, un objet `Graphics`.
- **Cet objet doit nous être fourni par le système** !
- C'est lui que nous allons utiliser pour dessiner dans notre container.
- Pour des dessins plus évolués, vous devrez utiliser l'objet `Graphics2D` qui s'obtient en faisant un cast sur l'objet `Graphics`.

Retour en haut

Et voilà, encore une partie rondement menée.
Maintenant que nous savons comment ajouter un conteneur sur notre fenêtre, nous allons voir comment positionner des composants !

Eh oui : si vous ne dites pas à vos composants où ils doivent aller, ils se mettront à la queue-leu-leu !
Bon : on y va ?

C'est parti pour : **Faire une animation simple**.

Retour en haut

Faire une animation simple

Dans ce chapitre, nous allons voir comment créer une animation simple.

Vous ne pourrez pas faire de jeu à la fin, mais je pense que vous aurez de quoi vous amuser un peu...

Let's go alors...

Les déplacements : principe

Voilà le compte rendu de ce que nous avons :

- une classe héritée de **JFrame**
- une classe héritée de **JPanel** dans laquelle nous faisons de zolis dessins. Un rond en l'occurrence...

Avec ces deux classes, nous allons pouvoir créer un effet de déplacement.
Vous avez bien entendu : j'ai dit un ***effet de déplacement*** !
En réalité, le principe réside dans le fait que vous allez donner des coordonnées différentes à votre rond, et vous allez forcer votre objet **Panneau** à se redessiner ! Tout ceci, *vous l'aviez deviné*, dans une boucle !

Nous allons donc nous préparer à ces nouveautés !
Jusqu'à présent, nous avons utilisé des valeurs fixes pour les coordonnées de notre rond, et il va falloir dynamiser tout ça...

Nous allons donc créer deux variables privées de type **int** dans notre classe **Panneau** : appelons-les **posX** et **posY**.
Pour l'animation que nous allons travailler, notre rond devra provenir de l'extérieur de notre fenêtre. Partons du principe que celui-ci va faire 50 pixels de diamètre : il faudra donc que notre panneau peigne ce rond en dehors de sa zone d'affichage, nous initialiserons donc nos deux variables d'instance à **-50**.

Voilà à quoi ressemble notre classe, maintenant :

Code : Java

```
import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {

        private int posX = -50;
        private int posY = -50;

        public void paintComponent(Graphics g){
                g.setColor(Color.red);
                g.fillOval(posX, posY, 50, 50);
        }

        public int getPosX() {
                return posX;
        }

        public void setPosX(int posX) {
                this.posX = posX;
        }

        public int getPosY() {
                return posY;
        }

        public void setPosY(int posY) {
                this.posY = posY;
        }

}
```

Il ne nous reste plus qu'à faire en sorte que notre rond se déplace : il nous faut donc un moyen de changer les coordonnées de celui-ci, le tout dans une boucle. Nous allons ainsi ajouter une méthode privée dans notre classe **Fenetre** afin de gérer tout cela ; nous appellerons celle-ci en dernier dans notre constructeur. Voici donc à quoi ressemble notre classe **Fenetre** :

Code : Java

```
import java.awt.Dimension;
```

```
import javax.swing.JFrame;


public class Fenetre extends JFrame{

	private Panneau pan = new Panneau();

	public Fenetre(){

		this.setTitle("Animation");
		this.setSize(300, 300);
		this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		this.setLocationRelativeTo(null);
		this.setContentPane(pan);
		this.setVisible(true);

		go();
	}

	private void go(){

		for(int i = -50; i < pan.getWidth(); i++)
		{
			int x = pan.getPosX(), y = pan.getPosY();
			x++;
			y++;
			pan.setPosX(x);
			pan.setPosY(y);
			pan.repaint();
			try {
				Thread.sleep(10);
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
		}

	}
}
```

Hep ! Qu'est-ce que c'est que ces deux instructions à la fin de la méthode **`go()`** ?

Tout d'abord, je pense que, les deux dernières instructions mises à part, vous ne devez pas avoir trop de mal à comprendre ce code.

La première des deux nouvelles instructions est **`pan.repaint()`**. Cette dernière donne l'ordre à votre composant, ici un **`JPanel`**, **de se redessiner**.
La toute première fois, dans le constructeur de notre classe **`Fenetre`**, votre composant invoque la méthode **`paintComponent`** et dessine un rond aux coordonnées que vous lui avez spécifiées. La méthode **`repaint()`** ne fait rien d'autre que de faire à nouveau appel à la méthode **`paintComponent`** ; mais avant, vous avez changé les coordonnées du rond par le biais des accesseurs créés précédemment. Donc à chaque tour de boucle, les coordonnées de notre rond vont changer.

La deuxième instruction est en fait un moyen de **faire une pause dans votre code**...
Celle-ci met en attente votre programme pendant un laps de temps défini dans la méthode **`sleep()`**, ce temps est exprimé en millièmes de secondes (plus le temps d'attente est court, plus votre animation sera rapide). **`Thread`** est en fait un objet qui permet de créer un nouveau processus dans un programme, ou de gérer le processus principal.
Dans tous les programmes, **il y a au moins un processus**, celui qui est en cours d'exécution. Mais vous verrez plus tard qu'il est possible de diviser certaines tâches en plusieurs processus afin de ne pas avoir de perte de temps et de performances dans vos programmes. Pour le moment, sachez que vous pouvez faire des pauses dans vos programmes avec cette instruction :

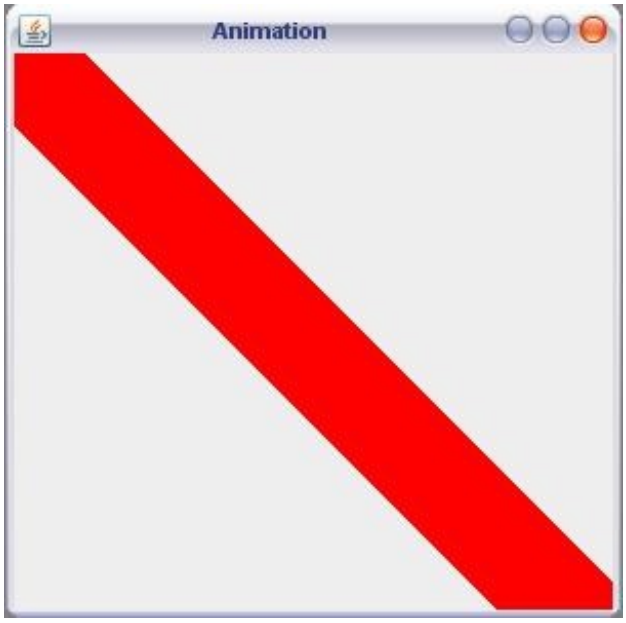
Code : Java

```
try{
```

```
        Thread.sleep(1000);//Ici une pause d'une seconde
        }catch(InterruptedException e) {

                e.printStackTrace();

        }
```

Cette instruction est dite "à risque", vous devez donc l'entourer d'un bloc **`try{}catch(){}`** afin de capturer les exceptions potentielles ! **Sinon : ERREUR DE COMPILATION !**

Maintenant que toute la lumière est faite sur cette affaire, exécutez votre code, et vous obtenez :



Bien sûr, cette image est le résultat final, vous devriez avoir vu votre rond bouger mais au lieu d'être clair, il a laissé une trainée derrière lui...

Pourquoi ?

C'est simple : vous avez demandé à votre objet **`Panneau`** de se redessiner, mais il a gardé les précédents passages de votre rond sur lui-même ! Pour résoudre ce problème, il suffit d'effacer ceux-ci avant de redessiner votre rond.

Comment fait-on ça ?

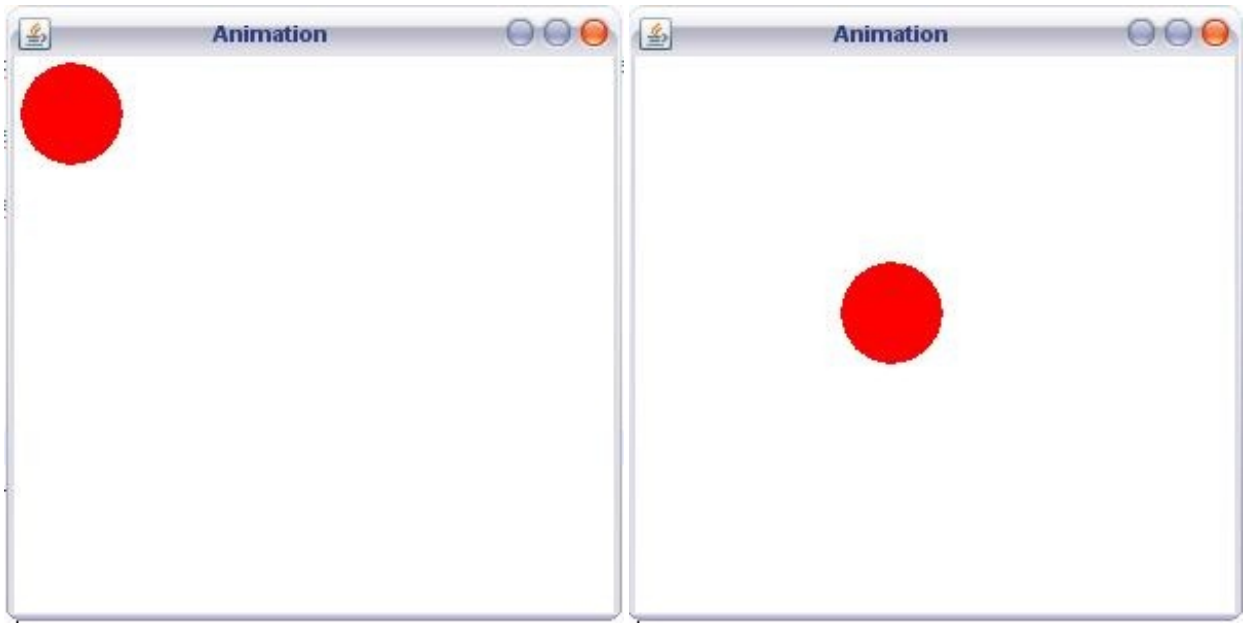
Il vous suffit de dessiner un rectangle, d'une quelconque couleur, prenant toute la surface disponible, avant de dessiner votre rond. Voici le code de notre classe **`Panneau`** mis à jour :

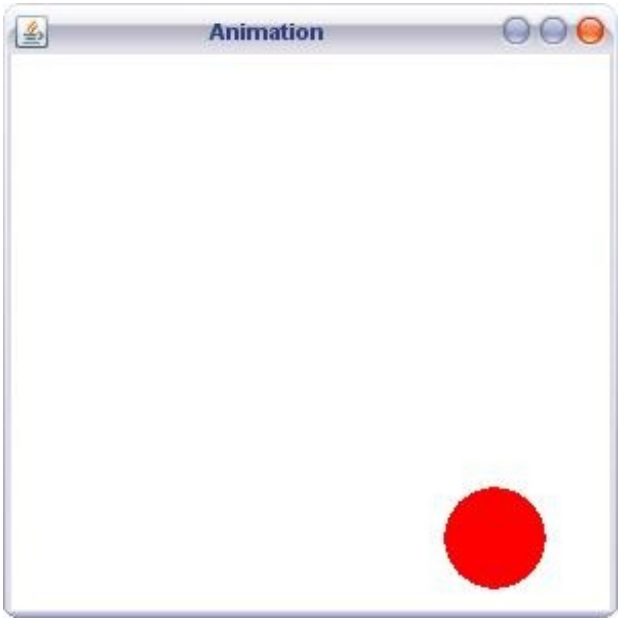
Code : Java

```
import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {

        private int posX = -50;
        private int posY = -50;

        public void paintComponent(Graphics g){
```

```
                //On décide d'une couleur de fond pour notre
    rectangle
                g.setColor(Color.white);
                //On dessine celui-ci afin qu'il prenne tout la
    surface
                g.fillRect(0, 0, this.getWidth(), this.getHeight());
                //On redéfinit une couleur pour notre rond
                g.setColor(Color.red);
                //On le dessine aux coordonnées souhaitées
                g.fillOval(posX, posY, 50, 50);
        }

        public int getPosX() {
                return posX;
        }

        public void setPosX(int posX) {
                this.posX = posX;
        }

        public int getPosY() {
                return posY;
        }

        public void setPosY(int posY) {
                this.posY = posY;
        }

}
```

Voici trois captures d'écran prises à différents moments de l'animation :





Je pense qu'il est temps d'améliorer encore notre animation... Est-ce que ça vous dirait que celle-ci continue tant que vous ne fermez pas votre fenêtre ?
Oui ? Alors continuons.
Retour en haut

Continue, ne t'arrêtes pas si vite !

Voici l'un des moments délicats que j'attendais... Si vous vous rappelez bien ce que je vous ai dit sur le fonctionnement des boucles, vous devez vous souvenir de mon avertissement sur les boucles infinies !

Eh bien ce que nous allons faire ici, c'est un exemple d'utilisation d'une boucle infinie...

Si vous y réfléchissez deux secondes, comment dire à une boucle de ne pas s'arrêter à moins qu'elle ne s'arrête jamais ?

Dans l'exemple que nous allons utiliser pour le moment, nous allons simplifier les choses, mais nous améliorerons cela lorsque nous commencerons à interagir avec notre application...

Il y a plusieurs manières de faire une boucle infinie : vous avez le choix entre une boucle **`for`**, **`while`** ou **`do...while`**. Regardez ces déclarations :

Code : Java

```
//Exemple avec un boucle while
while(true){
 // Ce code se répètera à l'infini car la condition est TOUJOURS
vraie !
}

//Exemple avec une boucle for
for(;;)
{
   // Idem que précédemment, ici il n'y a pas d'incrément => donc
la boucle ne se terminera jamais
}

//Exemple avec do...while
do{
   //Encore une boucle que ne se terminera pas !
}while(true);
```

Nous allons donc remplacer notre boucle finie par une boucle infinie dans la méthode **`go()`** de notre objet **`Fenetre`**. Ce qui nous donne :

Code : Java

```
private void go(){

                for(;;)
                {
                        int x = pan.getPosX(), y = pan.getPosY();
                        x++;
                        y++;
                        pan.setPosX(x);
                        pan.setPosY(y);
                        pan.repaint();
                        try {
                                Thread.sleep(10);
                        } catch (InterruptedException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                }

        }
```

Par contre, si vous avez exécuté notre nouvelle version, vous avez dû vous rendre compte qu'il reste un problème à gérer ! Eh oui. Votre rond ne se replace pas au départ lorsqu'il atteint l'autre côté de notre fenêtre.

Si vous ajoutez une instruction **System.out.println()** dans la méthode **paintComponent** inscrivant les coordonnées de notre rond, vous devez voir que celles-ci ne cessent de croître !

Le premier objectif est atteint mais il nous reste à gérer ce dernier problème.
Il faut donc réinitialiser les coordonnées de notre rond si celles-ci arrivent au bout de notre composant.
Voici donc notre méthode **go()** revue et corrigée :

Code : Java

```
	private void go(){

			for(;;)
			{
				int x = pan.getPosX(), y = pan.getPosY();
				x++;
				y++;
				pan.setPosX(x);
				pan.setPosY(y);
				pan.repaint();
				try {
					Thread.sleep(10);
				} catch (InterruptedException e) {
					// TODO Auto-generated catch block
					e.printStackTrace();
				}

				//Si nos coordonnées arrivent aux bords de
	notre composant
				//On réinitialise
				if(x == pan.getWidth() || y ==
	pan.getHeight())
				{
					pan.setPosX(-50);
					pan.setPosY(-50);
				}

			}

		}
```

Yes !
Le code fonctionne parfaitement. En tout cas, comme nous l'avions prévu !
Mais avant de passer au chapitre suivant, nous pouvons encore faire mieux...
On y va ?
Retour en haut

Attention aux bords, ne va pas te faire mal...

Maintenant, nous allons faire en sorte que notre rond puisse détecter les bords de notre **Panneau** et ricoche sur ceux-ci !
Vous devez vous imaginer un code monstrueux, et vous êtes très loin du compte...

Tout d'abord, jusqu'à maintenant, nous n'attachions aucune importance sur le bord que notre rond dépassait, ceci est terminé. Dorénavant, nous séparerons le dépassement des coordonnées **posX** et **posY** de notre **Panneau**.

Mais comment lui dire qu'il faut reculer ou avancer sur tel ou tel axe ?

Pour les instructions qui vont suivre, gardez en mémoire que les coordonnées de notre rond sont en fait les coordonnées du **coin supérieur gauche du carré entourant notre rond** ! !

Voilà la marche à suivre :

- si la coordonnée x de notre rond est inférieure à la largeur et qu'il avance, on continue d'avancer ;
- sinon, on recule.

Et nous allons faire de même pour la coordonnée y.

Comment savoir si on doit avancer ou reculer ? Avec un booléen.
Au tout début de notre application, deux booléens seront initialisés à **false** et si la coordonnée x est supérieure à la largeur du **Panneau**, alors on recule ; sinon, on avance, idem pour la coordonnée y.

Dans ce code, j'utilise deux variables de type **int** pour éviter de rappeler les méthodes **getPosX()** et **getPosY()**.

Voilà notre nouveau code de la méthode **go()** :

Code : Java

```
private void go(){

                //Les coordonnées de départ de notre rond
                int x = pan.getPosX(), y = pan.getPosY();
                //Le booléen pour savoir si on recule ou non sur
l'axe X
                boolean backX = false;
                //Le booléen pour savoir si on recule ou non sur
l'axe Y
                boolean backY = false;

                //Pour cet exemple, j'utilise une boucle while
                //Vous verrez qu'elle fonctionne très bien
                while(true){

                        //Si la coordonnée x est inférieure à 1, on
avance
                        if(x < 1)backX = false;
                        //Si la coordonnée x est supérieure à la
taille du Panneau
                        //moins la taille du rond, on recule
                        if(x > pan.getWidth()-50)backX = true;

                        //idem pour l'axe Y
                        if(y < 1)backY = false;
                        if(y > pan.getHeight()-50)backY = true;

                        //Si on avance, on incrémente la coordonnée
                        if(!backX)
                                pan.setPosX(++x);
                        //Sinon, on décrémente
```

```
                                else
                                        pan.setPosX(--x);

                                //Idem pour l'axe Y
                                if(!backY)
                                        pan.setPosY(++y);
                                else
                                        pan.setPosY(--y);

                                //On redessine notre Panneau
                                pan.repaint();

                                //Comme on dit : la pause s'impose ! Ici, 3
        millièmes de secondes
                                try {
                                        Thread.sleep(3);
                                } catch (InterruptedException e) {
                                        // TODO Auto-generated catch block
                                        e.printStackTrace();
                                }
                        }

                }
```

Exécutez votre application et vous devez voir que votre rond ricoche contre les bords de notre **Panneau**. Vous pouvez même étirer la fenêtre, la réduire et ça marche toujours !

On commence à faire des choses sympa, non ?
Un petit topo vous attend avant un petit QCM, et nous allons passer à la suite !

Retour en haut

Ce qu'il faut retenir

- À l'instanciation d'un composant, la méthode **`paintComponent`** est automatiquement appelée.
- Vous pouvez forcer un composant à se redessiner en invoquant la méthode **`repaint()`**.
- Pensez bien à ce que va donner votre composant après être redessiné.
- Pour éviter que votre animation bave, réinitialisez le fond de votre composant pour éviter ce phénomène...
- Vous verrez que tous les composants fonctionnent de la même manière.
- L'instruction **`Thread.sleep()`**, permet de faire une pause dans votre programme.
- Cette méthode prend un entier comme paramètre qui correspond à un temps exprimé en **millièmes de secondes**.
- Vous pouvez utiliser des boucles infinies pour faire des animations.

Retour en haut

Encore un chapitre rondement mené !
Maintenant, je pense que vous êtes paré pour : **Votre premier bouton**.

Retour en haut

Votre premier bouton

Voici l'un des moments que vous attendiez avec impatience !
Vous allez enfin pouvoir mettre un bouton dans votre application...

Mais ne chantez pas trop vite. Vous allez effectivement utiliser un bouton, mais voir aussi que les choses se compliquent dès que vous utilisez ce genre de composants... Et encore plus lorsque vous en utilisez plusieurs !

Toujours prêts ?
Go, alors.

Utiliser la classe JButton

Comme le titre l'indique, nous allons utiliser un objet **JButton**. Celui-ci se trouve aussi dans le package **javax.swing**.

Au cours de ce chapitre, nous allons un peu mettre de côté notre objet **Panneau** : en fait, notre projet précédent dans sa globalité sera mis à l'écart...

Créez un nouveau projet avec :

- une classe contenant une méthode `main` : appelons-la **Test** ;
- une classe héritée de `JFrame` (avec tout ce qu'on a déjà mis avant, sauf la méthode **go()**) : appelons-la **Fenetre**.

Je sais, j'aime donner dans l'originalité...

Dans votre classe **Fenetre**, nous allons créer une variable d'instance de type **JPanel** et une de type **JButton**.
Faites de votre **JPanel** le `contentPane` de votre **Fenetre**. Ce qui doit nous donner ceci :

Classe Fenetre

Code : Java

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;


public class Fenetre extends JFrame{

        private JPanel pan = new JPanel();
        private JButton bouton = new JButton();

        public Fenetre(){

                this.setTitle("Animation");
                this.setSize(300, 300);
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setLocationRelativeTo(null);
                this.setContentPane(pan);
                this.setVisible(true);

        }
}
```

Classe Test

Code : Java

```
public class Test {

        public static void main(String[] args){
                Fenetre fen = new Fenetre();
        }

}
```

Jusque-là, rien de bien nouveau mis à part l'instance de **JButton**, mais ce genre de choses ne doit plus vous épater, maintenant !

Définissons à présent un libellé à notre bouton et mettons-le sur... Sur... Ce qui nous sert de `contentPane` ! Notre **`JPanel`**, en l'occurrence !

Pour donner un libellé à un bouton, il existe deux méthodes, les voici :

Code : Java

```
//Méthode 1 : Instanciation avec le libellé
JButton bouton = new JButton("Mon bouton");
//Méthode 2 : Instanciation puis définition du libellé
JButton bouton2 = new JButton();
bouton2.setText("Mon deuxième bouton");
```

Personnellement, je préfère la première... Mais là n'est pas la question.
Il ne nous reste plus qu'à ajouter notre bouton sur notre contentPane et ceci, grâce à la méthode **`add()`** de l'objet **`JPanel`**.

Voici donc notre code :

Code : Java

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;


public class Fenetre extends JFrame{

        private JPanel pan = new JPanel();
        private JButton bouton = new JButton("Mon bouton");

        public Fenetre(){

                this.setTitle("Animation");
                this.setSize(300, 300);
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setLocationRelativeTo(null);

                //Ajout du bouton à notre contentPane
                pan.add(bouton);

                this.setContentPane(pan);
                this.setVisible(true);

        }
}
```

Et le résultat :

Alors ! Heureux ?
Je ne sais pas si vous avez remarqué mais... **votre bouton est centré sur votre conteneur !**
Ceci vous semble normal ? Sachez tout de même que votre `JPanel` fait de la mise en page par défaut.

Qu'est-ce que tu entends par là ?

En fait, en java, il existe des objets qui servent à agencer vos composants, et les objets `JPanel` en ont un par défaut !
Pour vous le prouver, je vais vous faire travailler sur le `contentPane` de votre `JFrame`.

Quoi ? On peut faire ça ?

Bien sûr, mais il était de bon ton de vous faire découvrir l'objet `Graphics` avant de vous jeter dans la fosse aux lions...
Vous allez voir que pour obtenir le même rendu que précédemment sur le `contentPane` de votre `JFrame`, nous allons être obligés d'utiliser un de ces fameux objets d'agencements !

Tout d'abord, pour utiliser le `contentPane` d'une `JFrame`, on doit appeler la méthode **`getContentPane()`**, qui retourne ce dernier, auquel nous ajoutons nos composants. Voici le nouveau code :

Code : Java

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;


public class Fenetre extends JFrame{

        private JPanel pan = new JPanel();
        private JButton bouton = new JButton("Mon bouton");

        public Fenetre(){

                this.setTitle("Bouton");
                this.setSize(300, 300);
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setLocationRelativeTo(null);

                //On ajoute le bouton au contentPane de la JFrame
                this.getContentPane().add(bouton);
```

```
                this.setVisible(true);

            }
        }
```

Voyez que le résultat n'est pas du tout le même :

Votre bouton est ÉNORME ! En fait, il prend toute la place disponible, et ceci parce que le `contentPane` de votre **`JFrame`** ne possède pas d'objet d'agencement...

Ces fameux objets, dont vous ne pourrez plus vous passer, s'appellent des ***layout managers***.

Faisons un petit tour d'horizon de ces objets et voyons comment ils fonctionnent !
Retour en haut

Les layout managers

Bon, vous allez voir qu'il existe plusieurs sortes de ***layout managers*** et que selon votre choix, il sera plus ou moins facile à utiliser... Je vais être méchant . Nous verrons le plus simple en dernier !

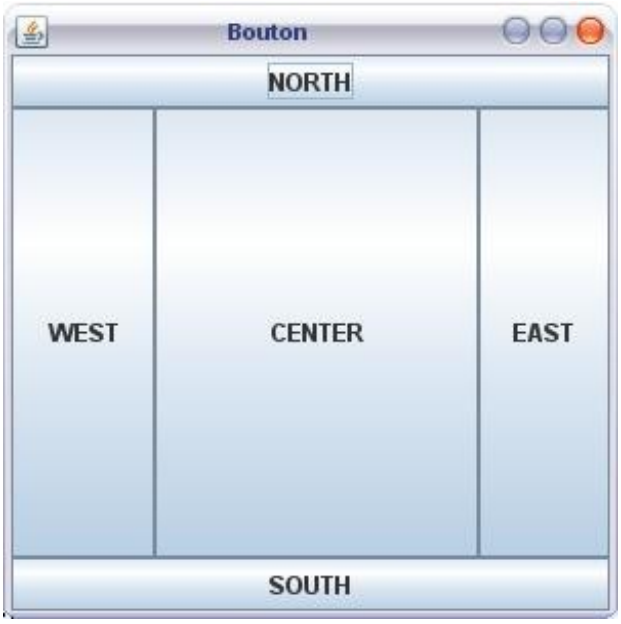
Vous devez aussi savoir que tous ces layout managers se trouvent dans la package : **`java.awt`**.

L'objet `BorderLayout`

Le premier que nous aborderons est le **`BorderLayout`**. Celui-ci est très pratique si vous voulez placer vos composants de façon simple par rapport à une position ***cardinale*** de votre conteneur. Si je parle de positionnement cardinal, c'est parce que vous allez utiliser les positions :

- NORTH
- SOUTH
- EAST
- WEST
- CENTER.

Mais je sais qu'un aperçu vaut mieux qu'un exposé sur le sujet. Alors voilà un exemple mettant en oeuvre un **`BorderLayout`**.



Cette fenêtre est composée de cinq **`JButton`** positionnés aux cinq endroits différents qu'offre un **`BorderLayout`**.

Voici le code de cette fenêtre :

Code : Java

```
import java.awt.BorderLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;


public class Fenetre extends JFrame{

        private JButton bouton = new JButton("BorderLayout.NORTH");

        public Fenetre(){

                this.setTitle("Bouton");
                this.setSize(300, 300);
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

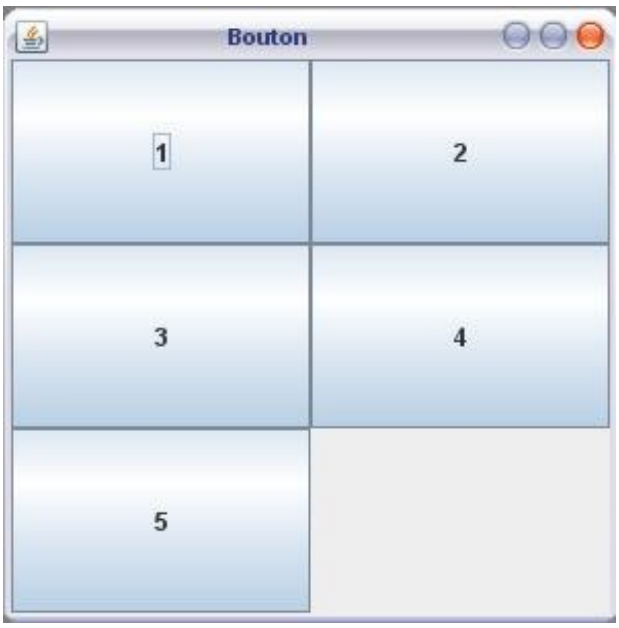
```
                this.setLocationRelativeTo(null);

                //On définit le layout à utiliser sur le
contentPane
                this.setLayout(new BorderLayout());

                //On ajoute le bouton au contentPane de la JFrame
                //Au centre
                this.getContentPane().add(new JButton("CENTER"),
BorderLayout.CENTER);
                //Au nord
                this.getContentPane().add(new JButton("NORTH"),
BorderLayout.NORTH);
                //Au sud
                this.getContentPane().add(new JButton("SOUTH"),
BorderLayout.SOUTH);
                //À l'ouest
                this.getContentPane().add(new JButton("WEST"),
BorderLayout.WEST);
                //À l'est
                this.getContentPane().add(new JButton("EAST"),
BorderLayout.EAST);


                this.setVisible(true);

        }
}
```

Ce n'est pas très difficile à comprendre. Vous définissez le *layout* à utiliser avec la méthode **`setLayout()`** de l'objet **`JFrame`** ; ensuite, pour chaque composant que vous souhaitez positionner avec **`add()`**, vous utilisez un attribut **`static`** de la classe **`BorderLayout`** en deuxième paramètre.
La liste des ces paramètres est celle citée plus haut...

Avec l'objet **`BorderLayout`**, vos composants sont soumis à des contraintes. Pour une position **`NORTH`** ou **`SOUTH`**, votre composant **prendra la taille nécessaire en hauteur mais prendra toute la largeur** ! Pour **`WEST`** et **`EAST`**, celui-ci prendra **la largeur nécessaire mais toute la hauteur** ! Et bien sûr pour **`CENTER`**, **tout l'espace est utilisé** !

L'objet `GridLayout`

Celui-ci permet d'ajouter des composants suivant une grille définie par un nombre de lignes et de colonnes. Les éléments sont disposés depuis la case située en haut à gauche. Dès qu'une ligne est remplie, on passe à la suivante.
Si nous définissons une grille de 3 lignes et de 2 colonnes, voici ce que nous aurions :



Voici le code de cet exemple :

Code : Java

```
import java.awt.GridLayout;

import javax.swing.JButton;
import javax.swing.JFrame;


public class Fenetre extends JFrame{

        private JButton bouton = new JButton("BorderLayout.NORTH");

        public Fenetre(){

                this.setTitle("Bouton");
                this.setSize(300, 300);
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setLocationRelativeTo(null);

                //On définit le layout à utiliser sur le
contentPane
                //3 lignes sur 2 colonnes
                this.setLayout(new GridLayout(3, 2));

                //On ajoute le bouton au contentPane de la JFrame
                this.getContentPane().add(new JButton("1"));
                this.getContentPane().add(new JButton("2"));
                this.getContentPane().add(new JButton("3"));
                this.getContentPane().add(new JButton("4"));
                this.getContentPane().add(new JButton("5"));

                this.setVisible(true);

        }
}
```

Ce code ne diffère pas beaucoup du précédent. Les seules différences résident dans le fait que nous utilisons un autre layout manager et qu'il n'y a pas besoin de définir le positionnement lors de l'ajout du composant avec la méthode **`add()`**.

Vous devez aussi savoir que vous pouvez définir le nombre de lignes et le nombre de colonnes avec des méthodes de l'objet :

Code : Java

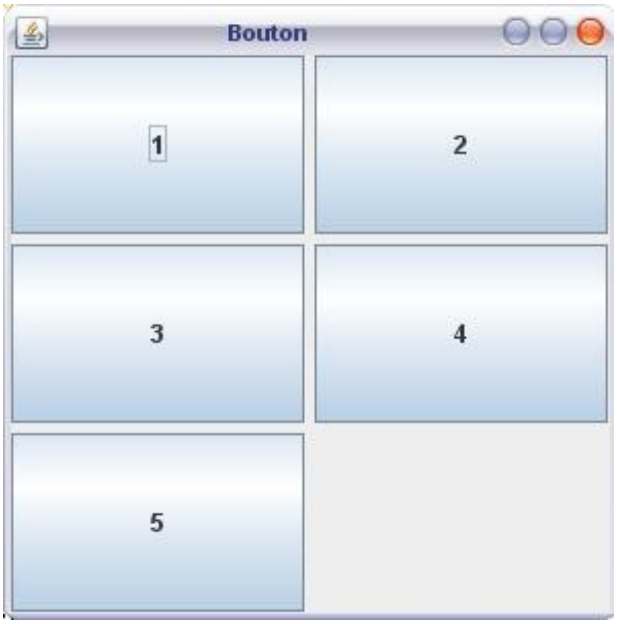
```
GridLayout gl = new GridLayout();
gl.setColumns(2);
gl.setRows(3);
this.setLayout(gl);

// ou en abrégé : GridLayout gl = new GridLayout(3, 2);
```

Vous pouvez aussi spécifier des espaces entre les colonnes et entre les lignes.

Code : Java

```
GridLayout gl = new GridLayout(3, 2);
gl.setHgap(5); // 5 pixels d'espace entre les colonnes (H comme
Horizontal)
gl.setVgap(5); // 5 pixels d'espace entre les lignes (V comme
Vertical)

//ou en abrégé : GridLayout gl = new GridLayout(3, 2, 5, 5);
```

Ce qui donnerait :



L'objet FlowLayout

Celui-ci est certainement le plus facile à utiliser ! Il ne fait que centrer les composants dans le conteneur. Regardez plutôt :

Ah ! On dirait bien que nous venons de trouver le layout manager défini par défaut dans les objets `JPanel`.
Si vous mettez plusieurs composants avec ce gestionnaire, dès que la place est devenu trop étroite, il passe à la ligne inférieure.
Voyez plutôt :

Il existe encore deux types de layout managers :

- le `CardLayout`
- le `GridBagLayout`.

Je suis dans l'incapacité de vous en parler car je ne les ai encore jamais utilisés... Sachez seulement que le dernier cité est le plus précis, mais aussi le plus compliqué à utiliser... Si vous êtes intéressés par ces deux gestionnaires, vous pouvez trouver des renseignements ici pour le `CardLayout` et là pour le `GridBagLayout`.

Il faut que vous sachiez que les IHM ne sont en fait qu'une imbrication de composants les uns dans les autres, positionnés avec des layout managers. Vous allez voir tout de suite de quoi je veux parler...
Retour en haut

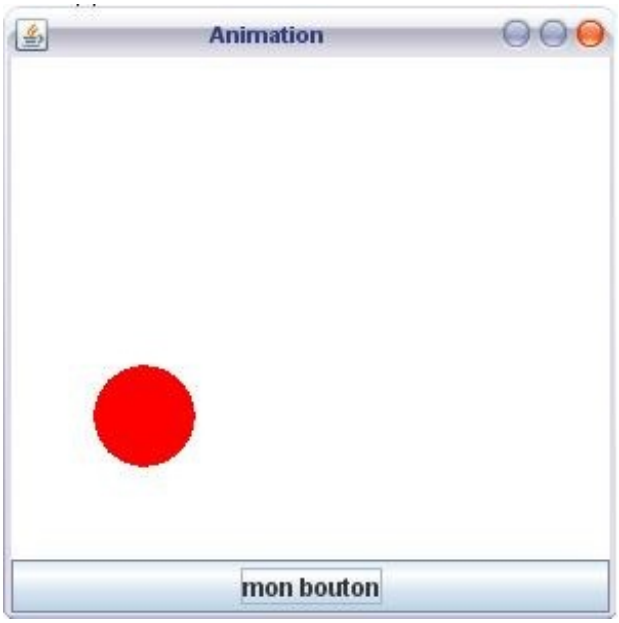
Continuons dans notre lancée

Vous avez vu comment utiliser les layout managers ; nous allons donc continuer à jouer avec nos composants.
Vous savez :

- créer une fenêtre
- faire un conteneur personnalisé
- placer celui-ci dans votre fenêtre
- créer un bouton et le placer sur votre fenêtre.

Nous allons maintenant utiliser notre conteneur personnalisé et un bouton.
Vous pouvez revenir dans votre projet contenant notre animation créée dans les chapitres précédents.

Le but est de mettre notre animation au centre de notre fenêtre et un bouton en bas de celle-ci. Comme ceci :



Je vous laisse réfléchir quelques minutes, vous avez tous les outils pour y arriver !

Secret (cliquez pour afficher)

Voilà le nouveau code de notre classe **Fenetre** :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;


public class Fenetre extends JFrame{

        private Panneau pan = new Panneau();
        private JButton bouton = new JButton("mon bouton");
        private JPanel container = new JPanel();

        public Fenetre(){

                this.setTitle("Animation");
                this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
                this.setLocationRelativeTo(null);

                container.setBackground(Color.white);
                container.setLayout(new BorderLayout());
                container.add(pan, BorderLayout.CENTER);
                container.add(bouton, BorderLayout.SOUTH);

                this.setContentPane(container);
                this.setVisible(true);

                go();
        }


        private void go(){

                //Les coordonnées de départ de notre rond
                int x = pan.getPosX(), y = pan.getPosY();
                //Le booléen pour savoir si on recule ou non sur
l'axe X
                boolean backX = false;
                //Le booléen pour savoir si on recule ou non sur
l'axe Y
                boolean backY = false;

                //Pour cet exemple, j'utilise une boucle while
                //Vous verrez qu'elle fonctionne très bien
                while(true){

                        //Si la coordonnée x est inférieure à 1,
on avance
                        if(x < 1)backX = false;
                        //Si la coordonnée x est supérieure à la
taille du panneau
                        //moins la taille du rond, on avance
                        if(x > pan.getWidth()-50)backX = true;

                        //idem pour l'axe Y
                        if(y < 1)backY = false;
                        if(y > pan.getHeight()-50)backY = true;

                        //Si on avance, on incrémente la
coordonnée
                        if(!backX)
                                pan.setPosX(++x);
                        //Sinon, on décrémente
                        else
                                pan.setPosX(--x);

                        //Idem pour l'axe Y
                        if(!backY)
                                pan.setPosY(++y);
                        else
                                pan.setPosY(--y);

                        //On redessine notre panneau
                        pan.repaint();

                        //Comme on dit : la pause s'impose ! Ici,
3 centièmes de secondes
                        try {
                                Thread.sleep(3);
                        } catch (InterruptedException e) {
                                // TODO Auto-generated catch
block
                                e.printStackTrace();
                        }
                }

        }
```

```
    }
```

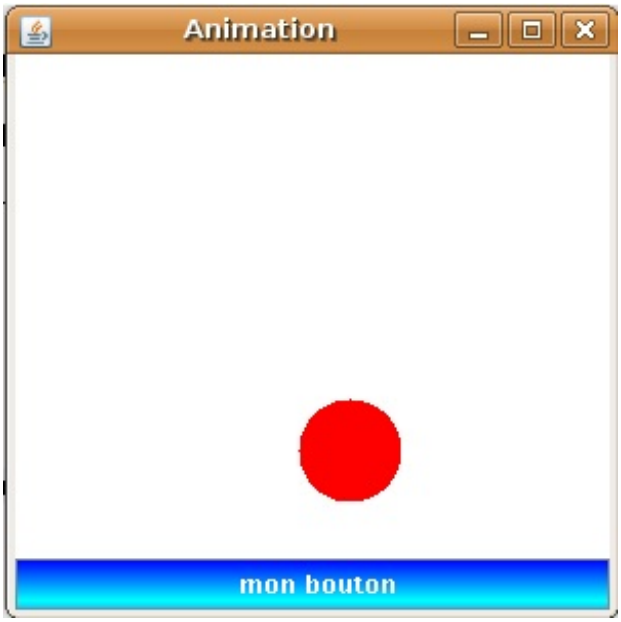
Assez facile... N'est-ce pas ?
Avant de voir comment les boutons interagissent avec l'application, je vous propose de voir comment personnaliser ceux-ci !

Retour en haut

Une classe Bouton personnalisée

Tout comme dans le deuxième chapitre, nous allons faire une classe : appelons-la **Bouton**, héritée (dans notre cas, nous allons faire hériter notre classe de **javax.swing.JButton**), et nous allons redéfinir la méthode **paintComponent**. Vous devriez y arriver tous seuls.

Pour cet exemple j'ai obtenu ceci :



Voici la classe **Bouton** de cette application :

Code : Java

```
import java.awt.Color;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;

import javax.swing.JButton;


public class Bouton extends JButton {

        private String name;

        public Bouton(String str){
                super(str);
                this.name = str;
        }

        public void paintComponent(Graphics g){

                Graphics2D g2d = (Graphics2D)g;

                GradientPaint gp = new GradientPaint(0, 0,
Color.blue, 0, 20, Color.cyan, true);
                g2d.setPaint(gp);
                g2d.fillRect(0, 0, this.getWidth(),
this.getHeight());

                g2d.setColor(Color.white);
                g2d.drawString(this.name, this.getWidth() / 2 -
(this.getWidth() / 2 /4), (this.getHeight() / 2) + 5);

        }
```

```
    }
```

J'ai aussi fait un bouton personnalisé avec une image de fond (un png...) que voici :

Et voilà le résultat :

C'est sympa aussi !
J'ai appliqué l'image sur l'intégralité du fond comme je l'ai montré lorsque nous nous amusions avec notre **Panneau**. Voici le code de la classe **Bouton** :

Code : Java

```
import java.awt.Color;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;
import javax.swing.JButton;


public class Bouton extends JButton{

        private String name;
    private Image img;

    public Bouton(String str){
            super(str);
            this.name = str;

            try {
```

```
                        img = ImageIO.read(new File("fondBouton.png"));
                    } catch (IOException e) {
                        e.printStackTrace();
                    }


        }

        public void paintComponent(Graphics g){

                Graphics2D g2d = (Graphics2D)g;

                GradientPaint gp = new GradientPaint(0, 0, Color.blue,
    0, 20, Color.cyan, true);
                g2d.setPaint(gp);
               // g2d.fillRect(0, 0, this.getWidth(),
    this.getHeight());
              g2d.drawImage(img, 0, 0, this.getWidth(),
    this.getHeight(), this);

                g2d.setColor(Color.black);
                g2d.drawString(this.name, this.getWidth() / 2 -
    (this.getWidth() / 2 /4), (this.getHeight() / 2) + 5);

        }

    }
```

Bien sûr, ladite image est à la racine de mon projet !

Rien de bien compliqué jusqu'à maintenant... Et c'est à partir de là que les choses vont devenir intéressantes !
Que diriez-vous si je vous proposais de pouvoir changer l'aspect de votre objet au passage de la souris, lorsque vous cliquez dessus, et même lorsque vous relâchez le clic ?

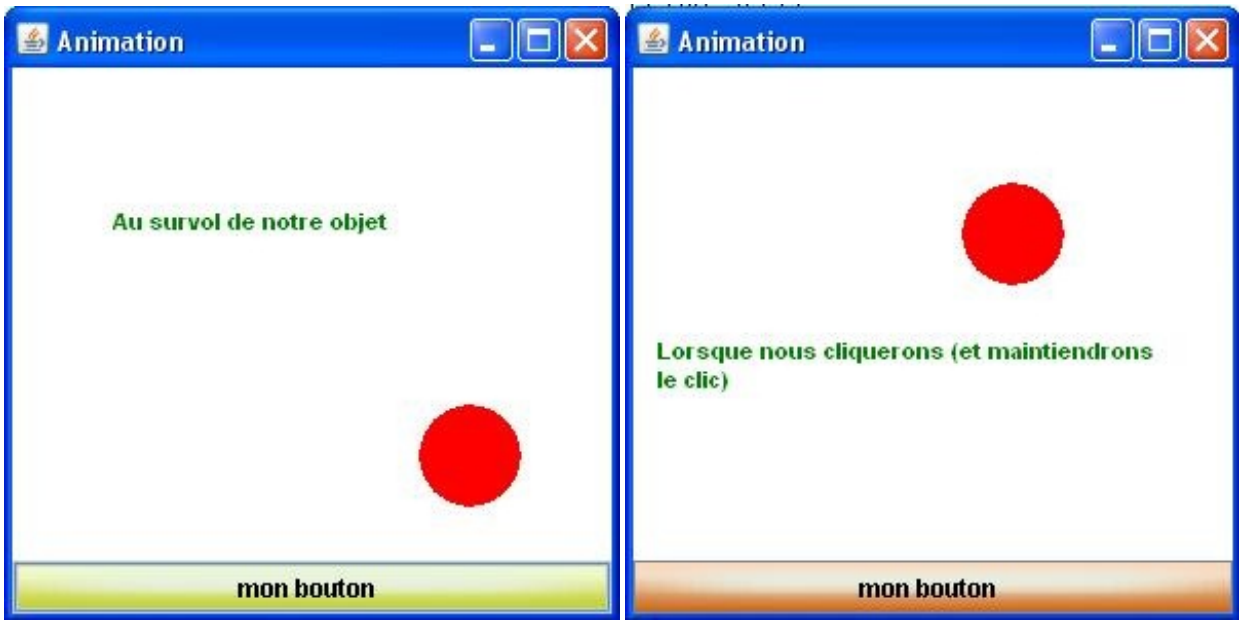
On peut faire ça ?

Bien entendu !
Il existe des interfaces à implémenter qui permettent de gérer toutes sortes d'événements sur votre IHM.
Le principe de fonctionnement est un peu déroutant au premier abord, mais il est assez simple lorsqu'on a un peu pratiqué.
N'attendons pas plus et voyons ça de plus près !

Retour en haut

Interaction avec la souris : l'interface MouseListener

Avant de nous lancer dans l'implémentation de cette dernière, nous allons voir ce que nous allons obtenir à la fin :



Vous avez vu ce que nous allons obtenir, mais vous allez tout de même passer par un peu de théorie avant d'arriver à ce résultat. Je ne me suis pas encore attardé sur le sujet (et je ne le ferai toujours pas), mais, pour arriver à détecter les événements qui surviennent à votre composant, Java utilise ce qu'on appelle le **design pattern Observer**.

Je ne vous l'expliquerai pas dans le détail tout de suite, une partie concernant les *design patterns* sera rédigée...
En fait, un **dp** est un modèle de conception, une idée directrice afin d'avoir des programmes stables, réutilisables à souhait et paramétrables au possible !

Le design pattern **Observer** consiste en un modèle qui permet à des objets de se tenir au courant automatiquement. Ceci se fait par le biais d'interfaces que les objets devant se tenir au courant doivent implémenter.
Pour le moment, retenez qu'un objet observable implémente une interface communément appelée **Observable** et qu'un observateur va, lui, implémenter une interface communément appelée **Observer** (ceci dans le cas où nous utilisons le pattern Observer fourni avec Java ; oui, ces interfaces existent...). Le principe est simple :

- l'objet observable va avoir un ou plusieurs objets observateurs
- lorsqu'une donnée change dans un objet observable, il met tous ses objets observateurs abonnés à sa "*news letter*" au courant !

Ce principe, adapté aux composants **swing**, permet aux objets composants d'être écoutés et donc de pouvoir faire réagir votre application en conséquence !

Ne nous attardons pas sur le sujet, nous y reviendrons.
Maintenant que vous avez une très vague idée de la façon dont Java gère les événements, nous allons commencer à gérer les passages de notre souris sur notre objet !

Comme vous l'avez sûrement deviné, vous allez devoir implémenter l'interface **MouseListener** dans votre classe **Bouton**. Utilisez l'astuce d'Eclipse vue lors des implémentations d'interfaces, afin de générer automatiquement les méthodes à implémenter.

Nous aurons aussi à dire à notre classe **Bouton** qu'elle va devoir tenir quelqu'un au courant de ses changements d'état par rapport à la souris. Ce quelqu'un n'est autre... qu'elle-même !
Eh oui... Notre classe va s'écouter elle-même, donc ; dès que notre objet observable, notre bouton, va avoir des informations concernant les actions de la souris, il va dire à l'objet qui l'observe, lui-même, ce qu'il doit faire !

Ceci se fait grâce à la méthode **addMouseListener(MouseListener obj)** qui prend un objet **MouseListener** en paramètre ; ici, elle prendra **this**. **Rappelez-vous que vous pouvez utiliser le type d'une interface comme super-type : ici, notre classe implémente l'interface MouseListener , nous pouvons donc utiliser cet objet comme référence de cette interface !**

Voici notre classe Bouton à présent :

Code : Java

```
import java.awt.Color;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;
import javax.swing.JButton;


public class Bouton extends JButton implements MouseListener{

    private String name;
    private Image img;

    public Bouton(String str){
        super(str);
        this.name = str;

        try {
            img = ImageIO.read(new
File("fondBouton.png"));
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        //Avec cette instruction, notre objet va
s'écouter lui-même
        //Dès qu'un événement de la souris sera
intercepté, il sera au courant !
        this.addMouseListener(this);
    }

    public void paintComponent(Graphics g){

        Graphics2D g2d = (Graphics2D)g;

        GradientPaint gp = new GradientPaint(0, 0, Color.blue,
0, 20, Color.cyan, true);
        g2d.setPaint(gp);
        // g2d.fillRect(0, 0, this.getWidth(),
this.getHeight());
        g2d.drawImage(img, 0, 0, this.getWidth(),
this.getHeight(), this);

        g2d.setColor(Color.black);
        g2d.drawString(this.name, this.getWidth() / 2 -
(this.getWidth() / 2 /4), (this.getHeight() / 2) + 5);

    }

    /**
* Méthode appelée lors du clic de souris
*/
    public void mouseClicked(MouseEvent event) {

    }

    /**
* Méthode appelée lors du survol de la souris
*/
```

```
        public void mouseEntered(MouseEvent event) {

        }

        /**
* Méthode appelée lorsque la souris sort de la zone du bouton
*/
        public void mouseExited(MouseEvent event) {

        }

        /**
* Méthode appelée lorsque l'on presse le clic gauche de la souris
*/
        public void mousePressed(MouseEvent event) {

        }

        /**
* Méthode appelée lorsque l'on relâche le clic de souris
*/
        public void mouseReleased(MouseEvent event) {

        }
}
```

C'est par le biais de ces différentes méthodes que nous allons gérer les différentes images à dessiner dans notre objet ! Mais rappelez-vous que : **Même si vous n'utilisez pas toutes les méthodes d'une interface, vous DEVEZ tout de même mettre le squelette des méthodes non utilisées (avec les accolades), et ceci est aussi valable pour les classes abstraites**.

Dans notre cas, la méthode **repaint()** est appelée de façon tacite. **Lorsqu'un événement est déclenché, notre objet se redessine automatiquement !** Comme lorsque vous redimensionnez votre fenêtre dans les premiers chapitres...

Avec cette information, nous n'avons plus qu'à changer notre image selon la méthode invoquée :

- notre objet aura une teinte jaune lors du survol de la souris
- une teinte orangée lorsque l'on pressera le clic gauche
- reviendra à la normale si on relâche le clic.

Pour ce faire, voici les fichiers .png dont je me suis servi (mais rien ne vous empêche de les faire vous-mêmes).

Je vous rappelle que dans le code qui suit, les fichiers images sont mis à la racine du projet ! !

Maintenant, voici le code de notre classe **Bouton** personnalisée :

Code : Java

```
import java.awt.Color;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;
import javax.swing.JButton;
```

```
public class Bouton extends JButton implements MouseListener{

	private String name;
    private Image img;

    public Bouton(String str){
            super(str);
            this.name = str;

            try {
                                img = ImageIO.read(new
File("fondBouton.png"));
                        } catch (IOException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }

                        this.addMouseListener(this);
    }

    public void paintComponent(Graphics g){

            Graphics2D g2d = (Graphics2D)g;

            GradientPaint gp = new GradientPaint(0, 0, Color.blue,
0, 20, Color.cyan, true);
            g2d.setPaint(gp);
           // g2d.fillRect(0, 0, this.getWidth(),
this.getHeight());
           g2d.drawImage(img, 0, 0, this.getWidth(),
this.getHeight(), this);

            g2d.setColor(Color.black);
            g2d.drawString(this.name, this.getWidth() / 2 -
(this.getWidth() / 2 /4), (this.getHeight() / 2) + 5);

    }

        @Override
        public void mouseClicked(MouseEvent event) {
                //Pas utile d'utiliser cette méthode ici
        }

        @Override
        public void mouseEntered(MouseEvent event) {

                //Nous changeons le fond en jaune pour notre image
lors du survol
                //avec le fichier fondBoutonHover.png
                try {
                        img = ImageIO.read(new
File("fondBoutonHover.png"));
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }

        @Override
        public void mouseExited(MouseEvent event) {

                //Nous changeons le fond en vert pour notre image
lorsqu'on quitte le bouton
                //avec le fichier fondBouton.png
                try {
                        img = ImageIO.read(new
File("fondBouton.png"));
```

```
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }

        @Override
        public void mousePressed(MouseEvent event) {

                //Nous changeons le fond en orangé pour notre image
lors du clic gauche
                //avec le fichier fondBoutonClic.png
                try {
                        img = ImageIO.read(new
File("fondBoutonClic.png"));
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }

        @Override
        public void mouseReleased(MouseEvent event) {

                //Nous changeons le fond en orangé pour notre image

                //lorsqu'on relâche le clic
                //avec le fichier fondBoutonHover.png
                try {
                        img = ImageIO.read(new
File("fondBoutonHover.png"));
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
}
```

Et voilà le travail ! Il va de soi que si vous avez fait un clic-droit / Enregistrer-sous sur mes images, elles ne doivent pas avoir le même nom que sur mon morceau de code... Vous **DEVEZ** mettre le nom que vous leur avez donné... Je sais ça va de soi... Mais on ne sait jamais !

Et maintenant, vous avez un bouton personnalisé qui réagit aux passages de souris !

Mais un bémol se profile à l'horizon...

Lequel ? L'objet marche très bien !

Je sais qu'il va y avoir des p'tits malins qui vont cliquer sur le bouton et relâcher le clic en dehors du bouton !
Dans ce cas, vous devez voir que le fond du bouton est jaune, vu que c'est ce que nous avons demandé de faire à notre méthode **mouseReleased** ! Pour palier à ce problème, nous allons vérifier que, lorsque le clic est relâché, la souris est toujours sur le bouton.

Comment tu vas réussir à faire ça ? J'ai eu beau regarder dans les méthodes proposées par notre objet, mais il n'y a rien qui nous permette de faire ça !

Je sais... Mais vous n'avez pas regardé au bon endroit !
Maintenant que nous avons implémenté l'interface **MouseListener**, il y a un autre objet que nous n'avons pas encore utilisé... Vous ne le voyez pas ? C'est le paramètre présent dans toutes les méthodes de cette interface ! Oui, c'est **MouseEvent**.

Grâce à cet objet, nous pouvons avoir beaucoup de renseignements sur les événements. Nous ne détaillerons pas tout ici mais

vous verrez quelques utilités de ces types d'objets tout au long de cette partie !
Dans notre cas, nous pouvons récupérer la position X et Y de notre souris par rapport à notre **Bouton**, tout ceci avec les méthodes **getX()** et **getY()**. Donc, si nous relâchons notre clic de souris en dehors de la zone où se trouve notre objet, la valeur retournée par l'une des méthodes **getX()** ou **getY()** sera négative ou supérieure aux dimensions du bouton !

Voici enfin le code final de notre classe **Bouton** :

Code : Java

```
import java.awt.Color;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;
import javax.swing.JButton;


public class Bouton extends JButton implements MouseListener{

    private String name;
    private Image img;

    public Bouton(String str){
        super(str);
        this.name = str;

        try {
            img = ImageIO.read(new
File("fondBouton.png"));
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        this.addMouseListener(this);
    }

    public void paintComponent(Graphics g){

        Graphics2D g2d = (Graphics2D)g;

        GradientPaint gp = new GradientPaint(0, 0, Color.blue,
0, 20, Color.cyan, true);
        g2d.setPaint(gp);
        // g2d.fillRect(0, 0, this.getWidth(),
this.getHeight());
        g2d.drawImage(img, 0, 0, this.getWidth(),
this.getHeight(), this);

        g2d.setColor(Color.black);
        g2d.drawString(this.name, this.getWidth() / 2 -
(this.getWidth() / 2 /4), (this.getHeight() / 2) + 5);

    }

    @Override
    public void mouseClicked(MouseEvent event) {
        //Pas utile d'utiliser cette méthode ici
    }

    @Override
    public void mouseEntered(MouseEvent event) {
```

```
                //Nous changeons le fond en jaune pour notre image
lors du survol
                //avec le fichier fondBoutonHover.png
                try {
                        img = ImageIO.read(new
File("fondBoutonHover.png"));
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }

        @Override
        public void mouseExited(MouseEvent event) {

                //Nous changeons le fond en vert pour notre image
lorsqu'on quitte le bouton
                //avec le fichier fondBouton.png
                try {
                        img = ImageIO.read(new
File("fondBouton.png"));
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }

        @Override
        public void mousePressed(MouseEvent event) {

                //Nous changeons le fond en orangé pour notre image
lors du clic gauche
                //avec le fichier fondBoutonClic.png
                try {
                        img = ImageIO.read(new
File("fondBoutonClic.png"));
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }

        @Override
        public void mouseReleased(MouseEvent event) {

                //Nous changeons le fond en orangé pour notre image

                //lorsqu'on relâche le clic
                //avec le fichier fondBoutonHover.png

                //Si on est à l'extérieur de l'objet, on dessine le
fond par défaut
                if(event.getY() < 0 || event.getX() < 0 ||
event.getY() > this.getHeight() || event.getX() > this.getWidth())
                {
                        try {
                                img = ImageIO.read(new
File("fondBouton.png"));
                        } catch (IOException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                }
                //Sinon on met le fond jaune, la souris est encore
dessus...
                else
                {
```

```
                                try {
                                        img = ImageIO.read(new
        File("fondBoutonHover.png"));
                                } catch (IOException e) {
                                        // TODO Auto-generated catch block
                                        e.printStackTrace();
                                }
                        }
                }

        }
```

Vous allez voir tout au long des chapitres qui suivent qu'il y a différentes interfaces pour les différentes actions possibles sur une IHM ! Sachez seulement qu'il y a une convention aussi pour ces interfaces.
Leur nom commence par le type d'action ensuite suivi du mot **Listener**. Nous avons vu ici les actions de la souris, et voyez le nom de l'interface : **MouseListener**

Voilà ! Ce chapitre est clos !

Eh ! Attends... Notre bouton ne sais toujours rien faire !

C'est justement le sujet du prochain chapitre. Passez faire un tour sur le topo de cette partie et si vous avez un score raisonnable au QCM, vous pourrez y aller...

Retour en haut

Ce qu'il faut retenir

- Un bouton s'utilise avec la classe **`JButton`** présente dans le package **`javax.swing`**.
- Pour ajouter un bouton à une fenêtre, vous devez utiliser la méthode `add()` de son `contentPane`.
- Il existe des objets afin de pouvoir positionner nos composants sur un `contentPane` ou un conteneur : **les layout managers**.
- Les layout managers se trouvent dans le package **`java.awt`**.
- Le layout manager par défaut d'un objet **`JPanel`** est le **`FlowLayout`**.
- Outre le **`FlowLayout`**, vous avez aussi le **`BorderLayout`**, le **`GridLayout`**, le **`CardLayout`** et le **`GridBagLayout`**. Il en existe sûrement d'autres sur le net... mais je ne les connais pas...
- On définit un layout sur un conteneur grâce à la méthode **`setLayout()`**.
- Vous pouvez **interagir sur un composant** avec votre souris en implémentant l'interface **`MouseListener`**.
- Lorsque vous implémentez une interface **`<...>Listener`**, vous dites à votre classe qu'elle doit se préparer à observer des événements du type de l'interface... **Vous devez donc spécifier QUI doit observer et QUE doit-elle observer**, avec les méthodes du type **`add<...>Listener(<...>Listener)`**.

Retour en haut

Toujours motivés pour continuer ?
Je ne vous ai pas trop fait peur ?

Très bien, alors voyons : **Interaction bouton(s) - application**.

Retour en haut

Interaction bouton(s) - application

Nous y voilà !
Dans ce chapitre, votre objet `Bouton` pourra enfin communiquer avec votre application !
Je pense tout de même que le chapitre précédent à dû vous plaire...

Nous allons voir comment faire, mais aussi qu'il y a plusieurs façons de faire...
Ensuite, il ne tiendra qu'à vous de bien choisir...

En avant, moussaillons.

Déclencher une action : l'interface ActionListener

Tout est dans le titre !
Afin de gérer les différentes actions à effectuer selon le bouton sur lequel on clique, nous allons utiliser l'interface **ActionListener**.

Cependant, nous n'allons pas implémenter cette dernière dans notre classe **Bouton**, mais dans notre classe **Fenetre**... Le but étant de faire en sorte que lorsque nous cliquons sur notre bouton, il se passe quelque chose dans notre application comme changer un état, une variable, faire une incrémentation... Enfin n'importe quelle action !

Comme je vous l'ai expliqué dans la partie précédente, lorsque nous faisons **addMouseListener**, nous prévenons l'objet observé qu'un objet doit être mis au courant ! Ici, nous voulons que ce soit notre application, notre **Fenetre**, qui écoute notre **Bouton**, le but final étant de pouvoir lancer ou d'arrêter l'animation de notre **Panneau**.

Avant d'en arriver là, nous allons faire plus simple. Nous allons voir dans un premier temps l'implémentation de l'interface **ActionListener**. Afin de vous montrer toute la puissance de cette interface, nous allons utiliser un nouvel objet présent dans le package **javax.swing** : le **JLabel**.
Cet objet est en fait comme une étiquette, il est spécialisé dans l'affichage de texte ou d'image... Il est donc parfait pour notre premier exemple !

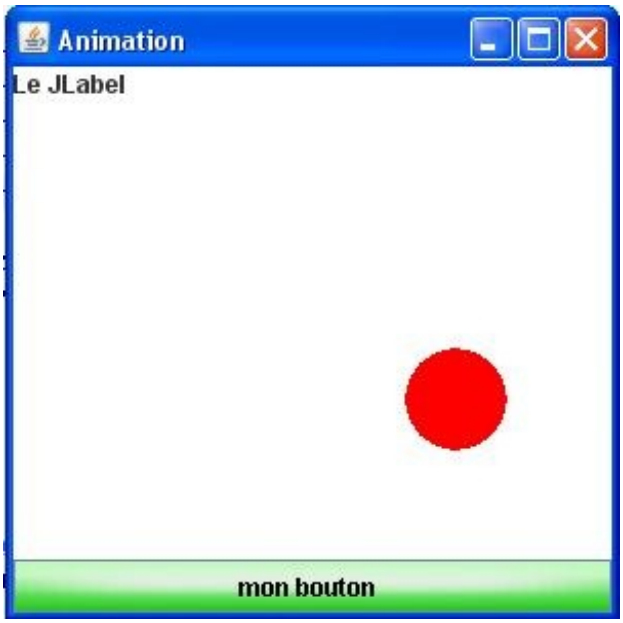
Pour l'instanciation ou l'initialisation, il fonctionne un peu comme le **JButton**, voyez plutôt :

Code : Java

```
JLabel label1 = new JLabel();
label1.setText("mon premier JLabel");
//Ou encore
JLabel label2 = new JLabel("Mon deuxième JLabel");
```

Créez une variable d'instance de type **JLabel** - appelons-la **label** - initialisez-la avec le texte qui vous plaît, puis ajoutez-la avec votre `contentPane` en **BorderLayout.NORTH**.

Voici le résultat :



Et le code :

Code : Java

```
public class Fenetre extends JFrame {
```

```
        private Panneau pan = new Panneau();
        private Bouton bouton = new Bouton("mon bouton");
        private JPanel container = new JPanel();
        private JLabel label = new JLabel("Le JLabel");

        public Fenetre(){

                this.setTitle("Animation");
                this.setSize(300, 300);
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setLocationRelativeTo(null);

                container.setBackground(Color.white);
                container.setLayout(new BorderLayout());
                container.add(pan, BorderLayout.CENTER);
                container.add(bouton, BorderLayout.SOUTH);
                container.add(label, BorderLayout.NORTH);

                this.setContentPane(container);
                this.setVisible(true);

                go();
        }

        //...
}
```

Vous pouvez voir le texte en haut à gauche... L'alignement par défaut de cet objet est à gauche mais vous pouvez changer quelques paramètres, comme :

- l'alignement
- la police à utiliser
- la couleur de police
- ...

Voilà un code qui met tout ceci en pratique, et son aperçu.

Code : Java

```
    public Fenetre(){

                this.setTitle("Animation");
                this.setSize(300, 300);
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setLocationRelativeTo(null);

                container.setBackground(Color.white);
                container.setLayout(new BorderLayout());
                container.add(pan, BorderLayout.CENTER);
                container.add(bouton, BorderLayout.SOUTH);

                //Définition d'une police d'écriture
                Font police = new Font("Tahoma", Font.BOLD, 16 );
                //On applique celle-ci aux JLabel
                label.setFont(police);
                //On change la couleur de police
                label.setForeground(Color.blue);
                //Et on change l'alignement du texte grâce aux
attributs static de la classe JLabel
                label.setHorizontalAlignment(JLabel.CENTER);

                container.add(label, BorderLayout.NORTH);

                this.setContentPane(container);
```

```
                this.setVisible(true);

                go();
        }
```

Aperçu :

Maintenant que notre étiquette est exactement comme nous le voulons, nous allons pouvoir implémenter l'interface **ActionListener**.
Lorsque vous avez implémenté les méthodes de l'interface, vous vous apercevez que celle-ci n'en contient qu'une seule !

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class Fenetre extends JFrame implements ActionListener{

    private Panneau pan = new Panneau();
    private Bouton bouton = new Bouton("mon bouton");
    private JPanel container = new JPanel();
    private JLabel label = new JLabel("Le JLabel");

    public Fenetre(){

            this.setTitle("Animation");
            this.setSize(300, 300);
            this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            this.setLocationRelativeTo(null);

            container.setBackground(Color.white);
            container.setLayout(new BorderLayout());
            container.add(pan, BorderLayout.CENTER);
            container.add(bouton, BorderLayout.SOUTH);
```

```
                //Définition d'une police d'écriture
                Font police = new Font("Tahoma", Font.BOLD, 16 );
                //On applique celle-ci aux JLabel
                label.setFont(police);
                //On change la couleur de police
                label.setForeground(Color.blue);
                //Et on change l'alignement du texte grâce aux attributs static de
    la classe JLabel
                label.setHorizontalAlignment(JLabel.CENTER);

                container.add(label, BorderLayout.NORTH);

                this.setContentPane(container);
                this.setVisible(true);

                go();
        }

     //...

    //******************************************************************************
    // LA VOILAAAAAAAAAAAAA
    //******************************************************************************
            /**
    * C'est la méthode qui sera appelée lors d'un clic sur notre bouton
    */
            public void actionPerformed(ActionEvent arg0) {

            }

    }
```

Nous allons maintenant prévenir notre objet **Bouton** que notre objet **Fenetre** l'écoute ! Vous l'avez deviné, nous ajoutons notre objet **Fenetre** à la liste des objets qui écoutent notre **Bouton** grâce à la méthode **addActionListener(ActionListener obj)** invoquée sur la variable **bouton**. Ajoutez cette instruction dans le constructeur, en passant **this** en paramètre (c'est notre **Fenetre** qui écoute notre bouton...).

Une fois ceci fait, nous allons changer le texte de notre **JLabel** dans la méthode **actionPerformed** ; en fait, nous allons compter combien de fois on clique sur notre bouton... Pour cela, nous ajoutons une variable d'instance de type **int** dans notre classe : appelons-la ***compteur***. Dans la méthode **actionPerformed**, nous allons incrémenter ce compteur et afficher son contenu dans notre étiquette.

Voici le code de notre objet mis à jour :

Code : Java

```
    import java.awt.BorderLayout;
    import java.awt.Color;
    import java.awt.Font;
    import java.awt.event.ActionEvent;
    import java.awt.event.ActionListener;

    import javax.swing.JFrame;
    import javax.swing.JLabel;
    import javax.swing.JPanel;

    public class Fenetre extends JFrame implements ActionListener{

        private Panneau pan = new Panneau();
        private Bouton bouton = new Bouton("mon bouton");
        private JPanel container = new JPanel();
        private JLabel label = new JLabel("Le JLabel");
        /**
     * Compteur de clics !
```

```
     */
        private int compteur = 0;

        public Fenetre(){

                this.setTitle("Animation");
                this.setSize(300, 300);
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setLocationRelativeTo(null);

                container.setBackground(Color.white);
                container.setLayout(new BorderLayout());
                container.add(pan, BorderLayout.CENTER);

                //On ajoute notre Fenetre à la liste des auditeurs de notre Bouton
                bouton.addActionListener(this);

                container.add(bouton, BorderLayout.SOUTH);

                //Définition d'une police d'écriture
                Font police = new Font("Tahoma", Font.BOLD, 16 );
                //On applique celle-ci aux JLabel
                label.setFont(police);
                //On change la couleur de police
                label.setForeground(Color.blue);
                //Et on change l'alignement du texte grâce aux attributs static de
    la classe JLabel
                label.setHorizontalAlignment(JLabel.CENTER);

                container.add(label, BorderLayout.NORTH);

                this.setContentPane(container);
                this.setVisible(true);

                go();
        }

            private void go(){

            //Les coordonnées de départ de notre rond
            int x = pan.getPosX(), y = pan.getPosY();
            //Le booléen pour savoir si on recule ou non sur l'axe X
            boolean backX = false;
            //Le booléen pour savoir si on recule ou non sur l'axe Y
            boolean backY = false;

            //Pour cet exemple, j'utilise une boucle while
            //Vous verrez qu'elle fonctionne très bien
            while(true){

                    //Si la coordonnée x est inférieure à 1, on avance
                    if(x < 1)backX = false;
                    //Si la coordonnée x est supérieure à la taille du Panneau
                    //moins la taille du rond on avance
                    if(x > pan.getWidth()-50)backX = true;

                    //idem pour l'axe Y
                    if(y < 1)backY = false;
                    if(y > pan.getHeight()-50)backY = true;

                    //Si on avance, on incrémente la coordonnée
                    if(!backX)
                            pan.setPosX(++x);
                    //Sinon on décrémente
                    else
                            pan.setPosX(--x);

                    //Idem pour l'axe Y
                    if(!backY)
                            pan.setPosY(++y);
```

```
                        else
                                pan.setPosY(--y);

                        //On redessine notre Panneau
                        pan.repaint();

                        //Comme on dit : la pause s'impose ! Ici, 3 centièmes de
secondes
                        try {
                                Thread.sleep(3);
                        } catch (InterruptedException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                }

        }

        //*************************************************************************
        // LA VOILAAAAAAAAAAAAA
        //*************************************************************************
                /**
        * C'est la méthode qui sera appelée lors d'un clic sur notre bouton
        */
                public void actionPerformed(ActionEvent arg0) {
                        //Lorsque nous cliquons sur notre bouton, on met à jour le
JLabel
                        this.compteur++;
                        label.setText("Vous avez cliqué " + this.compteur + " fois");
                }

        }
```

Et le résultat :

On commence à faire du sérieux, là ! !
Mais attendez, on ne fait que commencer... Eh oui ! Nous allons maintenant ajouter un deuxième bouton à notre **Fenetre**, à côté de notre premier bouton (vous êtes libres d'utiliser la classe personnalisée ou un **JButton**) ! Personnellement, je vais utiliser des boutons normaux maintenant ; en effet, la façon dont on écrit le nom de notre bouton, dans notre classe personnalisée, n'est pas assez souple et donc l'affichage peut être décevant...

Bref, nous avons maintenant deux boutons écoutés par notre objet **Fenetre**.

Vous devez créer un deuxième **JPanel** qui va contenir nos deux boutons et insérer celui-ci dans le `contentPane` en **BorderLayout.SOUTH**.
Si vous tentez de mettre deux composants au même endroit avec un BorderLayout, seul le dernier composant ajouté apparaîtra ! Eh oui, le composant prend toute la place dans un **BorderLayout** !

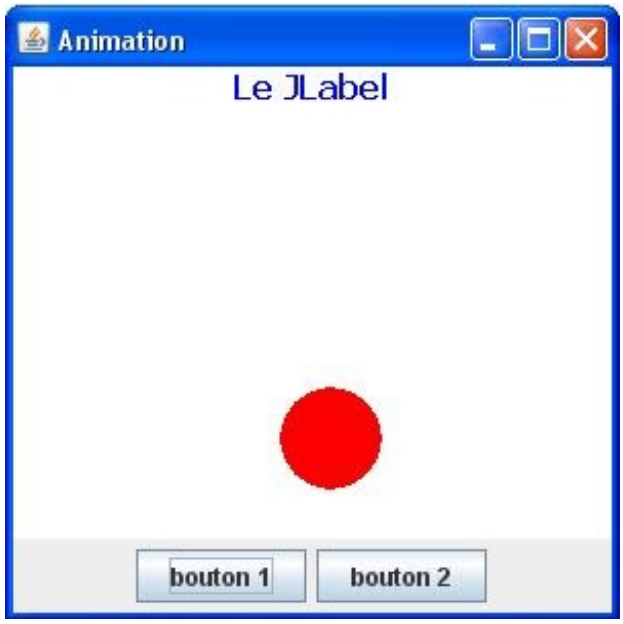
Voilà notre nouveau code :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class Fenetre extends JFrame implements ActionListener{

    private Panneau pan = new Panneau();
    private JButton bouton = new JButton("bouton 1");
    private JButton bouton2 = new JButton("bouton 2");
    private JPanel container = new JPanel();
    private JLabel label = new JLabel("Le JLabel");
    /**
* Compteur de clics !
*/
    private int compteur = 0;

    public Fenetre(){

            this.setTitle("Animation");
            this.setSize(300, 300);
            this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            this.setLocationRelativeTo(null);

            container.setBackground(Color.white);
            container.setLayout(new BorderLayout());
            container.add(pan, BorderLayout.CENTER);

            //On ajoute notre Fenetre à la liste des auditeurs de notre Bouton
            bouton.addActionListener(this);
            bouton2.addActionListener(this);

            JPanel south = new JPanel();
            south.add(bouton);
            south.add(bouton2);
            container.add(south, BorderLayout.SOUTH);


            //Définition d'une police d'écriture
            Font police = new Font("Tahoma", Font.BOLD, 16 );
            //On applique celle-ci aux JLabel
            label.setFont(police);
            //On change la couleur de police
            label.setForeground(Color.blue);
            //Et on change l'alignement du texte grâce aux attributs static de
la classe JLabel
            label.setHorizontalAlignment(JLabel.CENTER);

            container.add(label, BorderLayout.NORTH);
```

```
                this.setContentPane(container);
                this.setVisible(true);

                go();
        }

        //...

    //*******************************************************************************
    // LA VOILAAAAAAAAAAAAA
    //*******************************************************************************
            /**
    * C'est la méthode qui sera appelée lors d'un clic sur notre bouton
    */
            public void actionPerformed(ActionEvent arg0) {
                    //Lorsque nous cliquons sur notre bouton, on met à jour le
    JLabel
                    this.compteur++;
                    label.setText("Vous avez cliqué " + this.compteur + " fois");
            }

    }
```

Et le résultat :



Le problème maintenant est :

comment faire faire deux choses différentes dans la méthode **`actionPerformed`** ?

En effet ! Si nous laissons la méthode **`actionPerformed`** telle qu'elle est, les deux boutons auront la même action lorsque nous cliquerons dessus. Essayez, et vous verrez !

Il existe un moyen de savoir qui a déclenché l'événement, en utilisant l'objet passé en paramètre dans la méthode **`actionPerformed`**. Nous allons utiliser la méthode **`getSource()`** de cet objet pour connaître le nom de l'instance qui a généré l'événement.
Testez la méthode **`actionPerformed`** suivante, et voyez le résultat :

Code : Java

```
	public void actionPerformed(ActionEvent arg0) {

			if(arg0.getSource() == bouton)
				label.setText("Vous avez cliqué sur le
	bouton 1");

			if(arg0.getSource() == bouton2)
				label.setText("Vous avez cliqué sur le
	bouton 2");
		}
```

Résultat :

Vous pouvez constater que notre code fonctionne très bien ! Mais cette approche n'est pas très orientée objet... Si vous avez une multitude de boutons sur votre IHM... vous allez avoir une méthode **`actionPerformed`** très chargée !
Nous pourrions créer deux objets à part, chacun écoutant un bouton, dont le rôle serait de faire un traitement précis par bouton... Cependant, si dans nos traitements nous avons besoin de modifier des données internes à la classe contenant nos boutons, il faudrait passer ces données (ou objets) à cet objet... Pas terrible non plus.

On commence à te connaître, maintenant ! Tu as une idée derrière la tête...

Je suis démasqué !
Il existe en Java un type de classe particulière. Voyons ça tout de suite !
Retour en haut

Parlez avec votre classe intérieure

En Java, vous pouvez faire ce qu'on appelle des **classes internes**.
Ceci consiste à déclarer une classe dans une classe ! Je sais, ça paraît tordu mais vous allez voir que c'est très pratique.

En effet, ces classes possèdent tous les avantages des classes normales, héritant d'une super-classe ou implémentant une interface, elles bénéficieront donc des bénéfices du polymorphisme et de la covariance des variables !
En plus, elles ont l'avantage d'avoir accès aux attributs de la classe dans laquelle elles sont déclarées !

Dans le cas qui nous intéresse, ceci permet de faire une implémentation de l'interface **ActionListener**, détachée de notre classe **Fenetre**, mais pouvant utiliser ses attributs !
La déclaration d'une telle classe se fait exactement comme une classe normale, sauf qu'elle est dans une autre classe... Ce qui donne ceci :

Code : Java

```
public class MaClasseExterne{

      public MaClasseExterne(){
            //....
      }

      class MaClassInterne{

            public MaClassInterne(){
                  //...
            }
      }
}
```

Grâce à ceci, nous allons pouvoir faire une classe spécialisée dans l'écoute de composants et qui a un travail précis à faire ! Dans notre exemple, nous allons juste faire deux classes internes implémentant chacune l'interface **ActionListener** ; elles redéfiniront donc la méthode **actionPerformed** :

- **BoutonListener** : qui écoutera le premier bouton
- **Bouton2Listener** : qui écoutera le second !

Une fois que ceci est fait, il ne nous reste plus qu'à dire à chaque bouton : "***qui l'écoute***", avec la méthode **addActionListener**.

Voici la classe Fenetre mise à jour :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class Fenetre extends JFrame{

    private Panneau pan = new Panneau();
    private JButton bouton = new JButton("bouton 1");
    private JButton bouton2 = new JButton("bouton 2");
    private JPanel container = new JPanel();
    private JLabel label = new JLabel("Le JLabel");
    private int compteur = 0;

    public Fenetre(){
```

```
    public Fenetre(){

        this.setTitle("Animation");
        this.setSize(300, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());
        container.add(pan, BorderLayout.CENTER);

        //Ce sont maintenant nos classes internes qui écoutent
nos boutons
        bouton.addActionListener(new BoutonListener());
        bouton2.addActionListener(new Bouton2Listener());

        JPanel south = new JPanel();
        south.add(bouton);
        south.add(bouton2);
        container.add(south, BorderLayout.SOUTH);

        Font police = new Font("Tahoma", Font.BOLD, 16 );
        label.setFont(police);
        label.setForeground(Color.blue);
        label.setHorizontalAlignment(JLabel.CENTER);

        container.add(label, BorderLayout.NORTH);
        this.setContentPane(container);
        this.setVisible(true);

        go();
    }

    private void go(){

    //Les coordonnées de départ de notre rond
    int x = pan.getPosX(), y = pan.getPosY();
    //Le booléen pour savoir si on recule ou non sur l'axe X
    boolean backX = false;
    //Le booléen pour savoir si on recule ou non sur l'axe Y
    boolean backY = false;

    //Pour cet exemple, j'utilise une boucle while
    //Vous verrez qu'elle fonctionne très bien
    while(true){

        if(x < 1)backX = false;
        if(x > pan.getWidth()-50)backX = true;
        if(y < 1)backY = false;
        if(y > pan.getHeight()-50)backY = true;
        if(!backX)pan.setPosX(++x);
        else pan.setPosX(--x);
        if(!backY) pan.setPosY(++y);
        else pan.setPosY(--y);
        pan.repaint();

        try {
                Thread.sleep(3);
        } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
    }

    }


    /**
* classe qui écoute notre bouton
*/
```

```
        class BoutonListener  implements ActionListener{

                /**
	* Redéfinition de la méthode actionPerformed
	*/
                public void actionPerformed(ActionEvent arg0) {
                        label.setText("Vous avez cliqué sur le
	bouton 1");
                }

        }

        /**
	* classe qui écoute notre bouton2
	*/
        class Bouton2Listener  implements ActionListener{

                /**
	* Redéfinition de la méthode actionPerformed
	*/
                public void actionPerformed(ActionEvent e) {
                        label.setText("Vous avez cliqué sur le
	bouton 2");
                }

        }



	}
```

Et le résultat est parfait :

Vous pouvez même constater que nos classes internes ont accès aux attributs déclarés **private** dans notre classe **Fenetre** !

Nous n'avons plus à nous soucier de qui a déclenché l'événement maintenant, car nous avons une classe qui écoute chaque bouton ! Nous pouvons souffler un peu, une grosse épine vient de nous être retirée du pied.

Vous le savez, mais vous pouvez faire écouter votre bouton par plusieurs classes... Il vous suffit d'ajouter les classes qui écoutent le boutons avec **addActionListener**.

Eh oui, faites le test...
Créez une troisième classe interne, peu importe son nom (moi, je l'appelle **Bouton3Listener**), implémentez l'interface **ActionListener** dans celle-ci et contentez-vous de faire un simple **System.out.println** dans la méthode **actionPerformed**.
N'oubliez pas d'ajouter cette dernière à la liste des classes qui écoutent votre bouton (n'importe lequel des deux... Moi, j'ai choisi le premier).

Je ne vous donne que le code ajouté :

Code : Java

```
//Les imports
public class Fenetre extends JFrame{

     //Les variables d'instance...
      public Fenetre(){

            this.setTitle("Animation");
            this.setSize(300, 300);
            this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            this.setLocationRelativeTo(null);

            container.setBackground(Color.white);
            container.setLayout(new BorderLayout());
            container.add(pan, BorderLayout.CENTER);

            //Première classe écoutant mon bouton
            bouton.addActionListener(new BoutonListener());
            //Deuxième classe écoutant mon bouton
            bouton.addActionListener(new Bouton3Listener());

            bouton2.addActionListener(new Bouton2Listener());

            JPanel south = new JPanel();
            south.add(bouton);
            south.add(bouton2);
            container.add(south, BorderLayout.SOUTH);

            Font police = new Font("Tahoma", Font.BOLD, 16 );
            label.setFont(police);
            label.setForeground(Color.blue);
            label.setHorizontalAlignment(JLabel.CENTER);

            container.add(label, BorderLayout.NORTH);
            this.setContentPane(container);
            this.setVisible(true);

            go();
    }

    //...

        class Bouton3Listener  implements ActionListener{

              /**
* Redéfinition de la méthode actionPerformed
*/
              public void actionPerformed(ActionEvent e) {
                    System.out.println("Ma classe interne numéro
3 écoute bien !");
              }

        }
}
```

Et le résultat :

Les classes internes sont vraiment des classes à part entière ! Elles peuvent aussi être héritées d'une super-classe... De ce fait, c'est presque comme si nous avions de l'héritage multiple, ça n'en est pas... Mais ça y ressemble. Donc, ce code est valide :

Code : Java

```
public class MaClasseExterne extends JFrame{

	public MaClasseExterne(){
		//....
	}

	class MaClassInterne extends JPanel{

		public MaClassInterne(){
			//...
		}
	}

	class MaClassInterne2 extends JButton{

		public MaClassInterne(){
			//...
		}
	}

}
```

Rien ne vous empêche de faire de votre classe **Panneau** une classe interne !

Vous voyez bien que ce genre de classes peuvent s'avérer très utile...

Bon. Nous avons réglé le problème d'implémentation, nous avons deux boutons qui sont écoutés, il ne nous reste plus qu'à lancer et arrêter notre animation avec ceux-ci !

Retour en haut

Contrôler votre animation : lancement et arrêt

Nous attaquons la dernière ligne droite de ce chapitre.
Donc, pour réussir à gérer le lancement et l'arrêt de notre animation, nous allons devoir modifier un peu le code de notre classe **`Fenetre`**.
Il va falloir changer le libellé de ceux-ci, le premier affichera Go et le deuxième Stop et, pour éviter d'arrêter l'animation alors que celle-ci n'est pas lancée et ne pas l'animer alors qu'elle l'est déjà, nous allons tantôt activer / désactiver les boutons. Je m'explique.

- Au lancement, l'animation est lancée. Le bouton Go ne sera pas cliquable alors que Stop, oui.
- Si l'animation est stoppée, le bouton Stop ne sera plus cliquable et le bouton Go le sera.

Tu ne nous as pas dit comment on fait ça !

Je sais... Mais je vous cache encore pas mal de choses...
Ne vous inquiétez pas, c'est très simple à faire ; il existe une méthode pour faire ça :

Code : Java

```
JButton bouton = new JButton("bouton");
bouton.setEnabled(false); // Votre bouton n'est plus cliquable !
bouton.setEnabled(true); // Votre bouton de nouveau cliquable !
```

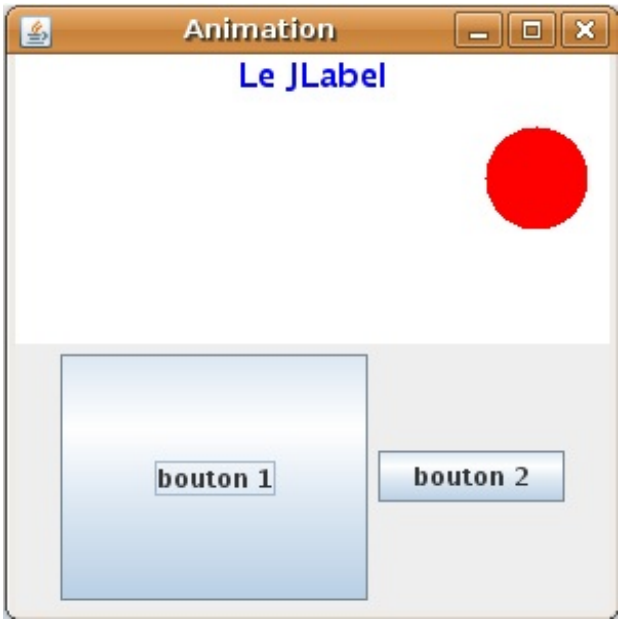
J'ajouterais que vous pouvez faire pas mal de choses avec ces objets ! Soyez curieux et testez les méthodes de ces objets, allez faire un tour sur le site de Sun Microsystems, fouillez, fouinez...
L'une des méthodes qui s'avère souvent utile et qui est utilisable pour tous ces objets (et les objets que nous verrons plus tard) est la méthode de gestion de dimension. Il ne s'agit pas de la méthode **`setSize()`**, mais la méthode **`setPreferredSize(Dimension preferredSize)`**. Cette méthode prend un objet **`Dimension`** en paramètre qui lui, prend **deux entiers** en paramètre.

Voilà un exemple :

Code : Java

```
bouton.setPreferredSize(new Dimension(150, 120));
```

Ce qui donne sur notre application :



Retournons à nos moutons...
Afin de bien gérer notre animation, il va falloir améliorer notre méthode **go()**.
Nous allons sortir de cette méthode les deux entiers dont nous nous servions afin de recalculer les coordonnées de notre rond et, concernant la boucle infinie, elle doit dorénavant pouvoir être stoppée !
Pour réussir ceci, nous allons déclarer une variable d'instance de type booléen qui changera d'état selon le bouton sur lequel nous cliquerons, et nous utiliserons celui-ci comme paramètre de notre boucle.

Voici le code de notre classe **Fenetre** :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class Fenetre extends JFrame{

    private Panneau pan = new Panneau();
    private JButton bouton = new JButton("Go");
    private JButton bouton2 = new JButton("Stop");
    private JPanel container = new JPanel();
    private JLabel label = new JLabel("Le JLabel");
    private int compteur = 0;
    private boolean animated = true;
    private boolean backX, backY;
    private int x,y ;

    public Fenetre(){

            this.setTitle("Animation");
            this.setSize(300, 300);
            this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            this.setLocationRelativeTo(null);

            container.setBackground(Color.white);
            container.setLayout(new BorderLayout());
            container.add(pan, BorderLayout.CENTER);

            //Ce sont maintenant nos classes internes qui écoutent
nos boutons
            bouton.addActionListener(new BoutonListener());
            bouton.setEnabled(false);
            bouton2.addActionListener(new Bouton2Listener());

            JPanel south = new JPanel();
            south.add(bouton);
            south.add(bouton2);
            container.add(south, BorderLayout.SOUTH);

            Font police = new Font("Tahoma", Font.BOLD, 16 );
            label.setFont(police);
            label.setForeground(Color.blue);
            label.setHorizontalAlignment(JLabel.CENTER);

            container.add(label, BorderLayout.NORTH);
            this.setContentPane(container);
            this.setVisible(true);
            go();

    }
```

```
        private void go(){
        //Les coordonnées de départ de notre rond
        x = pan.getPosX();
        y = pan.getPosY();
        //Pour cet exemple, j'utilise une boucle while
        //Vous verrez qu'elle fonctionne très bien
        while(this.animated){

            if(x < 1)backX = false;
            if(x > pan.getWidth()-50)backX = true;
            if(y < 1)backY = false;
            if(y > pan.getHeight()-50)backY = true;


            if(!backX)pan.setPosX(++x);
            else pan.setPosX(--x);
            if(!backY) pan.setPosY(++y);
            else pan.setPosY(--y);
            pan.repaint();

            try {
                    Thread.sleep(3);
            } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            }
        }
        }


        /**
* classe qui écoute notre bouton
*/
        class BoutonListener implements ActionListener{

                /**
* Redéfinitions de la méthode actionPerformed
*/
                public void actionPerformed(ActionEvent arg0) {
                        animated = true;
                        bouton.setEnabled(false);
                        bouton2.setEnabled(true);
                        go();
                }

        }

        /**
* classe qui écoute notre bouton2
*/
        class Bouton2Listener  implements ActionListener{

                /**
* Redéfinitions de la méthode actionPerformed
*/
                public void actionPerformed(ActionEvent e) {
                        animated = false;
                        bouton.setEnabled(true);
                        bouton2.setEnabled(false);
                }

        }
}
```

À l'exécution, vous pouvez voir :

- que le bouton Go n'est pas cliquable mais que l'autre l'est
- que l'animation se lance
- lorsque vous cliquez sur Stop, que l'animation s'arrête
- que le bouton Go devient cliquable
- et lors de votre tentative, que l'animation ne se lance pas !

Comment ça se fait ?

Comme je vous l'ai dit dans le chapitre sur les conteneurs, notre application, au démarrage de celle-ci, lance un **thread** : **le processus principal de votre programme !**
Au démarrage, l'animation est lancée, celle-ci dans le même `thread` que notre objet **`Fenetre`**.
Lorsque nous lui demandons de s'arrêter, aucun souci : les ressources mémoire sont libérées, on sort de la boucle infinie et l'application continue son court !
Mais lorsque nous redemandons à l'animation de se lancer... L'instruction dans la méthode **`actionPerformed`** appelle la méthode **`go`** et, vu que nous sommes dans une boucle infinie : **on reste dans la méthode `go` et on ne sort plus de la méthode `actionPerformed` de notre bouton !**

Voici l'explication de ce phénomène

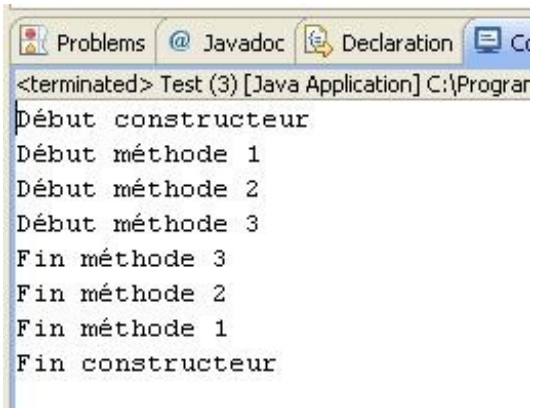
Java gère les appels aux méthodes selon ce qu'on appelle vulgairement : **La pile** !

Pour vous expliquer ceci, nous allons prendre un exemple tout bête ; regardez cet objet :

Code : Java

```
public class TestPile {

        public TestPile(){
                System.out.println("Début constructeur");
                methode1();
                System.out.println("Fin constructeur");
        }

        public void methode1(){
                System.out.println("Début méthode 1");
                methode2();
                System.out.println("Fin méthode 1");
        }

        public void methode2(){
                System.out.println("Début méthode 2");
                methode3();
                System.out.println("Fin méthode 2");
        }

        public void methode3(){
                System.out.println("Début méthode 3");
                System.out.println("Fin méthode 3");
        }

}
```

Si vous instanciez cet objet, vous obtiendrez dans votre console ceci :

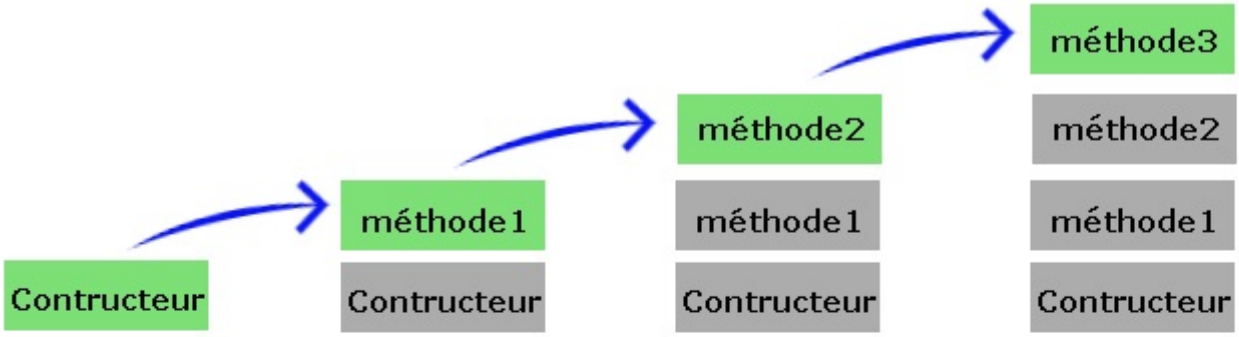


Je suppose que vous avez remarqué, avec stupéfaction, que l'ordre des instructions est un peu bizarre !
Voici ce qu'il s'est passé :

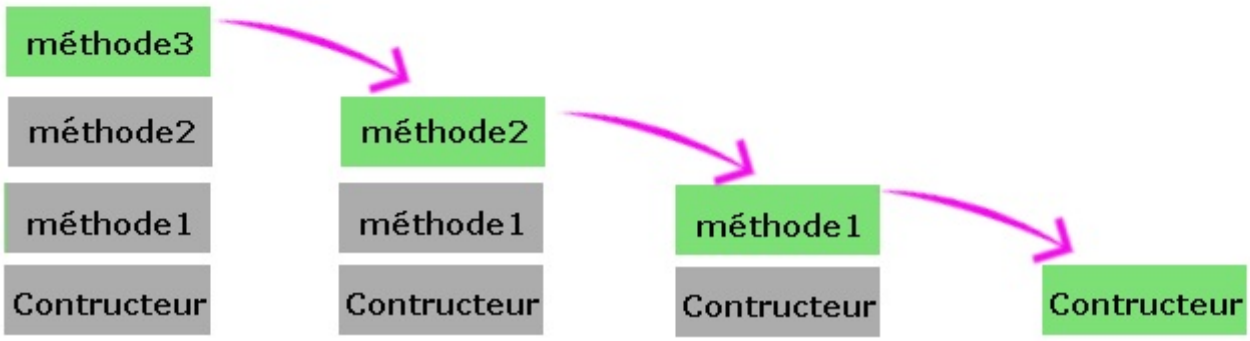
- à l'instanciation, notre objet appelle la **`méthode1`** ;
- cette dernière invoque la **`méthode2`** ;
- celle-ci utilise la **`méthode3`** et une fois terminée, la JVM retourne dans la **`méthode2`** ;
- celle-ci terminée, on remonte à la fin de la **`méthode1`**... jusqu'à la dernière instruction appelante : le contructeur.

Lors de tous les appels, on dit que la JVM empile les invocations sur la pile ! Et, une fois la dernière méthode empilée terminée, la JVM dépile celle-ci !

Voilà un schéma résumant la situation :



Tant que la méthode au sommet de la pile n'est pas terminée, celle-ci n'est pas dépilée ! Les autres méthodes de la pile sont bloquées !



Dans notre programme, imaginez que la méthode **`actionPerformed`** est représentée, dans le schéma ci-dessus, par **`méthode2`** et que notre méthode **`go`** soit représentée par **`méthode3`**. Lorsque nous entrons dans **`méthode3`**, nous entrons

dans une boucle infinie ! La conséquence directe : on ne ressort jamais de cette méthode et la JVM ne **dépile** plus !

Comment faire, alors ?

Nous allons utiliser une nouveau `thread` ! En gros, grâce à ceci, la méthode **go** se trouvera dans une pile à part !

Attends... On arrive pourtant à arrêter l'animation alors qu'elle est dans une boucle infinie ? Pourquoi ?

Tout simplement parce que nous demandons de faire une bête initialisation de variable dans la gestion de notre événement ! Si vous faites une deuxième méthode comprenant une boucle infinie et que vous l'invoquez lors du clic sur Stop, vous aurez exactement le même problème !

Je ne vais pas m'éterniser là-dessus tout de suite... Nous allons voir ça au prochain chapitre !
Mais avant, je pense qu'un TP serait le bienvenu... Histoire de réviser un peu !
Vous êtes d'accord ? Alors, direction le topo, le QCM, et en avant pour un TP !

Retour en haut

Cadeau : votre bouton personnalisé optimisé !

Vu que je vous ai fait un peu saliver pour rien... je vous donne le code java d'un bouton corrigeant le problème d'écriture du libellé. Je ne ferai pas trop de commentaire à son sujet, tout est dans le code...

Code : Java

```
import java.awt.Color;
import java.awt.FontMetrics;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;
import javax.swing.JButton;


public class Bouton extends JButton implements MouseListener{

	private String name;
	private Image img;

	public Bouton(String str){
		super(str);
		this.name = str;

		try {
			img = ImageIO.read(new File("fondBouton.png"));
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		this.addMouseListener(this);
	}

	public void paintComponent(Graphics g){

		Graphics2D g2d = (Graphics2D)g;

		GradientPaint gp = new GradientPaint(0, 0, Color.blue,
0, 20, Color.cyan, true);
		g2d.setPaint(gp);
		// g2d.fillRect(0, 0, this.getWidth(),
this.getHeight());
		g2d.drawImage(img, 0, 0, this.getWidth(),
this.getHeight(), this);

		g2d.setColor(Color.black);

		//Objet qui permet de connaître les propriétés d'une
police, dont la taille !
		FontMetrics fm = g2d.getFontMetrics();
		//Hauteur de la police d'écriture
		int height = fm.getHeight();
		//Largeur totale de la chaîne passée en paramètre
		int width = fm.stringWidth(this.name);

		//On calcule donc la position du texte dans le bouton,
et le tour est joué !
		g2d.drawString(this.name, this.getWidth() / 2 - (width
/ 2), (this.getHeight() / 2) + (height / 4));

	}
```

```
        @Override
        public void mouseClicked(MouseEvent event) {
                 //Pas utile d'utiliser cette méthode ici
        }

        @Override
        public void mouseEntered(MouseEvent event) {

                //Nous changeons le fond en jaune pour notre image
lors du survol
                //avec le fichier fondBoutonHover.png
                try {
                        img = ImageIO.read(new
File("fondBoutonHover.png"));
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }

        @Override
        public void mouseExited(MouseEvent event) {

                //Nous changeons le fond en vert pour notre image
lorsqu'on quitte le bouton
                //avec le fichier fondBouton.png
                try {
                        img = ImageIO.read(new
File("fondBouton.png"));
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }

        @Override
        public void mousePressed(MouseEvent event) {

                //Nous changeons le fond en orangé pour notre image
lors du clic gauche
                //avec le fichier fondBoutonClic.png
                try {
                        img = ImageIO.read(new
File("fondBoutonClic.png"));
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }

        @Override
        public void mouseReleased(MouseEvent event) {

                //Nous changeons le fond en orangé pour notre image

                //lorsqu'on relâche le clic
                //avec le fichier fondBoutonHover.png

                //Si on est à l'extérieur de l'objet, on dessine le
fond par défaut
                if(event.getY() > 0)
                {
                        try {
                                img = ImageIO.read(new
File("fondBoutonHover.png"));
                        } catch (IOException e) {
```

```
                                        // TODO Auto-generated catch block
                                        e.printStackTrace();
                                }
                        }
                        //Sinon on met le fond jaune, la souris est encore
        dessus...
                        else
                        {
                                try {
                                         img = ImageIO.read(new
        File("fondBouton.png"));
                                } catch (IOException e) {
                                        // TODO Auto-generated catch block
                                        e.printStackTrace();
                                }
                        }
                }

        }
```

Essayez et vous verrez !

Retour en haut

Ce qu'il faut retenir

- L'interface utilisée dans ce chapitre est : **`ActionListener`**. Celle-ci est présente dans le package **`java.awt`**.
- La méthode de cette interface à implémenter est **`actionPerformed`**.
- Lorsque vous avez plusieurs composants pouvant être écoutés, préférez faire un objet écouteur par composants.
- Vous pouvez, et c'est conseillé, utiliser des **classes internes** pour écouter vos composants.
- Une classe interne est une classe à l'intérieur d'une classe.
- Une telle classe a accès à toutes les données et méthodes de sa classe externe.
- Ce type de classe peut être déclarée **`private`**.
- La JVM traite les méthodes appelées par une **pile** de méthode, définissant ainsi l'ordre d'exécution de celles-ci.
- **Une méthode est empilée à l'invocation de celle-ci, mais n'est dépilée que lorsque toutes ses intructions sont terminées !**
- Vous connaissez la méthode permettant d'écrire dans un **`JLabel`**, vous pouvez aussi récupérer le texte d'un tel objet en utilisant la méthode **`getText()`**.

Une classe interne privée ?

Oui, parce que vous pouvez créer une instance de la classe interne à l'extérieur de sa classe externe, comme ceci :

Code : Java

```
Fenetre.BoutonListener boutonL;
boutonL = new Fenetre().new BoutonListener();
```

Par contre, vous ne pouvez pas déclarer de variable **`static`** dans une classe interne... Mais une classe interne peut être déclarée **`static`**.

Retour en haut

Encore un chapitre de bouclé !
Vous tenez toujours bon ?

À tout de suite pour votre premier TP en événementiel !

Retour en haut

TP : une calculatrice

Ah ! Ça faisait longtemps... un petit TP...

Dans celui-ci, nous allons - enfin vous allez - pouvoir réviser tout ce que vous avez appris dans cette partie !

- Fenetre
- Conteneur
- Bouton
- **`ActionPerformed`**
- Classe interne
- Exception. Bon, ça, c'était dans celle d'avant.

Allez mes ZérOs, en avant !

Élaboration

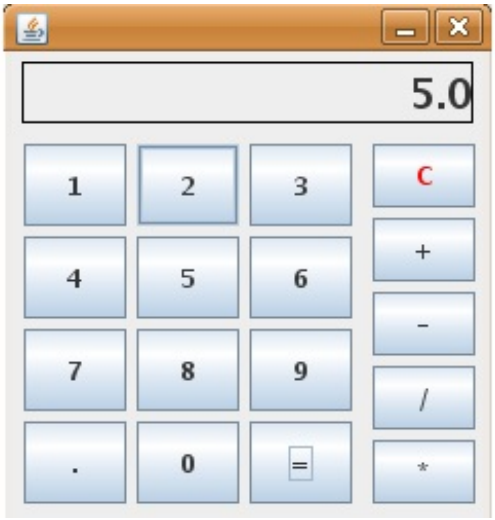
Bon, nous allons voir ce que doit faire notre calculatrice.

- Un calcul simple : 12 + 3 =
- Des calculs à la chaîne... Exemple : 1 + 2 +, lorsque vous cliquez à nouveau sur un opérateur, vous affichez le résultat du calcul précédent !
- Donner la possibilté de faire un reset, donc de tout faire recommencer à zéro.
- Gérer l'exception pour une division par 0 !

Retour en haut

Conception

Voilà ce que vous devriez avoir :



Maintenant, voyons ce dont nous avons besoin pour réussir ça.

- Autant de boutons qu'il en faut.
- Autant de conteneurs que nécessaire.
- Un **`JLabel`** pour l'affichage.
- Un booléen pour savoir si un opérateur a été choisi.
- Un booléen pour savoir si nous devons effacer ce qu'il y a à l'écran et écrire un nouveau nombre.
- Nous allons utiliser une variable de type `double` pour nos calculs.
- Il va falloir des classes internes qui implémenteront l'interface **`ActionListener`**.
- Et c'est à peu près tout...

Pour alléger le nombre de classes internes, vous pouvez en faire une qui va se charger d'écrire ce qui doit être affiché à l'écran.
Utiliser la méthode **`getSource()`** pour savoir de quel bouton il s'agit...

Je ne vais pas tout vous dire... Il faut que vous cherchiez par vous-mêmes... La réflexion est très bonne !
Par contre, je vous le dis tout de suite :

la correction que je vous fournis n'est pas **LA** correction.
Il n'y a pas de bonne solution : si, en fait, considérez un programme qui fonctionne comme une solution.
Je vous propose seulement une **solution POSSIBLE**.

Allez, en avant mes ZérOs !
Retour en haut

Correction

Vous avez bien réfléchi ?
Vous vous êtes brûlé quelques neurones... Vous avez mérité votre correction...

Secret (cliquez pour afficher)

Classe Main.java

Code : Java

```
public class Main {

	/**
* @param args
*/
	public static void main(String[] args) {
		Calculatrice calculette = new Calculatrice();

	}

}
```

Classe Calculatrice.java

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;


public class Calculatrice extends JFrame {

	private JPanel container = new JPanel();

	String[] tab_string = {"1", "2", "3", "4", "5", "6", "7",
"8", "9", "0", ".", "=", "C", "+", "-", "*", "/"};
	JButton[] tab_button = new JButton[tab_string.length];

	private JLabel ecran = new JLabel();
	private Dimension dim = new Dimension(50, 40);
	private Dimension dim2 = new Dimension(50, 31);
	private double chiffre1;
	private boolean clicOperateur = false, update = false;
	private String operateur = "";

	public Calculatrice(){

		this.setSize(240, 260);
		this.setTitle("Calculette");
```

```
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                        this.setLocationRelativeTo(null);
                        this.setResizable(false);
                        initComposant();

                        this.setContentPane(container);
                        this.setVisible(true);
                }

                private void initComposant(){

                        Font police = new Font("Arial", Font.BOLD, 20);
                        ecran = new JLabel("0");
                        ecran.setFont(police);
                        ecran.setHorizontalAlignment(JLabel.RIGHT);
                        ecran.setPreferredSize(new Dimension(220, 20));

                        JPanel operateur = new JPanel();
                        operateur.setPreferredSize(new Dimension(55,
        225));
                        JPanel chiffre = new JPanel();
                        chiffre.setPreferredSize(new Dimension(165, 225));
                        JPanel panEcran = new JPanel();
                        panEcran.setPreferredSize(new Dimension(220, 30));


                        for(int i = 0; i < tab_string.length; i++)
                        {

                            tab_button[i] = new JButton(tab_string[i]);
                            tab_button[i].setPreferredSize(dim);

                            switch(i){

                             case 11 :
                              tab_button[i].addActionListener(new
        EgalListener());
                              chiffre.add(tab_button[i]);
                              break;

                             case 12 :
                              tab_button[i].setForeground(Color.red);
                                 tab_button[i].addActionListener(new
        ResetListener());
                                 tab_button[i].setPreferredSize(dim2);
                                 operateur.add(tab_button[i]);
                              break;

                             case 13 :
                              tab_button[i].addActionListener(new
        PlusListener());
                              tab_button[i].setPreferredSize(dim2);
                              operateur.add(tab_button[i]);
                              break;

                             case 14 :
                              tab_button[i].addActionListener(new
        MoinsListener());
                              tab_button[i].setPreferredSize(dim2);
                              operateur.add(tab_button[i]);
                              break;

                             case 15 :
                              tab_button[i].addActionListener(new
        MultiListener());
                              tab_button[i].setPreferredSize(dim2);
                              operateur.add(tab_button[i]);
                              break;

                             case 16 :
```

```
                        tab_button[i].addActionListener(new
DivListener());
                        tab_button[i].setPreferredSize(dim2);
                        operateur.add(tab_button[i]);
                        break;

                       default :
                        chiffre.add(tab_button[i]);
                        tab_button[i].addActionListener(new
ChiffreListener());
                        break;
                      }

                }

                panEcran.add(ecran);

panEcran.setBorder(BorderFactory.createLineBorder(Color.black));

                container.add(panEcran, BorderLayout.NORTH);
                container.add(chiffre, BorderLayout.CENTER);
                container.add(operateur, BorderLayout.EAST);

        }


        private void calcul(){
                if(operateur.equals("+"))
                {
                        chiffre1 = chiffre1 +
Double.valueOf(ecran.getText()).doubleValue();
                        ecran.setText(String.valueOf(chiffre1));
                }

                if(operateur.equals("-"))
                {
                        chiffre1 = chiffre1 -
Double.valueOf(ecran.getText()).doubleValue();
                        ecran.setText(String.valueOf(chiffre1));
                }

                if(operateur.equals("*"))
                {
                        chiffre1 = chiffre1 *
Double.valueOf(ecran.getText()).doubleValue();
                        ecran.setText(String.valueOf(chiffre1));
                }

                if(operateur.equals("/"))
                {
                        try{
                                chiffre1 = chiffre1 /
Double.valueOf(ecran.getText()).doubleValue();

ecran.setText(String.valueOf(chiffre1));
                        }catch(ArithmeticException e){
                                ecran.setText("0");
                        }
                }
        }

        class ChiffreListener implements ActionListener{

                @Override
                public void actionPerformed(ActionEvent e) {
                        //On affiche le chiffre en plus dans le
label
                        String str =
((JButton)e.getSource()).getText();
```

```
                        if(update)
                        {
                                update = false;
                        }
                        else
                        {
                                if(!ecran.getText().equals("0"))
                                        str = ecran.getText() +
str;
                        }

                        ecran.setText(str);
                }

        }


        class EgalListener implements ActionListener{

                @Override
                public void actionPerformed(ActionEvent arg0) {
                        calcul();
                        update = true;
                        clicOperateur = false;
                }

        }


        class PlusListener implements ActionListener{

                @Override
                public void actionPerformed(ActionEvent arg0) {

                        if(clicOperateur)
                        {
                                calcul();

ecran.setText(String.valueOf(chiffre1));
                        }
                        else
                        {
                                chiffre1 =
Double.valueOf(ecran.getText()).doubleValue();
                                clicOperateur = true;
                        }
                        operateur = "+";
                        update = true;
                }

        }

        class MoinsListener implements ActionListener{

                @Override
                public void actionPerformed(ActionEvent arg0) {
                        if(clicOperateur)
                        {
                                calcul();

ecran.setText(String.valueOf(chiffre1));
                        }
                        else
                        {
                                chiffre1 =
Double.valueOf(ecran.getText()).doubleValue();
                                clicOperateur = true;
                        }
                        operateur = "-";
                        update = true;
```

```
                }

        }


        class MultiListener implements ActionListener{

                @Override
                public void actionPerformed(ActionEvent arg0) {
                        if(clicOperateur)
                        {
                                calcul();

ecran.setText(String.valueOf(chiffre1));
                        }
                        else
                        {
                                chiffre1 =
Double.valueOf(ecran.getText()).doubleValue();
                                clicOperateur = true;
                        }
                        operateur = "*";
                        update = true;
                }

        }


        class DivListener implements ActionListener{

                @Override
                public void actionPerformed(ActionEvent arg0) {
                        if(clicOperateur)
                        {
                                calcul();

ecran.setText(String.valueOf(chiffre1));
                        }
                        else
                        {
                                chiffre1 =
Double.valueOf(ecran.getText()).doubleValue();
                                clicOperateur = true;

                        }
                        operateur = "/";
                        update = true;
                }

        }


        class ResetListener implements ActionListener{

                @Override
                public void actionPerformed(ActionEvent arg0) {
                        clicOperateur = false;
                        update = true;
                        chiffre1 = 0;
                        operateur = "";
                        ecran.setText("");
                }

        }

}
```

Je ne m'attarderai pas trop là-dessus, ce code est très simple et je suis sûr que vous trouverez des améliorations .
Regardez bien comment tout interagit, et vous comprendrez comment il fonctionne !

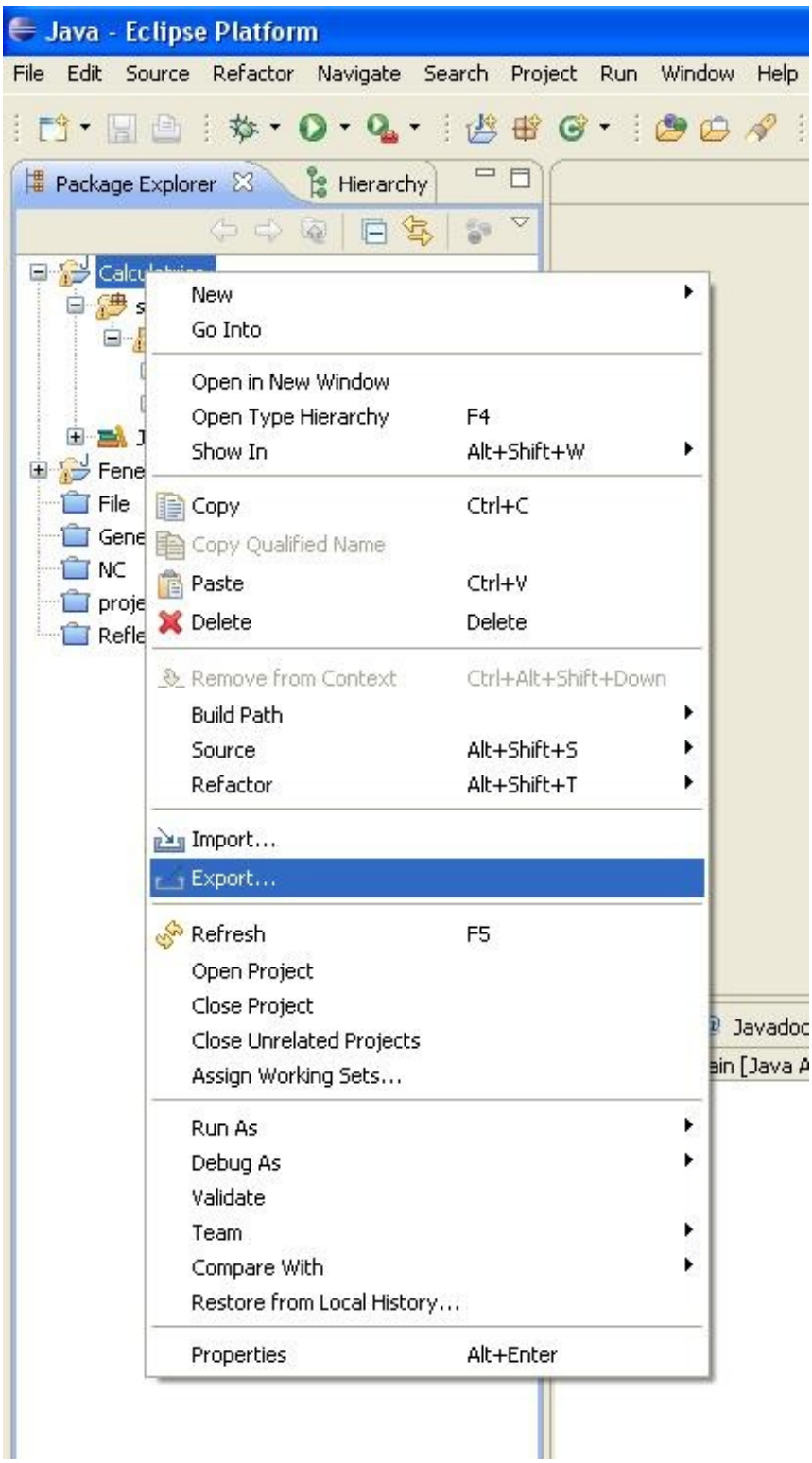
Je vais vous donner une petite astuce pour faire un **.jar** exécutable en java...
Retour en haut

Astuce Eclipse : faire un jar exécutable

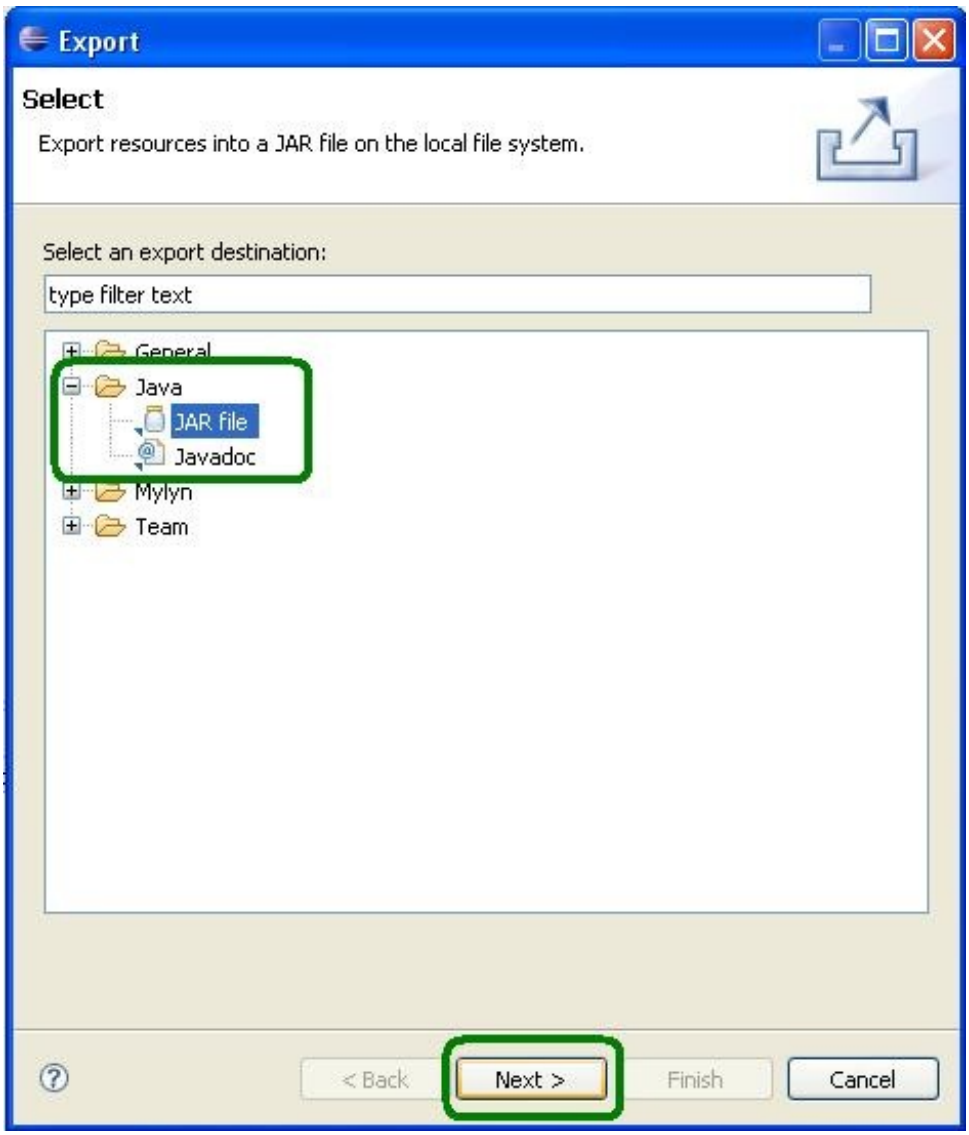
Tout d'abord, qu'est-ce qu'un **.jar** ?
Un tel fichier est une archive Java (**J**ava **AR**chive). Ce type de fichier contient tout ce dont a besoin la JVM pour lancer votre programme ! Une fois votre archive créée, il vous suffira juste de double cliquer sur celle-ci pour lancer l'application !

Ceci sous réserve que vous ayez ajouté les exécutables de votre JRE (présent dans le répertoire /bin) dans votre variable d'environnement `PATH` !
Si ceci n'est pas fait, allez faire un tour ici en remplaçant le répertoire du JDK par celui du JRE (si vous n'avez pas téléchargé le JDK, sinon, prenez ce dernier...).

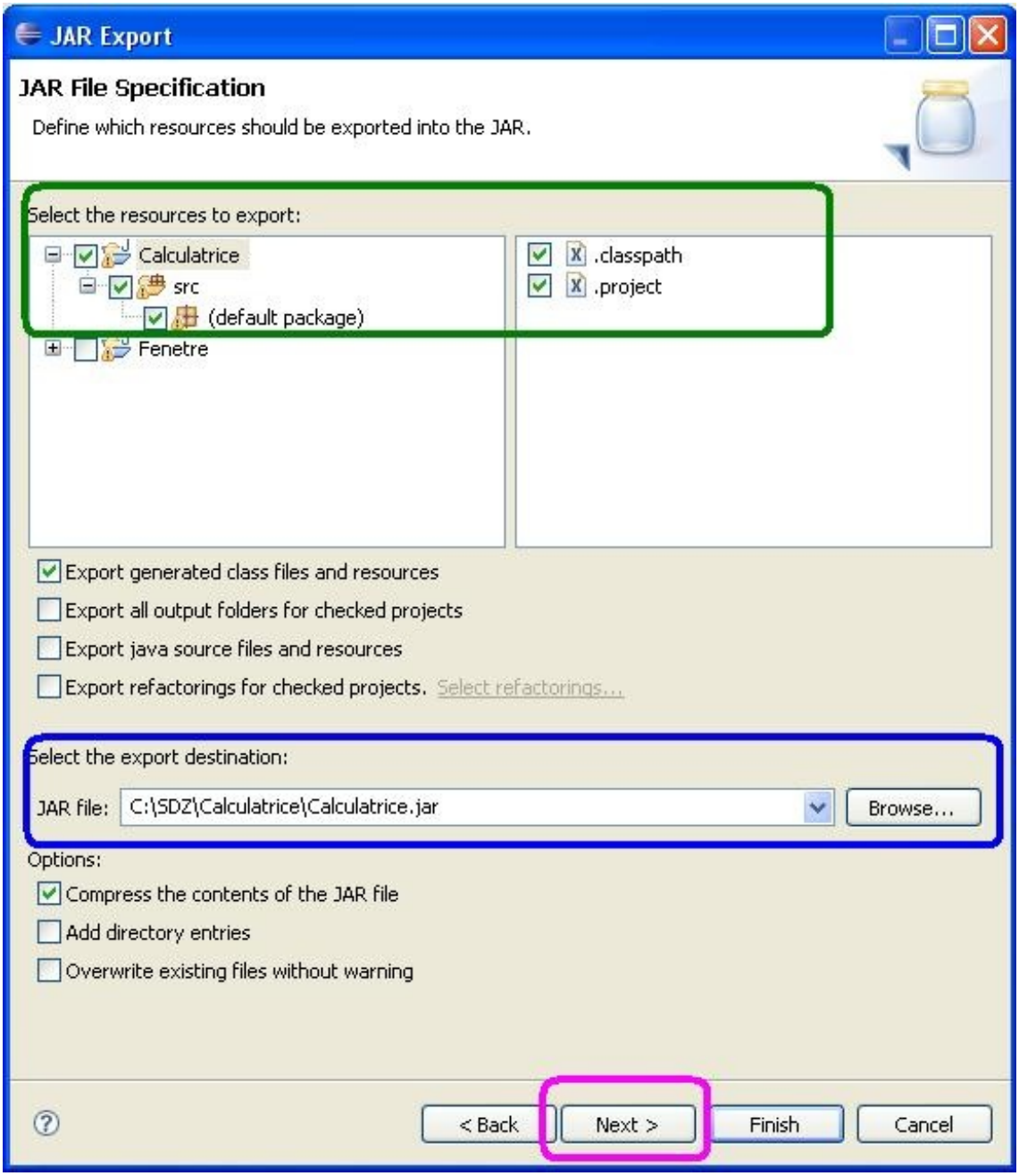
Vous allez voir que créer un **.jar** est très simple !
Commencez par faire un clic droit sur votre projet et choisissez l'option Export, comme ceci :



Vous voici dans la gestion des exports. Eclipse vous demande quel type d'export vous souhaitez faire :



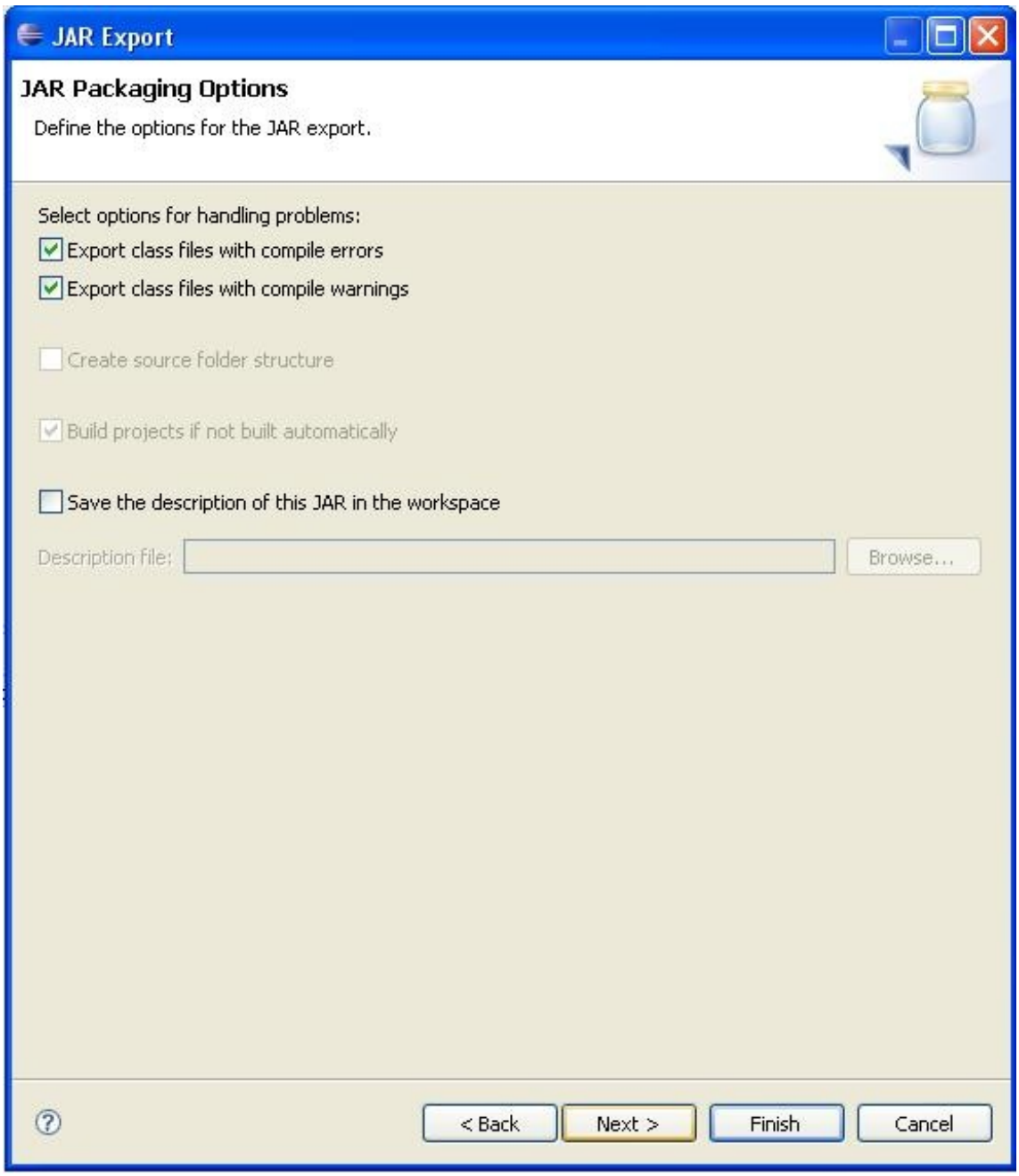
Comme sur l'image ci-dessus, ssélectionnez JAR File et cliquez sur Next. Vous voici maintenant dans le section qui vous demande quels fichiers vous voulez mettre dans votre archive.



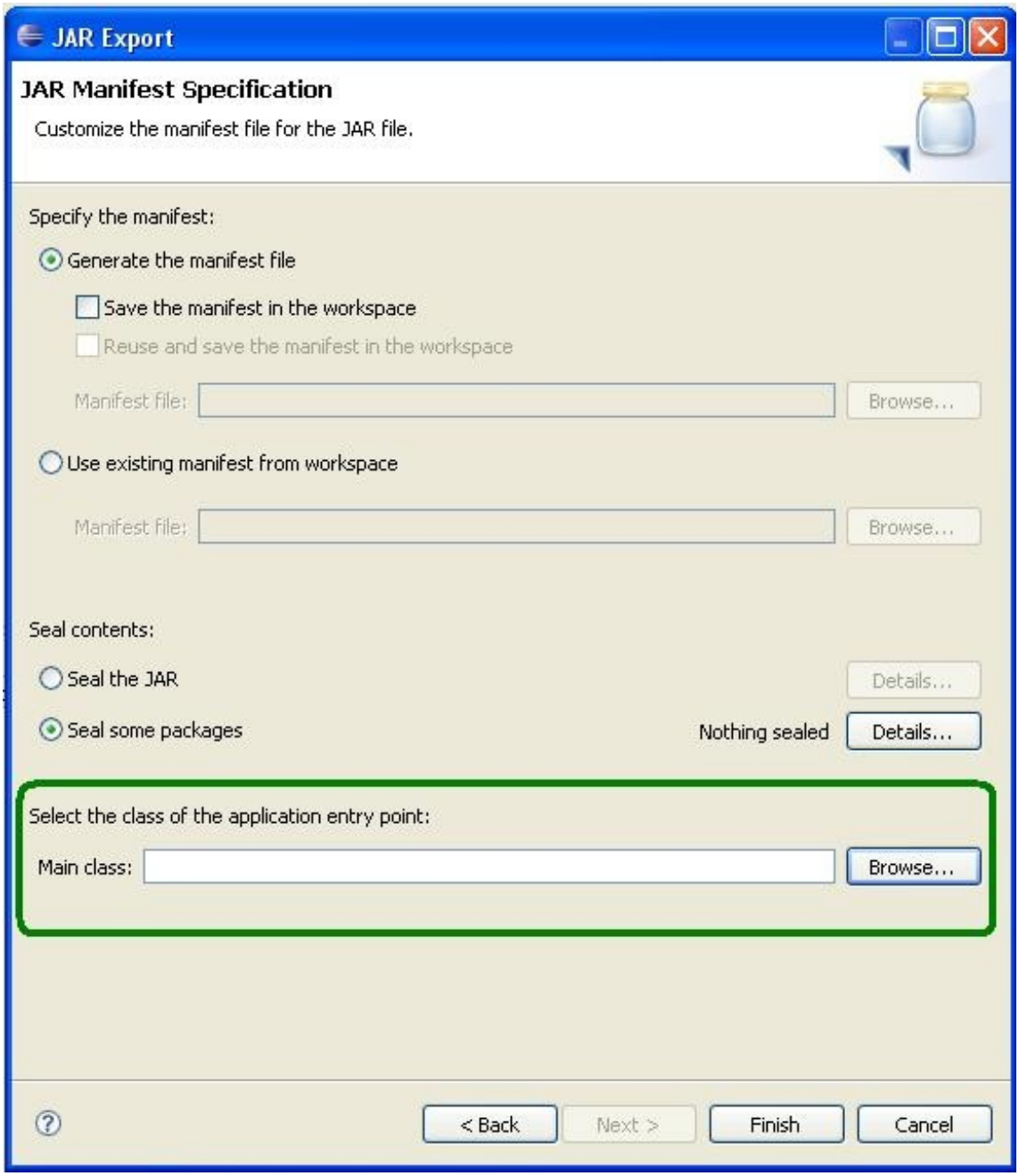
- Dans le cadre vert, nous choisissons tous les fichiers qui composeront notre exécutable .jar.
- Dans le cadre bleu, nous spécifions à eclipse où créer l'archive et quel nom elle doit avoir.
- Ensuite... On clique sur Next.

La page suivante n'est pas pertinente ici. Allez hop ! Circulez, y a rien à voir... (Il manque une émoticône gendarme sur le SDZ...)

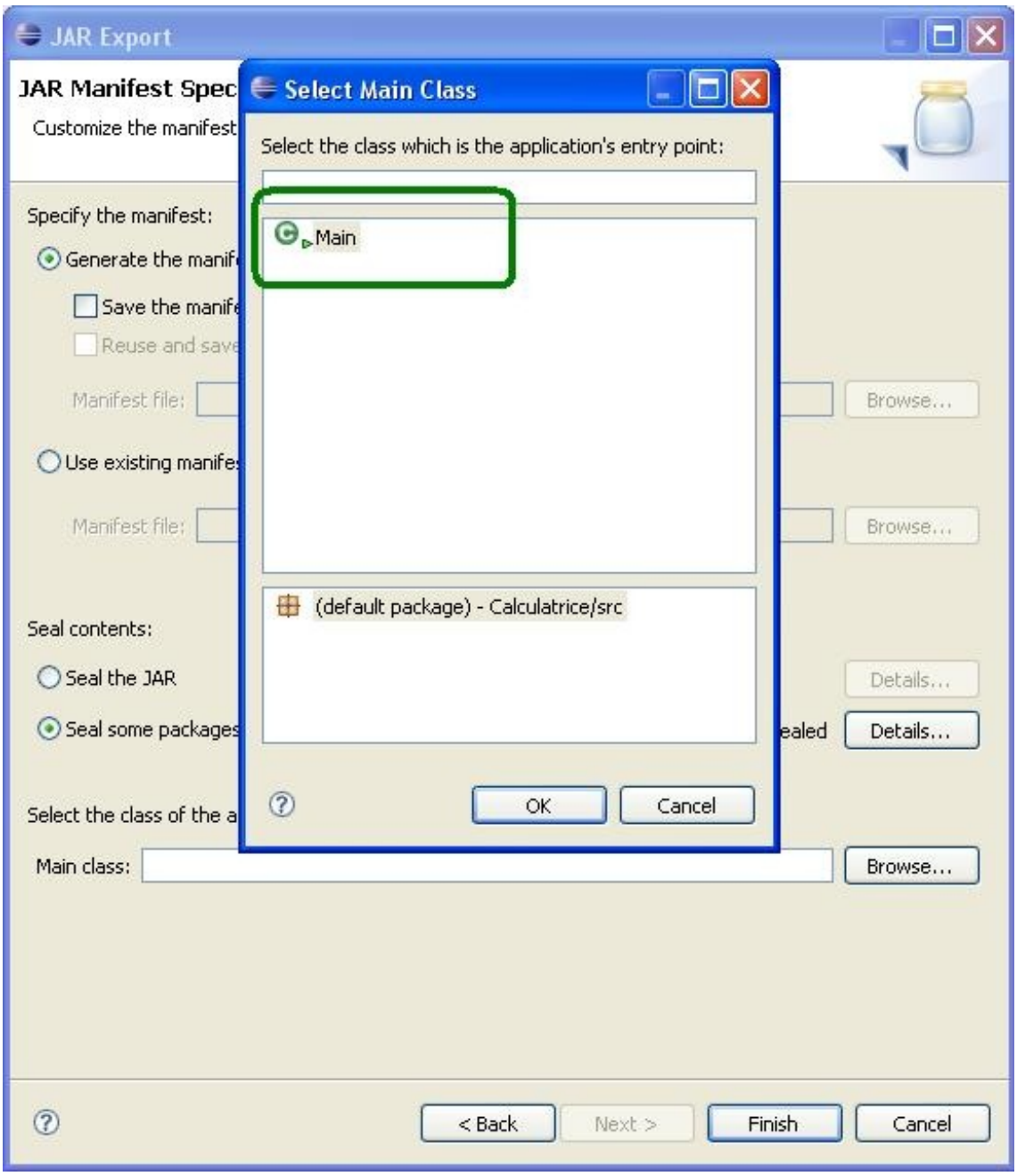
La voici tout de même :



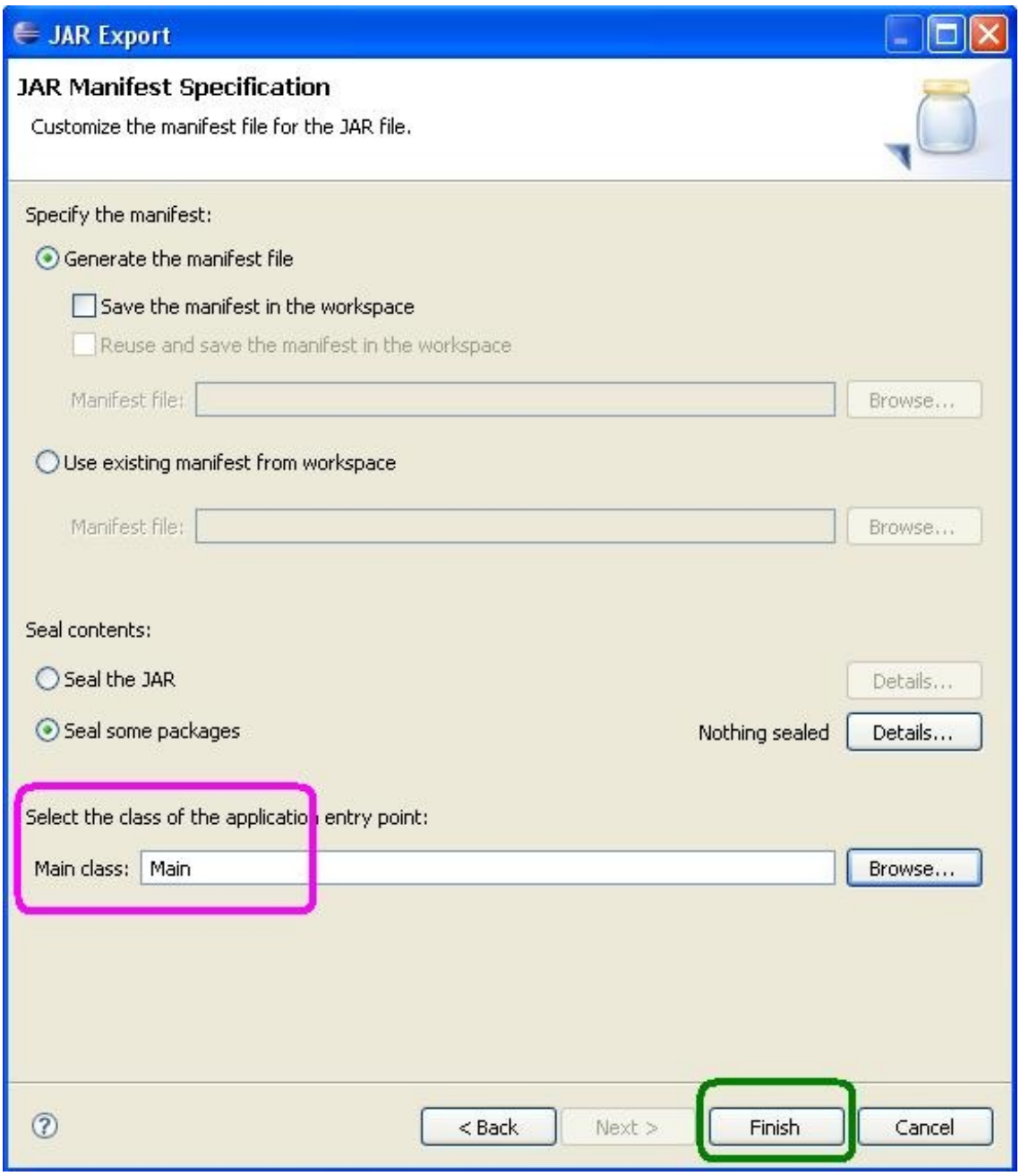
Cliquez sur Next et nous arrivons sur la page qui va nous demander où se trouve la méthode **`main`** dans notre programme :



Cliquez sur Browse... ; vous êtes maintenant sur une petite popup qui liste les fichiers de programmes contenant une méthode `main`. Ici, nous n'en avons qu'un... Mais **il est possible d'avoir plusieurs méthodes `main` déclarées MAIS SEULEMENT UNE SEULE SERA EXÉCUTÉE !**



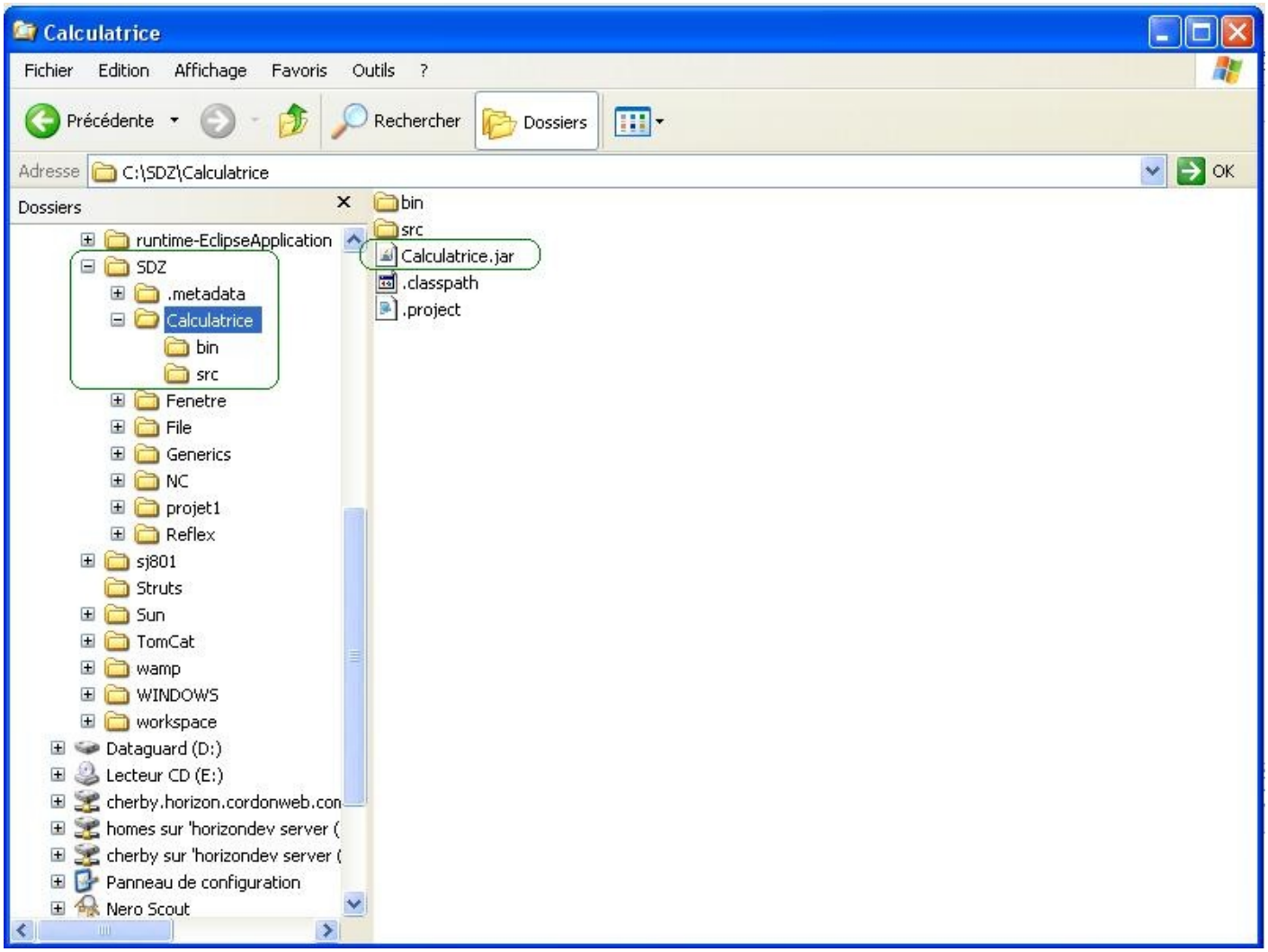
Sélectionnez le point de départ de votre application et validez !
Voici ce que nous avons :



Vous pouvez maintenant cliquez sur Finish et vous devriez avoir un message du genre...

Ce type de message n'est pas alarmant ! Celui-ci vous dit qu'il existe des choses qu'Eclipse ne juge pas très claires... Mais celles-ci n'empêcheront pas votre application de fonctionner... Contrairement à un message d'erreur... Mais là, vous allez le repérer... Il est rouge.

Une fois cette étape validée, vous pouvez voir avec joie, qu'un fichier .jar a bien été généré dans le dossier spécifié !



Double cliquez sur lui et votre calculatrice se lance !

Si vous voulez en savoir plus sur les fichiers .jar, je vous recommande le tuto de bartimeus.
Retour en haut
J'espère que ce TP vous a plu !
Je vous le répète encore une fois, mais mon code est loin d'être parfait et vous pouvez vous y attarder pour l'améliorer !
Par exemple, vous pourriez créer un objet dont la fonction est de faire les calculs...
Vous pourriez aussi faire une calculette avec plus de fonctionnalités ! Enfin, vous voyez quoi...

Bon, continuons notre tuto. Nous en étions restés au problème de `thread` évoqué lorsque nous avions essayé de relancer notre animation.
Je pense que vous avez deviné la suite, en avant pour : **les threads**.
Retour en haut

Les threads

Dans ce chapitre, nous allons voir comment créer une nouvelle pile de fonctions et même plusieurs ; tout ceci avec ce qu'on appelle des threads.

Il y a une classe **`Thread`** dans java qui gère cela, mais vous allez voir qu'il y a en fait deux façons de créer un nouveau thread, et donc une nouvelle pile d'invocation de méthodes !

Ne tardons pas...

Principes et bases

Je vous le répète encore mais, lorsque vous lancez votre programme, il y a un thread lancé ! Imaginez que votre thread corresponde à la pile, et, **pour chaque nouveau thread créé, celui-ci donne une nouvelle pile d'exécution.**

Pour le moment, nous n'allons pas travailler avec notre IHM, nous allons revenir en mode console. Créez-vous un nouveau projet et une classe contenant votre méthode **`main`**. Maintenant, testez ce code :

Code : Java

```
public class Test {

        public static void main(String[] args) {

                System.out.println("Le nom du thread principal est "
+ Thread.currentThread().getName());

        }
}
```

Vous devriez obtenir ceci :

Code : Console

```
Le nom du thread principal est main
```

Oui, vous ne rêvez pas... Il s'agit bien de notre méthode **`main`**, c'est le thread principal de notre application !
Voyez les threads comme une machine bien huilée capable d'effectuer les tâches que vous lui spécifierez. Une fois instancié, un thread attend son lancement ; une fois celui-ci fait, il invoque sa méthode **`run()`** ; c'est dans cette méthode que le thread connaît les tâches qu'il a à faire !

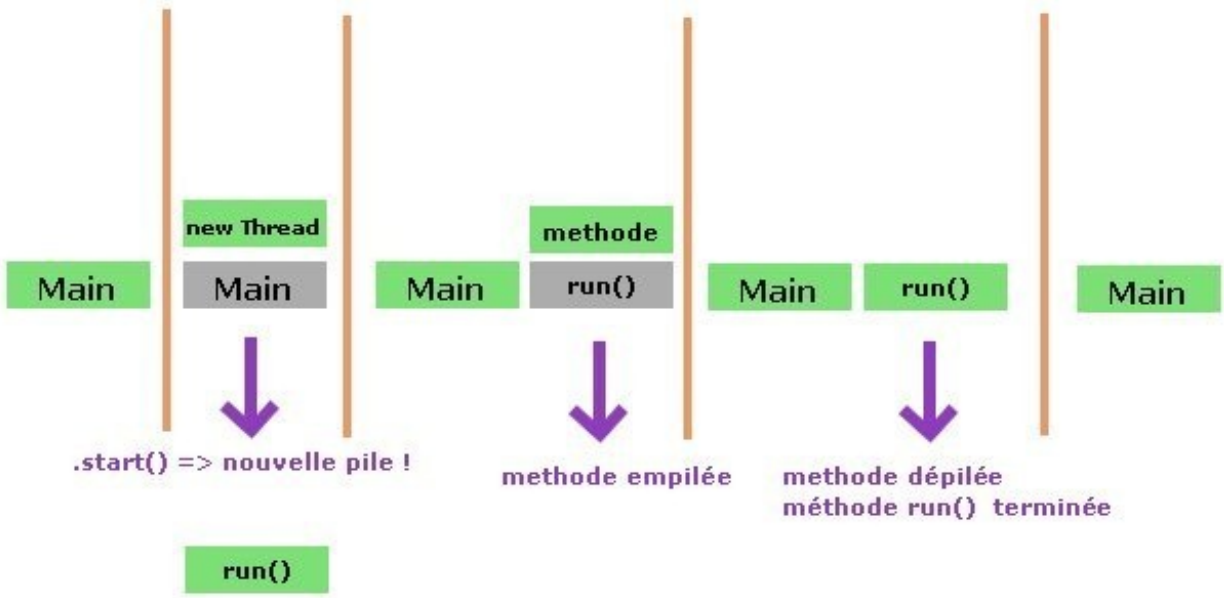
Nous allons maintenant voir comment créer un nouveau thread.
Comme je vous l'ai dit dans l'introduction, il existe deux manières de faire :

- faire une classe héritée de la classe **`Thread`** ;
- créer une implémentation de l'interface **`Runnable`** et instancier un objet **`Thread`** avec une implémentation de cette interface.

Vous devez avoir les sourcils qui se lèvent, là ... un peu comme ça :
Ne vous en faites pas, nous allons y aller *crescendo*...
Retour en haut

Une classe héritée de Thread

Nous allons commencer par le plus simple à comprendre.
Comme je vous le disais, nous allons créer un classe héritée, et tout ce que nous avons à faire, c'est redéfinir la méthode **`run()`** de notre objet afin qu'il sache quoi faire... Vu que nous allons en utiliser plusieurs, autant pouvoir les différencier par un nom...
Créez la classe correspondant à ce diagramme :



On crée ici un constructeur avec un **`String`** en paramètre pour spécifier le nom du thread... Cette classe a une méthode **`getName()`** afin de retourner celui-ci. La classe **`Thread`** se trouve dans la package **`java.lang`**, aucune instruction **`import`** n'est nécessaire !

Voilà le code de cette classe :

Code : Java

```
public class TestThread extends Thread {


        public TestThread(String name){
                super(name);
        }

        public void run(){

                for(int i = 0; i < 10; i++)
                                System.out.println(this.getName());

        }
}
```

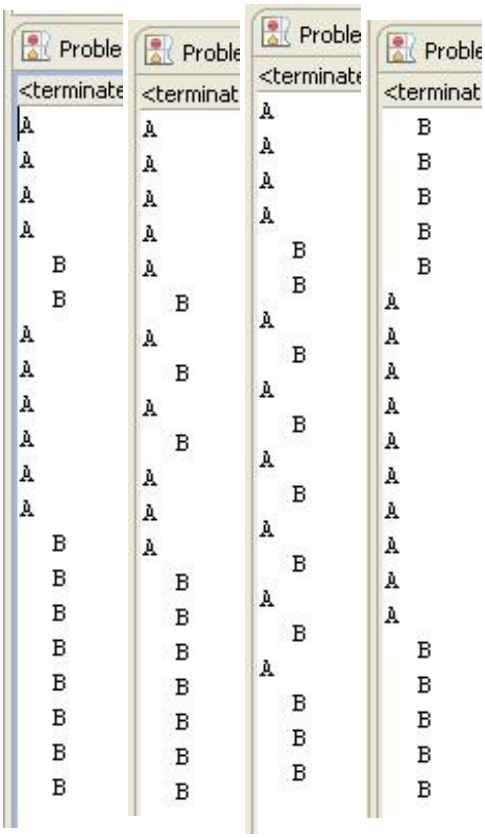
Et maintenant, testez ce code plusieurs fois :

Code : Java

```
public class Test {

        public static void main(String[] args) {
```

```
                TestThread t = new TestThread("A");
                TestThread t2 = new TestThread(" B");
                t.start();
                t2.start();
            }
        }
```

Voici quelques screenshots de mes tests consécutifs :



Vous pouvez voir que l'ordre d'exécution est totalement aléatoire !
Ceci car java utilise un **ordonnanceur**.
Vous devez savoir que si vous utilisez plusieurs threads dans une application, ceux-ci **ne s'exécutent pas en même temps !**
En fait, l'ordonnanceur gère les différents thread de façon aléatoire : il va en utiliser un, pendant un certain laps de temps, puis un autre, puis revenir au premier... Jusqu'à ce que les threads soit terminés ! Et, lorsque l'ordonnanceur passe d'un thread à un autre, le thread interrompu est mis en **sommeil** pendant que l'autre est en **éveil** !

Un thread peut avoir plusieurs états :

- **`NEW`** : lors de sa création.
- **`RUNNABLE`** : lorsque vous invoquez la méthode **`start()`**, celui-ci est prêt à travailler.
- **`TERMINATED`** : lorsque celui-ci a terminé toutes ses tâches, on dit aussi que le thread est **mort**. Une fois un thread mort, vous ne pouvez plus le relancer avec la méthode **`start()`** !
- **`TIMED_WAITING`** : lorsque celui-ci est en pause, quand vous utilisez la méthode **`sleep()`** par exemple.
- **`WAITING`** : en attente indéfinie...
- **`BLOCKED`** : lorsque l'ordonnanceur met un thread en sommeil pour en utiliser un autre... Le statut de celui en sommeil est celui-ci.

Un thread est considéré comme terminé lorsque la méthode `run()` est dépilée de sa pile d'exécution !

En effet, une nouvelle pile d'exécution a, à sa base, la méthode **`run()`** de notre thread... Une fois celle-ci dépilée, notre nouvelle pile est détruite !

Notre thread principal crée un second thread, celui-ci se lance et crée une pile avec comme base sa méthode **`run()`** ; celle-ci appelle **`methode`**, l'empile, fait tous les traitements, et, une fois terminé, dépile cette dernière. La méthode **`run()`** prend fin, la pile est détruite !

Nous allons un peu modifier notre classe **`TestThread`** afin de voir les états de nos threads que nous pouvons récupérer grâce à la méthode **`getState()`**.

Voici notre classe `TestThread` modifiée :

Code : Java

```
public class TestThread extends Thread {

        Thread t;

        public TestThread(String name){
                super(name);
                System.out.println("statut du thread " + name + " =
" +this.getState());
                this.start();
                System.out.println("statut du thread " + name + " =
" +this.getState());
        }

        public TestThread(String name, Thread t){
                super(name);
                this.t = t;
                System.out.println("statut du thread " + name + " =
" +this.getState());
                this.start();
                System.out.println("statut du thread " + name + " =
" +this.getState());
        }



        public void run(){
                for(int i = 0; i < 10; i++){
                        System.out.println("statut " +
this.getName() + " = " +this.getState());
                        if(t != null)System.out.println("statut de "
+ t.getName() + " pendant le thread " + this.getName() +" = "
+t.getState());
                }
        }

        public void setThread(Thread t){
                this.t = t;
        }

}
```

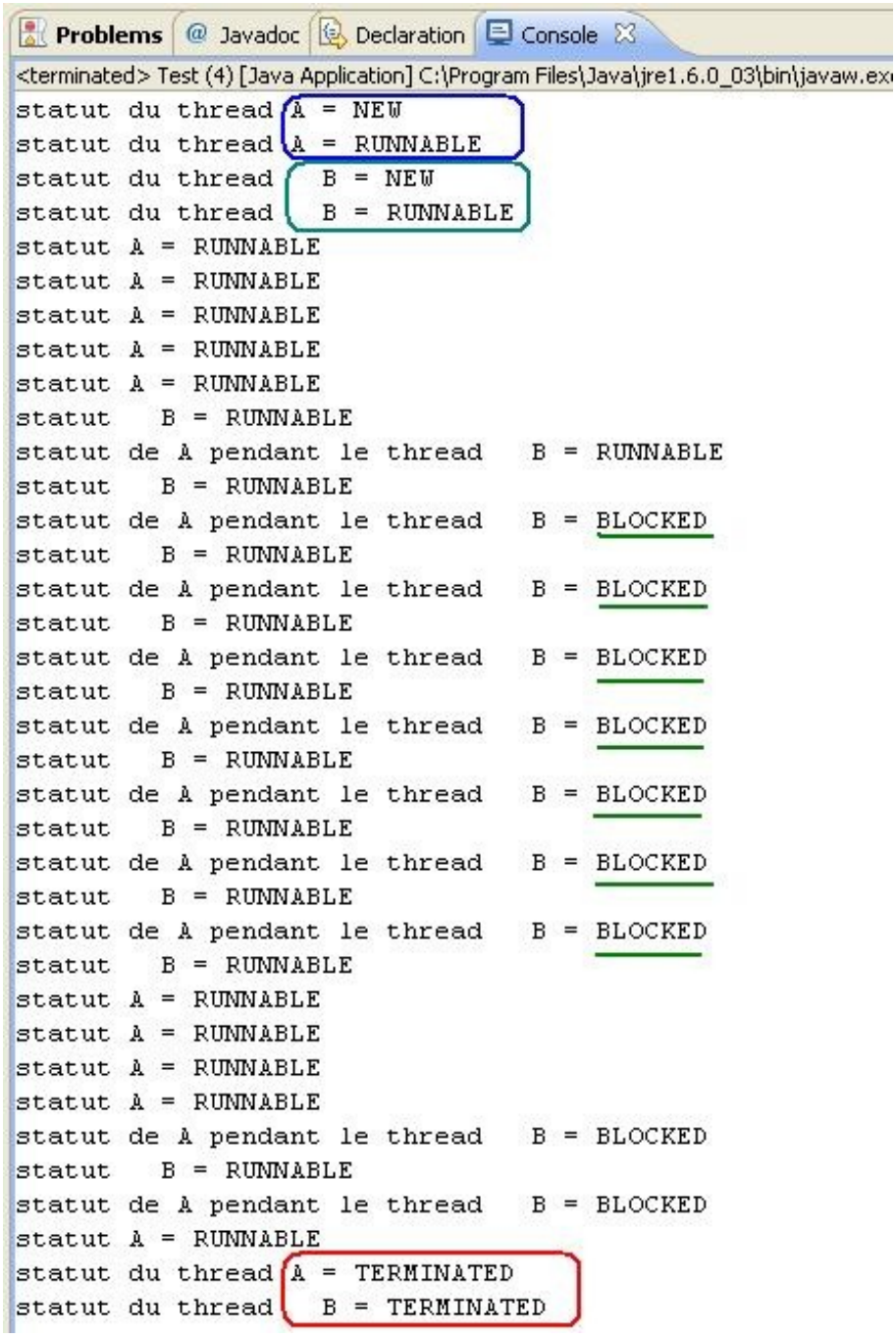
Ainsi que notre `main` :

Code : Java

```
public class Test {

        public static void main(String[] args) {

                TestThread t = new TestThread("A");
                TestThread t2 = new TestThread(" B", t);

                try {
```

```
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                    System.out.println("statut du thread " + t.getName()
+ " = " + t.getState());
                    System.out.println("statut du thread " +
t2.getName() + " = " +t2.getState());

            }
}
```

Et un jeu d'essai représentatif :



Alors, dans notre classe **`TestThread`**, nous avons ajouté quelques instructions d'affichage afin de voir l'état en cours de nos objets, mais nous avons aussi ajouté un constructeur avec un **`Thread`** en paramètre supplémentaire, ceci afin de voir l'état de

notre premier thread lors de l'exécution du second !

Dans notre jeu d'essai vous pouvez voir les différents statuts qu'ont pris nos threads... Et vous pouvez voir que le premier est `BLOCKED` lorsque le second est en cours de traitement, ce qui justifie ce dont je vous parlais :

les threads ne s'exécutent pas en même temps !

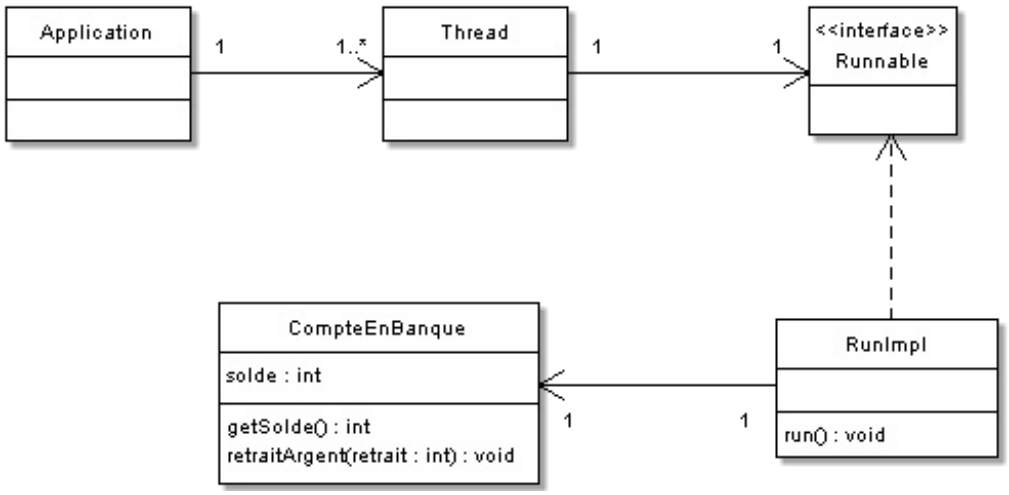
Vous pouvez voir aussi que les traitements effectués par nos threads sont en fait codés dans la méthode `run()`. Reprenez l'image que j'ai utilisée :
"***un thread est une machine bien huilée capable d'effectuer les tâches que vous lui spécifierez*** ".
Le fait de faire un objet hérité de `Thread` permet de créer un nouveau thread très facilement. Cependant, vous pouvez procéder autrement, en redéfinissant uniquement ce que doit faire le nouveau thread, ceci grâce à l'interface `Runnable`. Et dans ce cas, ma métaphore prend tout son sens :
vous ne redéfinissez que ce que doit faire la machine et non pas la machine tout entière !
Retour en haut

Utiliser l'interface Runnable

Le fait de ne redéfinir que ce que doit faire notre nouveau thread a aussi un autre avantage... Le fait d'avoir une classe qui n'hérite d'aucune autre ! Eh oui : dans notre précédent test, notre classe **TestThread** ne pourra plus jamais hériter d'une classe ! Tandis qu'avec une implémentation de **Runnable**, rien n'empêche votre classe d'hériter de JFrame, par exemple...

Trêve de bavardages : codons notre implémentation de **Runnable** ; vous ne devriez avoir aucun problème à faire ceci sachant qu'il n'y a que la méthode **run()** à redéfinir...

Pour cet exemple, nous allons utiliser un exemple que j'ai trouvé intéressant lorsque j'ai appris à me servir des threads...
Vous allez créer un objet **CompteEnBanque** avec une somme d'argent par défaut, disons 50, et une méthode pour retirer de l'argent (**retraitArgent**) et une méthode qui retourne le solde (**getSolde**).
Mais avant de retirer de l'argent, nous irons vérifier que nous ne sommes pas à découvert...
Notre thread va faire autant d'opérations que nous le souhaitons. Voici un petit diagramme de classe résumant la situation :



Je résume.

- Notre application peut avoir 1 ou plusieurs objets **Thread**.
- Ceux-ci ne peuvent avoir qu'un objet de type **Runnable**.
- Dans notre cas, nos objets Thread auront une implémentation de **Runnable** : **RunImpl**.
- Celui-ci à un objet **CompteEnBanque**.

Voici les codes source :

RunImpl.java

Code : Java

```
public class RunImpl implements Runnable {

        private CompteEnBanque cb;

        public RunImpl(CompteEnBanque cb){
                this.cb = cb;
        }

        public void run() {

                for(int i = 0; i < 25; i++){

                        if(cb.getSolde() > 0){
                                cb.retraitArgent(2);
                                System.out.println("Retrait effectué
");
```

```
                        }
                }
        }
}
```

CompteEnBanque.java

Code : Java

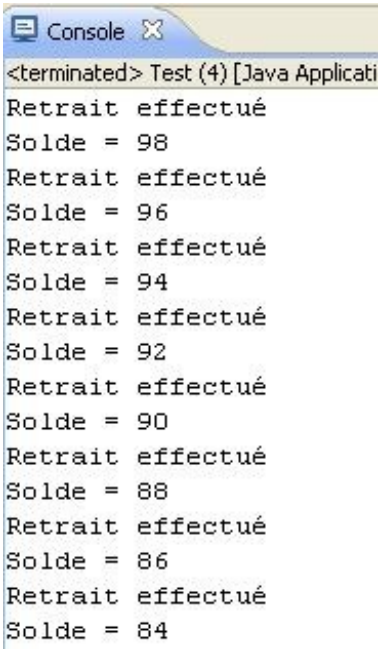
```
public class CompteEnBanque {

        private int solde = 100;

        public int getSolde(){
                if(this.solde < 0)
                        System.out.println("Vous êtes à découvert
!");

                return this.solde;
        }

        public void retraitArgent(int retrait){
                solde = solde - retrait;
                System.out.println("Solde = " + solde);
        }
}
```

Test.java

Code : Java

```
public class Test {

        public static void main(String[] args) {

                CompteEnBanque cb = new CompteEnBanque();

                Thread t = new Thread(new RunImpl(cb));
                t.start();
        }
}
```

Ce qui nous donne :



Rien d'extraordinaire ici... Une simple boucle aurait fait la même chose...
Ajoutons un nom à notre implémentation et créez un deuxième thread utilisant un deuxième compte.
Pensez à modifier votre implémentation afin que nous puissions voir sur quel thread nous sommes.
Bon : je suis sympa, voici les codes :

Code : Java

```
public class RunImpl implements Runnable {

        private CompteEnBanque cb;
        private String name;

        public RunImpl(CompteEnBanque cb, String name){
                this.cb = cb;
                this.name = name;
        }

        public void run() {

                for(int i = 0; i < 50; i++){

                        if(cb.getSolde() > 0){
                                cb.retraitArgent(2);
                                System.out.println("Retrait effectué
par " + this.name);
                        }
                }
        }

}
```

Code : Java

```
public class Test {

        public static void main(String[] args) {

                CompteEnBanque cb = new CompteEnBanque();
                CompteEnBanque cb2 = new CompteEnBanque();

                Thread t = new Thread(new RunImpl(cb, "Cysboy"));
```

```
                Thread t2 = new Thread(new RunImpl(cb2, "ZérO"));
                t.start();
                t2.start();
        }
}
```

Pour vérifier que nos threads fonctionnent, voici une partie de mon résultat :

```
Retrait effectué par Cysboy
Solde = 92
Retrait effectué par Cysboy
Solde = 90
Solde = 86
Retrait effectué par ZérO
Solde = 84
Retrait effectué par ZérO
Retrait effectué par Cysboy
Solde = 88
Solde = 82
Retrait effectué par ZérO
Solde = 80
Retrait effectué par ZérO
Solde = 78
Retrait effectué par ZérO
Solde = 76
Retrait effectué par ZérO
Solde = 74
Retrait effectué par ZérO
Solde = 72
Retrait effectué par ZérO
Solde = 70
Retrait effectué par ZérO
Solde = 68
Retrait effectué par ZérO
Retrait effectué par Cysboy
Solde = 66
Solde = 86
Retrait effectué par ZérO
Retrait effectué par Cysboy
```

Jusqu'ici, rien de perturbant... Nous avons utilisé deux instances distinctes de `RunImpl` utilisant deux instances distinctes de `CompteEnBanque`.
Mais d'après vous, que ce passerait-il si nous utilisions le même instance de `CompteEnBanque` dans deux threads différents ? Essayez plusieurs fois ce code :

Code : Java

```
public class Test {

        public static void main(String[] args) {

                CompteEnBanque cb = new CompteEnBanque();

                Thread t = new Thread(new RunImpl(cb, "Cysboy"));
                Thread t2 = new Thread(new RunImpl(cb, "ZérO"));
                t.start();
                t2.start();
```

```
            }
        }
```

Voici juste deux morceaux de résultats obtenus lors de l'exécution :

```
Retrait effectué par Cysboy
Solde = 48
Retrait effectué par Cysboy
Solde = 46
Retrait effectué par Cysboy
Solde = 50
Solde = 44
Retrait effectué par Cysboy
Solde = 40
Retrait effectué par Cysboy
Solde = 38
```

```
Solde = 64
Retrait effectué par ZérO
Solde = 62
Retrait effectué par ZérO
Retrait effectué par Cysboy
Solde = 60
Solde = 60
Retrait effectué par Cysboy
Retrait effectué par ZérO
Solde = 56
Solde = 56
Retrait effectué par Cysboy
Retrait effectué par ZérO
Solde = 52
Retrait effectué par Cysboy
```

Vous pouvez voir des incohérences monumentales !
J'imagine que vous avez été comme moi au départ, vous pensiez que le compte aurait été débité de deux en deux jusqu'à la fin, sans avoir ce genre d'abérrations, vu que nous utilisons le même objet... Eh bien non !
Pourquoi ? Tout simplement parce que l'ordonnanceur de java met les threads en sommeil quand il le veut et, lorsque celui qui était en sommeil se réveille, il reprend le travail où il s'était arrêté !

Voyons comment résoudre le problème.

Retour en haut

Synchronisez vos threads

Tout est dans le titre !

En gros, ce qu'il faut faire, c'est prévenir la JVM qu'un thread est en train d'utiliser des données qu'un autre thread est susceptible d'utiliser !

Ainsi, lorsque l'ordonnanceur met un thread en sommeil et que celui-ci traitait des données utilisables par un autre thread, ce thread garde la priorité sur les données, et tant que celui-ci n'a pas terminé son travail, les autres threads n'ont pas la possibilité d'y toucher.

Ceci s'appelle synchroniser les threads.

Comment fait-on ça ? Je sens que ça va être encore un truc tordu !

Cette opération est très délicate et demande beaucoup de compétences en programmation...
Voici à quoi ressemble votre méthode **`retraitArgent`** synchronisée :

Code : Java

```
public class CompteEnBanque {

        private int solde = 100;

        public int getSolde(){
                if(this.solde < 0)
                        System.out.println("Vous êtes à découvert
!");

                return this.solde;
        }

        public synchronized void retraitArgent(int retrait){
                solde = solde - retrait;
                System.out.println("Solde = " + solde);
        }
}
```

Il vous suffit d'ajouter le mot clé **`synchronized`** dans la déclaration de votre méthode !
Grâce à ce mot clé, cette méthode est inaccessible à un thread si celle-ci est déjà utilisée par un autre thread ! Les threads cherchant à utiliser des méthodes déjà prises en charge par un autre thread sont mises dans une "*liste d'attente*".

Je récapitule encore une fois, voici un contexte ludique.
Je serai représenté par le thread A, vous par le thread B et notre boulangerie favorite par la méthode synchronisée M. Voici ce qu'il se passe :

- le thread A (moi) appelle la méthode M ;
- je commence par demander une baguette, la boulangère me la pose sur le comptoir et commence à calculer le montant ;
- c'est là que le thread B (vous) cherche aussi à utiliser la méthode M, cependant, elle est déjà prise par un thread (moi...) ;
- vous êtes donc mis en attente ;
- l'action revient sur moi (thread A) ; au moment de payer, je dois chercher de la monnaie dans ma poche...
- au bout de quelques instant, je m'endors...
- l'action revient sur le thread B (vous)... Mais la méthode M n'est toujours pas libérée du thread A... Remise en attente ;
- on revient sur le thread A qui arrive enfin à payer et à quitter la boulangerie, la méthode M est libérée !
- le thread B (vous) peut enfin utiliser la méthode M ;
- et là, les threads C, D, E, F, G, H, I, J entrent dans la boulangerie...
- etc.

Je pense qu'avec ceci vous avez dû comprendre...

Dans un contexte informatique, il peut être pratique et sécurisé d'utiliser des threads et des méthodes synchronisées lors d'accès à des services distants tel qu'un serveur d'application, ou encore un SGBD...
Les threads, pour soulager le thread principal et ne pas bloquer l'application pendant une tâche et des méthodes synchronisées, pour la sécurité et l'intégrité des données !

Je vous propose maintenant de retourner à notre animation qui n'attend qu'un petit thread pour pouvoir fonctionner correctement !

Retour en haut

Contrôlez votre animation

À partir de là, il n'y a rien de bien compliqué...
Il nous suffit de créer un nouveau thread lorsqu'on clique sur le bouton Go en lui passant une implémentation de **Runnable** qui, elle, va appeler la méthode **go()** (ne pas oublier de remettre le booléen de controle à **true**).

Pour l'implémentation de l'interface **Runnable**, une classe interne est toute indiquée !

Voici le code de notre classe **Fenetre** avec le thread :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class Fenetre extends JFrame{

        private Panneau pan = new Panneau();
        private JButton bouton = new JButton("Go");
        private JButton bouton2 = new JButton("Stop");
    private JPanel container = new JPanel();
    private JLabel label = new JLabel("Le JLabel");
    private int compteur = 0;
    private boolean animated = true;
    private boolean backX, backY;
    private int x,y ;
    private Thread t;

    public Fenetre(){

            this.setTitle("Animation");
            this.setSize(300, 300);
            this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            this.setLocationRelativeTo(null);

            container.setBackground(Color.white);
            container.setLayout(new BorderLayout());
            container.add(pan, BorderLayout.CENTER);

            //Ce sont maintenant nos classes internes qui écoutent
nos boutons
            bouton.addActionListener(new BoutonListener());

            bouton2.addActionListener(new Bouton2Listener());
            bouton2.setEnabled(false);

            JPanel south = new JPanel();
            south.add(bouton);
            south.add(bouton2);
            container.add(south, BorderLayout.SOUTH);

            Font police = new Font("Tahoma", Font.BOLD, 16 );
            label.setFont(police);
            label.setForeground(Color.blue);
            label.setHorizontalAlignment(JLabel.CENTER);

            container.add(label, BorderLayout.NORTH);
```

```
                this.setContentPane(container);
                this.setVisible(true);

        }

            private void go(){
            //Les coordonnées de départ de notre rond
                    x = pan.getPosX();
                    y = pan.getPosY();
            //Pour cet exemple, j'utilise une boucle while
            //Vous verrez qu'elle marche très bien
            while(this.animated){

                    if(x < 1)backX = false;
                if(x > pan.getWidth()-50)backX = true;
                if(y < 1)backY = false;
                if(y > pan.getHeight()-50)backY = true;


                    if(!backX)pan.setPosX(++x);
                else pan.setPosX(--x);
                if(!backY) pan.setPosY(++y);
                else pan.setPosY(--y);
                pan.repaint();

                try {
                        Thread.sleep(3);
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
            }
            }


            /**
    * classe qui écoute notre bouton
    */
            public class BoutonListener implements ActionListener{

                    /**
    * Redéfinitions de la méthode actionPerformed
    */
                    public void actionPerformed(ActionEvent arg0) {
                            animated = true;
                            t = new Thread(new PlayAnimation());
                            t.start();
                            bouton.setEnabled(false);
                            bouton2.setEnabled(true);

                    }

            }

            /**
    * classe qui écoute notre bouton2
    */
            class Bouton2Listener  implements ActionListener{

                    /**
    * Redéfinitions de la méthode actionPerformed
    */
                    public void actionPerformed(ActionEvent e) {
                            animated = false;
                            bouton.setEnabled(true);
                            bouton2.setEnabled(false);
                    }

            }
```

```
        class PlayAnimation implements Runnable{

                @Override
                public void run() {
                        go();
                }
        }
}
```

Vous pouvez tester et tester encore, ce code fonctionne très bien ! Vous avez enfin le contrôle sur votre animation !

Ceci fait, nous pouvons allez faire un tour sur le topo, et je crois qu'un QCM vous attend...
Retour en haut

Ce qu'il faut retenir

- **Un nouveau thread permet de créer une nouvelle pile d'exécution**.
- La classe `Thread` et l'interface `Runnable` se trouvent dans le package `java.lang`, donc aucun import spécifique est nécessaire pour leur utilisation.
- Un thread se lance lorsqu'on invoque la méthode `start()`.
- Cette dernière invoque automatiquement la méthode `run()`.
- Les traitements que vous souhaitez mettre dans une autre pile d'exécution sont à mettre dans la méthode `run()`, qu'il s'agisse d'une classe héritée de `Thread` ou d'une implémentation de `Runnable`.
- Pour protéger l'intégrité de vos données accessibles à plusieurs threads, utilisez le mot clé `synchronized` dans la déclaration de vos méthodes.
- **Un thread est déclaré mort lorsque celui-ci a dépilé sa méthode `run()` de sa pile d'exécution !**
- Les threads peuvent avoir plusieurs états : `BLOCKED`, `NEW`, `TERMINATED`, `WAITING`, `RUNNABLE` et `TIMED_WATING`.

Retour en haut

Voilà encore un gros chapitre, et très important qui plus est !
Prenez le temps de bien assimiler les choses et de faire des tests, plein de tests : c'est la meilleure façon de bien comprendre les choses...

Pour ceux qui se sentent d'attaque, en avant pour : **les listes.**

Retour en haut

Les listes : l'objet JComboBox

Nous allons continuer à explorer les objets que nous propose `swing`.
Ici, nous parlerons des listes de choix ou `JComboBox`, nous pourrons ainsi améliorer notre animation...

Let's go, ZérO boys... and girls !

Première utilisation

Comme à l'accoutumée, nous allons d'abord utiliser cet objet dans un contexte vierge de tout code... Alors créez-vous un projet avec une classe contenant la méthode **main** et une classe qui sera héritée de **JFrame**.

Dans cet exemple, nous allons bien évidemment utiliser une liste, donc créez en une...

Ce type d'objet ne possède pas de constructeur avec un libellé en paramètre, comme les boutons par exemple ! Il faut donc mettre un **JLabel** à côté !

Voici un premier code :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;

import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;


public class Fenetre extends JFrame {

        private JPanel container = new JPanel();
        private JComboBox combo = new JComboBox();
        private JLabel label = new JLabel("Une ComboBox");

        public Fenetre(){

        this.setTitle("Animation");
        this.setSize(300, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());

        JPanel top = new JPanel();
        top.add(label);
        top.add(combo);

        container.add(top, BorderLayout.NORTH);
        this.setContentPane(container);
        this.setVisible(true);
        }

}
```

Et l'aperçu :

Vous constatez que votre objet est ridiculement petit...
Mais vous connaissez le remède !

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;

import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;


public class Fenetre extends JFrame {

        private JPanel container = new JPanel();
        private JComboBox combo = new JComboBox();
        private JLabel label = new JLabel("Une ComboBox");

        public Fenetre(){

        this.setTitle("Animation");
        this.setSize(300, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());

        combo.setPreferredSize(new Dimension(100,20));

        JPanel top = new JPanel();
        top.add(label);
        top.add(combo);

        container.add(top, BorderLayout.NORTH);
        this.setContentPane(container);
        this.setVisible(true);
        }
}
```

Et voilà :

Et comment on renseigne cette liste ?

Nous y voilà ! Pour ça, il suffit d'utiliser la méthode **`addItem(Object obj)`**.

Vous devez savoir que lorsque l'objet affiche les éléments ajoutés, celui-ci appelle la méthode **`toString()`** des objets ajoutés... Dans cet exemple, nous avons utilisé des objets **`String`**, mais essayez avec un autre objet et vous verrez...

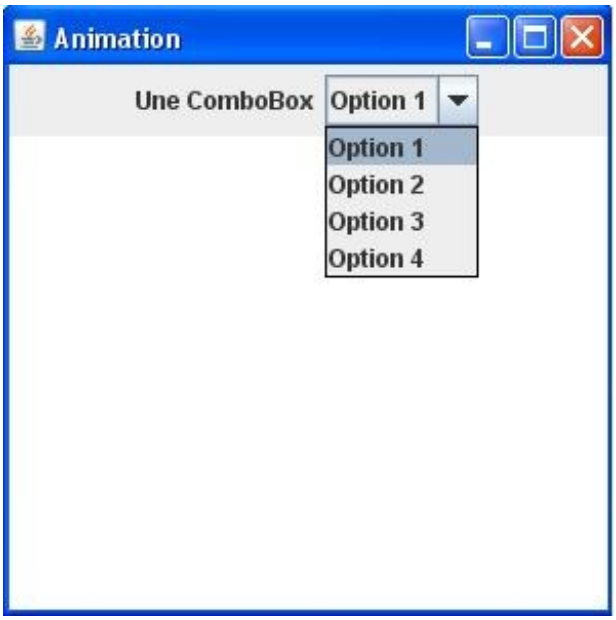
Voici le nouveau code :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;

import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;


public class Fenetre extends JFrame {

        private JPanel container = new JPanel();
        private JComboBox combo = new JComboBox();
        private JLabel label = new JLabel("Une ComboBox");

        public Fenetre(){

        this.setTitle("Animation");
        this.setSize(300, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());

        combo.setPreferredSize(new Dimension(100,20));
        combo.addItem("Option 1");
```

```
                combo.addItem("Option 2");
                combo.addItem("Option 3");
                combo.addItem("Option 4");

                JPanel top = new JPanel();
                top.add(label);
                top.add(combo);

                container.add(top, BorderLayout.NORTH);
                this.setContentPane(container);
                this.setVisible(true);
                }
        }
```

Et le résultat :



Vous pouvez utiliser le constructeur qui prend un tableau d'objets en paramètre afin de renseigner tous les éléments d'un coup !

Ceci est donc équivalent :

Code : Java

```
String[] tab = {"Option 1", "Option 2", "Option 3", "Option 4"};
combo = new JComboBox(tab);
```

Vous pouvez présélectionner un choix avec la méthode **`setSelectedIndex(int index)`**. Vous avez aussi la possibilité de changer la couleur d'écriture, la couleur de fond, la police exactement comme un **`JLabel`**.

Maintenant que vous voyez comment fonctionne cet objet, nous allons voir comment communiquer avec lui.
Retour en haut

L'interface ItemListener

Cette interface à une méthode à redéfinir. Celle-ci est appelée lorsqu'un élément a changé d'état !
Vu qu'un exemple est toujours plus éloquent, voici un code implémentant cette interface :

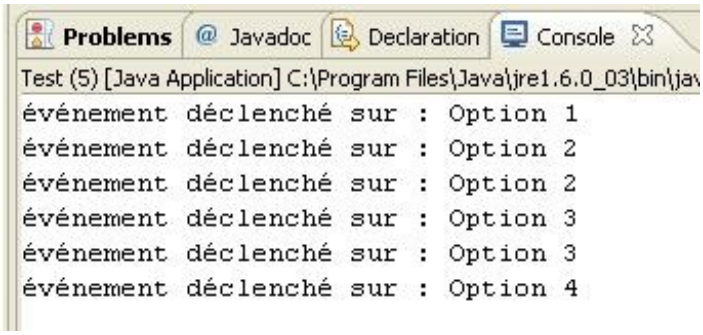
Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;

import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;


public class Fenetre extends JFrame {

        private JPanel container = new JPanel();
        private JComboBox combo = new JComboBox();
        private JLabel label = new JLabel("Une ComboBox");

        public Fenetre(){

        this.setTitle("Animation");
        this.setSize(300, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());

        String[] tab = {"Option 1", "Option 2", "Option 3", "Option
4"};

        combo = new JComboBox(tab);
        //Ajout du listener
        combo.addItemListener(new ItemState());
        combo.setPreferredSize(new Dimension(100,20));
        combo.setForeground(Color.blue);

        JPanel top = new JPanel();
        top.add(label);
        top.add(combo);

        container.add(top, BorderLayout.NORTH);
        this.setContentPane(container);
        this.setVisible(true);
        }

        /**
* Classe interne implémentant l'interface ItemListener
*/
        class ItemState implements ItemListener{

                public void itemStateChanged(ItemEvent e) {
                        System.out.println("événement déclenché sur
: " + e.getItem());
                }
        }

}
```

Dans mon exemple, j'ai cliqué sur :

- Option 2, puis
- Option 3, puis
- Option 4.

Ce qui me donne :



Vous voyez bien que lorsque vous cliquez sur une option non sélectionnée, notre objet change d'abord l'état de l'option précédente (l'état passe en **DESELECTED**) avant de changer l'état de l'option choisie (celle-ci passe à l'état **SELECTED**).
Vous pouvez donc suivre très facilement l'état de vos éléments grâce à cette interface ; cependant, pour plus de simplicité, nous utiliserons l'interface **ActionListener** pour récupérer l'option sélectionnée !

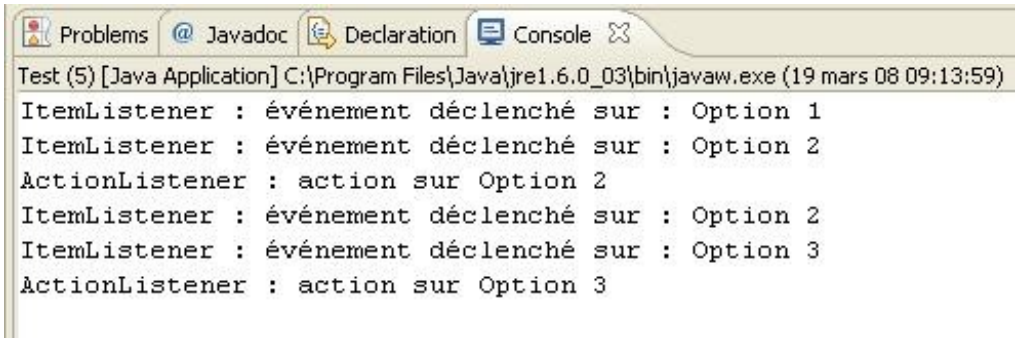
Voici un code implémentant cette interface :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;

import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;


public class Fenetre extends JFrame {

        private JPanel container = new JPanel();
        private JComboBox combo = new JComboBox();
        private JLabel label = new JLabel("Une ComboBox");

        public Fenetre(){

        this.setTitle("Animation");
        this.setSize(300, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());

        String[] tab = {"Option 1", "Option 2", "Option 3", "Option
4"};

        combo = new JComboBox(tab);
        //Ajout du listener
        combo.addItemListener(new ItemState());
        combo.addActionListener(new ItemAction());
```

```
            combo.setPreferredSize(new Dimension(100,20));
            combo.setForeground(Color.blue);

            JPanel top = new JPanel();
            top.add(label);
            top.add(combo);

            container.add(top, BorderLayout.NORTH);
            this.setContentPane(container);
            this.setVisible(true);
            }

            /**
    * Classe interne implémentant l'interface ItemListener
    */
            class ItemState implements ItemListener{

                    public void itemStateChanged(ItemEvent e) {
                            System.out.println("ItemListener : événement
    déclenché sur : " + e.getItem());
                    }
            }

            class ItemAction implements ActionListener{

                    public void actionPerformed(ActionEvent e) {
                            System.out.println("ActionListener : action
    sur " + combo.getSelectedItem());
                    }
            }

    }
```

Et le résultat :



Vous constatez qu'avec une telle implémentation, nous pouvons récupérer l'option sur laquelle l'action a été produite. L'appel de la méthode **`getSelectedItem()`** retourne la valeur de l'option sélectionnée ; une fois ceci récupérée, nous allons pouvoir travailler avec notre liste !

Il y a d'autres méthodes pour récupérer le contenu d'une liste... Je vous invite à chercher ce genre d'information.
En effet, une information apprise par nos propres moyens s'oublie beaucoup moins vite que celle qu'on vous sert toute cuite sur un plateau...

Maintenant que nous savons comment récupérer les informations dans une liste, je vous invite à continuer notre animation.
Retour en haut

Changer la forme de notre animation

Comme le titre l'indique, nous allons faire en sorte que notre animation ne se contente plus d'afficher un rond... Nous allons pouvoir choisir quelle forme nous voulons afficher !

Bien sûr, je ne vais pas vous faire réaliser toutes les formes possibles et imaginables... Je vous en fournis quelques-unes et, si le coeur vous en dit, ajoutez-en de votre composition !

Très bien : pour arriver à faire ceci, nous allons dynamiser un peu notre classe **`Panneau`**, celle-ci devra pouvoir peindre différentes forme selon notre bon vouloir.
Pour y parvenir, nous allons ajouter une variable d'instance de type **`String`** qui contiendra l'intitulé de la forme que nous souhaitons dessiner - appelons-la **`forme`** - et ajoutons un mutateur afin de pouvoir redéfinir cette variable.
Notre méthode **`paintComponent`** doit pouvoir dessiner la forme demandée ; ici, deux cas de figures se profilent :

- soit nous intégrons les instructions **`if`** dans cette méthode et l'objet **`Graphics`** dessinera en fonction,
- soit nous développons une méthode privée, appelée dans la méthode **`paintComponent`**, et qui dessinera la forme demandée.

J'ai l'habitude de procéder de la deuxième façon pour éviter les conditions à rallonges dans mes méthodes...

Faites toutefois attention lorsque vous faites ceci ! Il se peut que votre classe devienne énorme et que vous vous y perdiez ! Nous verrons, dans une partie traitant des ***design patterns***, que lorsque votre code a des portions de code contenant beaucoup de conditions, il vaut mieux programmer une implémentation, mais nous n'en sommes pas encore là...

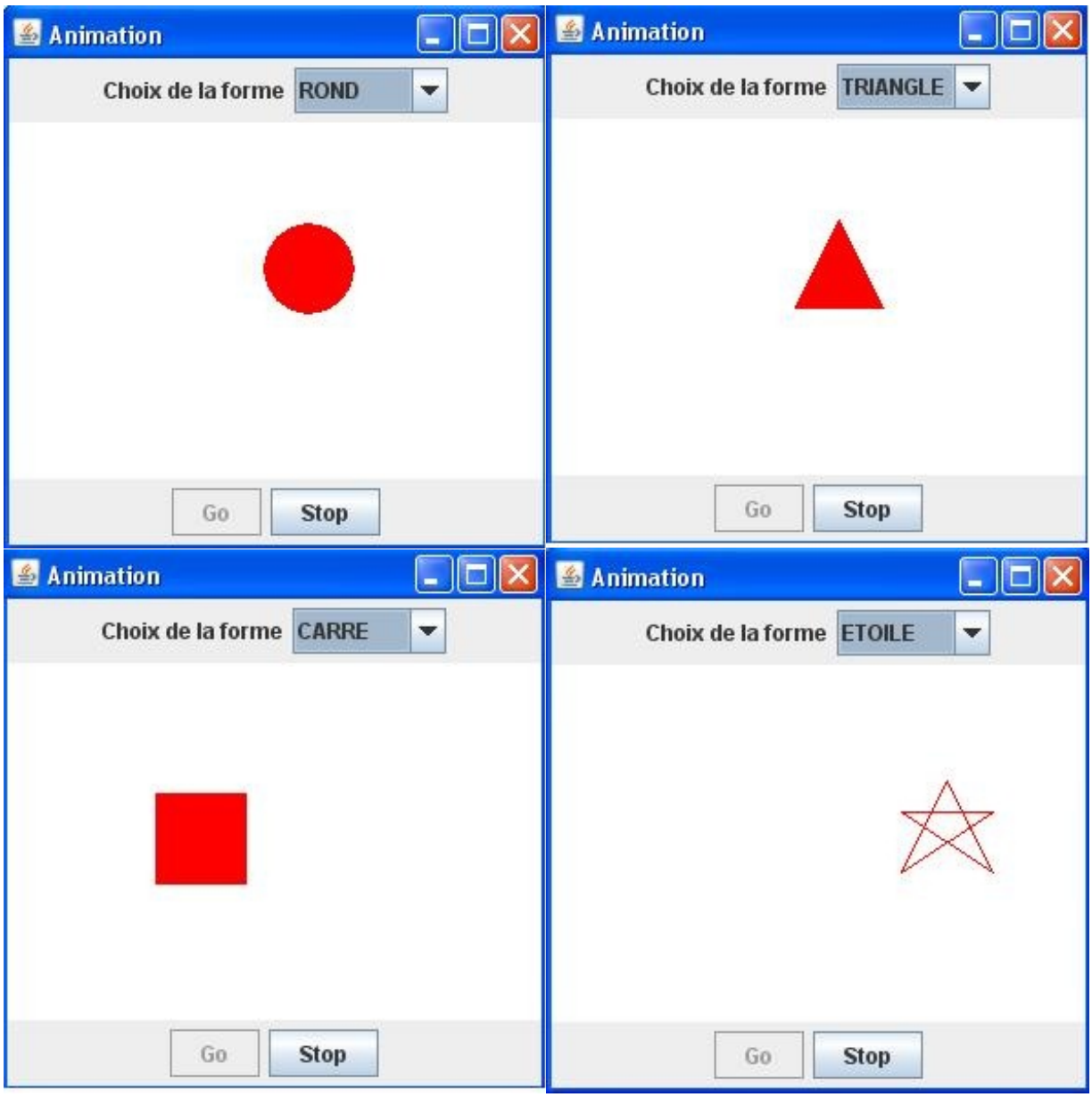
Nous allons donc développer une méthode privée, appelons-la **`draw(Graphics g)`**, qui aura pour tâche de dessiner la forme voulue. Nous passerons l'objet **`Graphics`** utilisé dans la méthode **`paintComponent`** afin qu'elle puisse l'utiliser, et c'est dans cette méthode que nous mettrons nos conditions.

Je vous propose les formes suivantes :

- le rond, forme par défaut,
- le carré,
- le triangle,
- l'étoile (soyons fous).

Ce qui veut dire que notre liste contiendra ces choix, et le rond sera le premier !
Nous créerons aussi une implémentation d'**`ActionListener`** dans une classe interne pour gérer les actions sur notre liste ; je l'ai appelée **`FormeListener`**, c'est fou ce que je suis original...

Une fois n'est pas coutume, voici ce que vous devez obtenir :



Essayez de trouver comment faire ces formes... Il n'y a rien de compliqué, je vous assure ! Bon, l'étoile est peut-être un peu plus difficile que le reste, mais elle n'est pas insurmontable...

Voici les codes :

Secret (cliquez pour afficher)

La classe `Panneau`

Code : Java

```
import java.awt.Color;
import java.awt.Font;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;

import javax.swing.JPanel;

public class Panneau extends JPanel {
```

```
            private int posX = -50;
        private int posY = -50;
        private String forme = "ROND";


        public void paintComponent(Graphics g){
                //On décide d'une couleur de fond pour notre
    rectangle
                g.setColor(Color.white);
                //On dessine celui-ci afin qu'il prenne toute la
    surface
                g.fillRect(0, 0, this.getWidth(), this.getHeight());
                //On redéfinit une couleur pour notre rond
                g.setColor(Color.red);
                //On délègue la méthode de dessin à la méthode draw()
                draw(g);
        }

        private void draw(Graphics g){

            if(this.forme.equals("ROND")){
                    g.fillOval(posX, posY, 50, 50);
            }
            if(this.forme.equals("CARRE")){
                    g.fillRect(posX, posY, 50, 50);
            }
            if(this.forme.equals("TRIANGLE")){

                    //calcul des sommets
                    //Le sommet 1 est à la moitié du côté supérieur
    du carré de 50
                    int s1X = posX + 25;
                    int s1Y = posY;
                    //Le sommet deux est en bas à droite
                    int s2X = posX + 50;
                    int s2Y = posY + 50;
                    //Le sommet Trois en bas à gauche
                    int s3X = posX;
                    int s3Y = posY + 50;

                    // Nous créons deux tableaux de coordonnées
                    int[] ptsX = {s1X, s2X, s3X};
                    int[] ptsY = {s1Y, s2Y, s3Y};

                    //Et nous utilisons la méthode fillPolygon
                    g.fillPolygon(ptsX, ptsY, 3);
            }
            if(this.forme.equals("ETOILE")){

                    //Pour l'étoile, je me contente de tracer des
    lignes
                    //dans le carré correspondant à peu près à une
    étoile...
                    //Mais ce code peut être amélioré !
                    int s1X = posX + 25;
                    int s1Y = posY;
                    int s2X = posX + 50;
                    int s2Y = posY + 50;
                    g.drawLine(s1X, s1Y, s2X, s2Y);

                    int s3X = posX;
                    int s3Y = posY + 17;
                    g.drawLine(s2X, s2Y, s3X, s3Y);

                    int s4X = posX + 50;
                    int s4Y = posY + 17;
                    g.drawLine(s3X, s3Y, s4X, s4Y);

                    int s5X = posX;
                    int s5Y = posY + 50;
```

```
                    g.drawLine(s4X, s4Y, s5X, s5Y);
                    g.drawLine(s5X, s5Y, s1X, s1Y);
                }

            }


            public void setForme(String form){
                this.forme = form;
            }

            public int getPosX() {
                    return posX;
            }

            public void setPosX(int posX) {
                    this.posX = posX;
            }

            public int getPosY() {
                    return posY;
            }

            public void setPosY(int posY) {
                    this.posY = posY;
            }
        }
```

La classe Fenetre

Code : Java

```
        import java.awt.BorderLayout;
        import java.awt.Color;
        import java.awt.event.ActionEvent;
        import java.awt.event.ActionListener;
        import java.awt.event.ItemEvent;
        import java.awt.event.ItemListener;

        import javax.swing.JButton;
        import javax.swing.JComboBox;
        import javax.swing.JFrame;
        import javax.swing.JLabel;
        import javax.swing.JPanel;

        public class Fenetre extends JFrame{

                private Panneau pan = new Panneau();
                private JButton bouton = new JButton("Go");
                private JButton bouton2 = new JButton("Stop");
            private JPanel container = new JPanel();
            private JLabel label = new JLabel("Choix de la forme");
            private int compteur = 0;
            private boolean animated = true;
            private boolean backX, backY;
            private int x,y ;
            private Thread t;

            private JComboBox combo = new JComboBox();

            public Fenetre(){

                    this.setTitle("Animation");
                    this.setSize(300, 300);
                    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
                this.setLocationRelativeTo(null);

                container.setBackground(Color.white);
                container.setLayout(new BorderLayout());
                container.add(pan, BorderLayout.CENTER);

                bouton.addActionListener(new BoutonListener());
                bouton2.addActionListener(new Bouton2Listener());
                bouton2.setEnabled(false);

                JPanel south = new JPanel();
                south.add(bouton);
                south.add(bouton2);
                container.add(south, BorderLayout.SOUTH);

                combo.addItem("ROND");
                combo.addItem("CARRE");
                combo.addItem("TRIANGLE");
                combo.addItem("ETOILE");

                combo.addActionListener(new FormeListener());

                JPanel top = new JPanel();
                top.add(label);
                top.add(combo);

                container.add(top, BorderLayout.NORTH);
                this.setContentPane(container);
                this.setVisible(true);

        }


            private void go(){
                    x = pan.getPosX();
                    y = pan.getPosY();
               while(this.animated){

                    if(x < 1)backX = false;
                    if(x > pan.getWidth()-50)backX = true;
                    if(y < 1)backY = false;
                    if(y > pan.getHeight()-50)backY = true;
                    if(!backX)pan.setPosX(++x);
                    else pan.setPosX(--x);
                    if(!backY) pan.setPosY(++y);
                    else pan.setPosY(--y);
                    pan.repaint();

                   try {
                        Thread.sleep(3);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
               }
            }


            /**
    * classe qui écoute notre bouton
    */
            public class BoutonListener implements ActionListener{

                    /**
    * Redéfinitions de la méthode actionPerformed
    */
                    public void actionPerformed(ActionEvent arg0) {
                            animated = true;
                            t = new Thread(new PlayAnimation());
                            t.start();
                            bouton.setEnabled(false);
```

```
                        bouton2.setEnabled(true);

                }

        }

        /**
* classe qui écoute notre bouton2
*/
        class Bouton2Listener  implements ActionListener{

                /**
* Redéfinitions de la méthode actionPerformed
*/
                public void actionPerformed(ActionEvent e) {
                        animated = false;
                        bouton.setEnabled(true);
                        bouton2.setEnabled(false);
                }

        }

        class PlayAnimation implements Runnable{

                public void run() {
                        go();
                }
        }


        class FormeListener implements ActionListener{

                public void actionPerformed(ActionEvent e) {
                        //la méthode retourne un Object puisque
nous passons des Object dans une liste
                        //Il faut donc utiliser la méthode
toString() pour retourner un String (ou utiliser un cast)

pan.setForme(combo.getSelectedItem().toString());
                }
        }
}
```

Et voilà le travail !
Vous avez vu qu'il n'y avait rien de sorcier ici. En fait, vu que vous avez l'habitude d'utiliser des objets graphiques et des implémentations d'interfaces, les choses vont s'accélérer car le principe reste le même pour tous les objets graphiques de base !

Bon, j'ai tout de même fait un léger topo et un mini-QCM...
Retour en haut

Ce qu'il faut retenir

- L'objet **`JComboBox`** est dans le package **`javax.swing`**.
- Vous pouvez ajouter des éléments dans une liste avec la méthode **`addItem(Object obj)`**.
- Vous pouvez aussi instancier une liste avec un tableau de données.
- L'interface **`ItemListener`** vous permet de gérer les états de vos éléments.
- Vous pouvez aussi utiliser l'interface **`ActionListener`**.
- La méthode **`getSelectedItem()`** retourne une variable de type **`Object`**, pensez donc à faire un cast ou utiliser la méthode **`toString()`** si les éléments sont de ce type.

Retour en haut

Rien de bien méchant ici, vous êtes habitués à utiliser les objets `swing` et les interfaces...
Continuons notre petit tour parmi les objets du package `javax.swing`, alors ! En route vers : **les cases à cocher** !

Retour en haut

Les cases à cocher : l'objet JCheckBox

Nous allons ici aborder un autre objet, mais comme je vous l'ai dit, le principe de fonctionnement est le même pour tous !
Nous allons donc pouvoir accélérer notre rythme d'apprentissage !

Au passage, il y a un autre objet qui s'utilise *quasiment* de la même manière que celui-ci, nous le verrons donc en même temps dans ce chapitre !

Premier exemple

Avant de commencer, nous allons nous familiariser avec cet objet dans un environnement vierge... Comme d'habitude .

Créez donc un projet vide avec une classe contenant une méthode **main** et une classe héritée de **JFrame**.

Lorsque vous créez un nouveau projet, comme ici, et que vous souhaitez utiliser une classe présente dans un autre projet, vous pouvez faire un copier-coller de celle-ci dans votre nouveau projet...
Ceci soit en faisant un **clic droit** > Copy, puis dans votre nouveau projet **clic droit** > Paste, soit avec le menu Edition en utilisant les mêmes commandes, ou encore avec les **raccourcis clavier** associés à ces commandes. On gagne du temps !

Ceci fait, nous allons utiliser notre nouvel objet. Celui-ci peut être instancié avec un String en paramètre, ce paramètre servira de libellé à notre objet. Vu qu'il n'y a rien de compliqué à faire ceci, voici le code et le résultat :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;

import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;


public class Fenetre extends JFrame {

        private JPanel container = new JPanel();
        private JCheckBox check1 = new JCheckBox("Case 1");
        private JCheckBox check2 = new JCheckBox("Case 2");
        private JLabel label = new JLabel("Une ComboBox");

        public Fenetre(){

        this.setTitle("Animation");
        this.setSize(300, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());

        JPanel top = new JPanel();
        top.add(check1);
        top.add(check2);

        container.add(top, BorderLayout.NORTH);
        this.setContentPane(container);
        this.setVisible(true);
        }
}
```

Résultat :

Vous pouvez aussi cocher la case en appelant la méthode **setSelected(boolean bool)** en lui passant **true**.
Cet objet a, comme tous les autres, une multitude de méthodes afin de pouvoir travailler, je vous invite aussi à fouiner un peu...

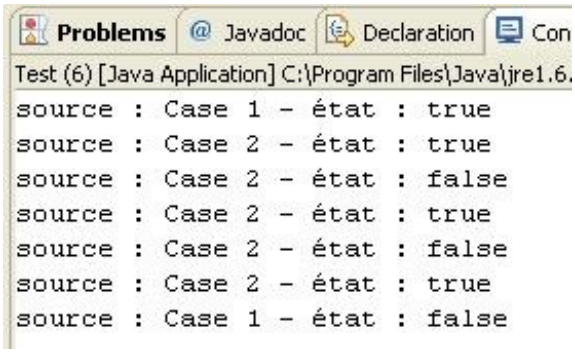
Créez maintenant une implémentation de l'interface **ActionListener**, vous connaissez bien la marche à suivre... Et contrôlez que votre objet est coché ou non avec la méthode **isSelected()** qui retourne un booléen ; voici un code mettant en oeuvre cette demande :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;


public class Fenetre extends JFrame {

        private JPanel container = new JPanel();
        private JCheckBox check1 = new JCheckBox("Case 1");
        private JCheckBox check2 = new JCheckBox("Case 2");
        private JLabel label = new JLabel("Une ComboBox");

        public Fenetre(){

        this.setTitle("Animation");
        this.setSize(300, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());

        JPanel top = new JPanel();

        check1.addActionListener(new StateListener());
        check2.addActionListener(new StateListener());

        top.add(check1);
        top.add(check2);
```

```
            container.add(top, BorderLayout.NORTH);
            this.setContentPane(container);
            this.setVisible(true);
            }

            class StateListener implements ActionListener{

                    public void actionPerformed(ActionEvent e) {
                            System.out.println("source : " +
    ((JCheckBox)e.getSource()).getText() + " - état : " +
    ((JCheckBox)e.getSource()).isSelected());
                    }

            }
    }
```

Résultat :



Ici, je me suis amusé à cocher et décocher mes cases. Rien de bien difficile ici, ça devient la routine...

Retour en haut

Un pseudo-morphing pour notre animation

Nous allons utiliser cet objet afin de faire en sorte que nos formes changent de taille et nous donnent un pseudo-effet de morphing...

Que nous faut-il pour arriver à cela ?

Déjà, la taille de notre forme est fixe, il faut changer ça ! Allez, hop... Une variable de type `int` dans notre classe `Panneau`, disons `drawSize`, initialisée à ***50***.
Tout comme le déplacement, nous devons savoir quand augmenter ou réduire la taille de notre forme : nous utiliserons la même méthode que pour le déplacement.

Un deuxième booléen sera nécessaire pour savoir si le "***mode morphing***" est activé ou non...

En ce qui concerne la taille, si on réduit (ou augmente) celle-ci d'une unité à chaque rafraîchissement, l'effet de morphing sera **ULTRA-RAPIDE**. Donc, pour ralentir l'effet, nous allons utiliser une méthode qui va retourner ***1*** ou ***0*** selon le nombre de rafraîchissements. Ceci signifie que nous allons avoir besoin d'une variable pour dénombrer ceux-ci. Nous ferons une réduction (ou augmentation) tous les 10 tours !

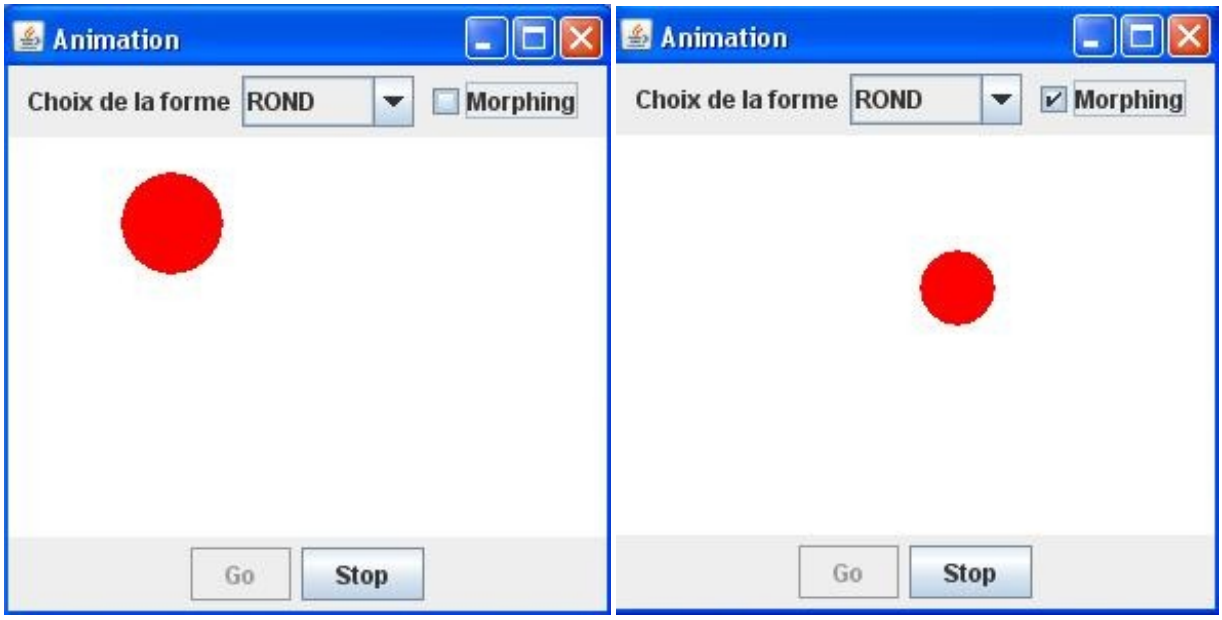
Pour bien séparer les deux cas de figure, nous allons faire une deuxième méthode de dessin dans la classe `Panneau` qui aura pour rôle de dessiner le morphing ; appelons-la `drawMorph(Graphics g)`.

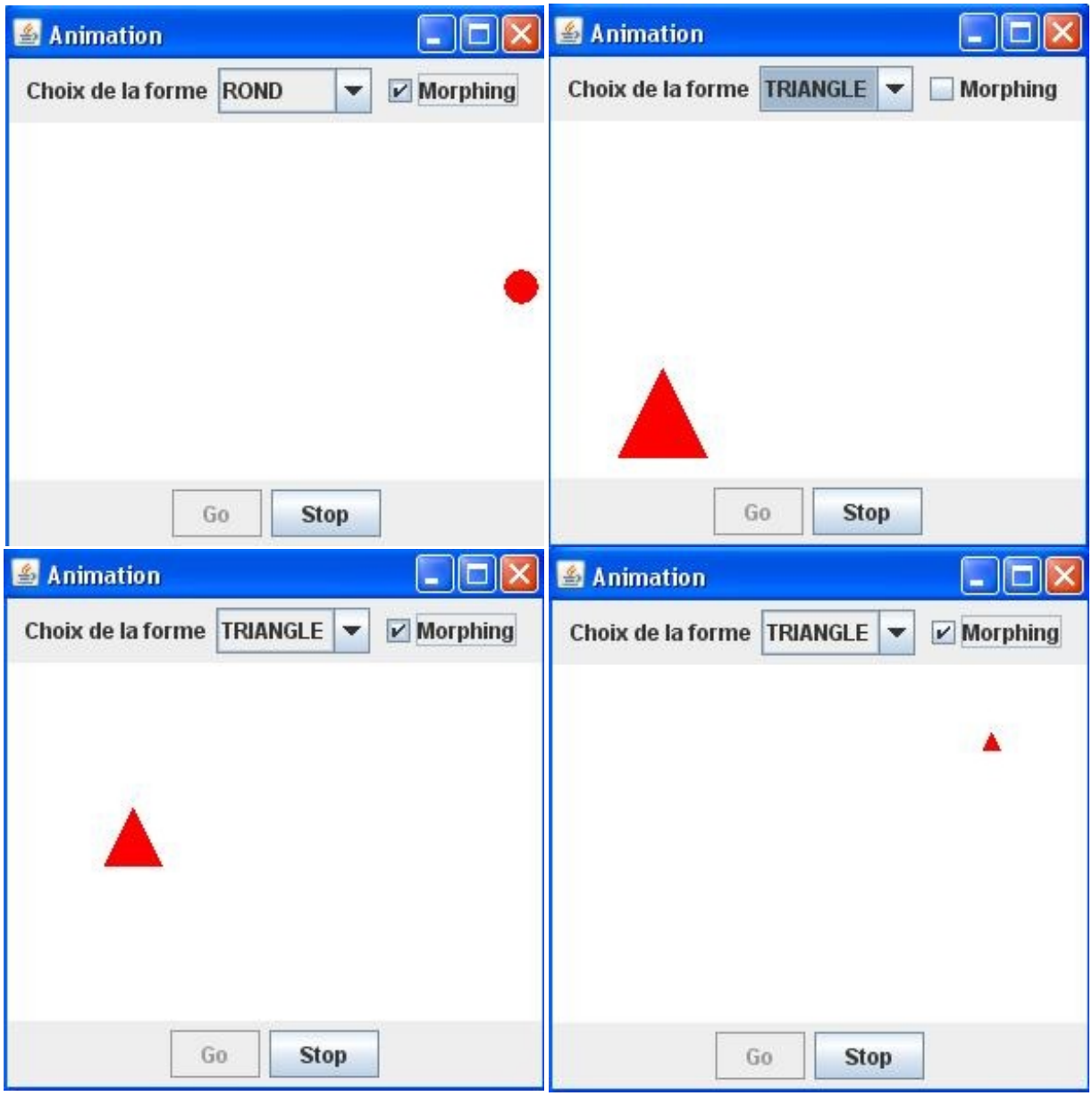
Lorsque nous cocherons notre case à cocher, le morphing sera lancé, et inversement ! La classe `Panneau` devra donc avoir un mutateur pour le booléen de morphing...

Cependant, dans notre classe `Fenetre`, nous gérons la collision avec les bords... Mais en "***mode morphing***", la taille de notre forme n'est plus la même... Donc, il va falloir gérer ce nouveau cas de figure dans notre méthode `go()`. Notre classe `Panneau` devra avoir un accesseur pour la taille actuelle de la forme...

Vous avez tous les éléments en main pour réussir ceci.

Voici ce que vous devriez obtenir (je n'ai mis que le rond et le triangle... mais ça fonctionnera avec toutes les formes) :





Voici les codes :

Secret (cliquez pour afficher)

Fichier Panneau.java

Code : Java

```
import java.awt.Color;
import java.awt.Font;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;

import javax.swing.JPanel;

public class Panneau extends JPanel {

        private int posX = -50;
    private int posY = -50;
    private int drawSize = 50;
    //boolean pour le mode morphing et pour savoir si la taille
```

```
        doit réduire
            private boolean morph = false, reduce = false;;
            private String forme = "ROND";
            //Le compteur de rafraîchissements
            private int increment = 0;

            public void paintComponent(Graphics g){
                    g.setColor(Color.white);
                    g.fillRect(0, 0, this.getWidth(), this.getHeight());
                    g.setColor(Color.red);
                    //Si le mode morphing est activé, on peint le
        morphing
                    if(this.morph)
                        drawMorph(g);
                    //sinon, mode normal
                    else
                        draw(g);
            }

            private void draw(Graphics g){

                if(this.forme.equals("ROND")){
                        g.fillOval(posX, posY, 50, 50);
                }
                if(this.forme.equals("CARRE")){
                        g.fillRect(posX, posY, 50, 50);
                }
                if(this.forme.equals("TRIANGLE")){

                        int s1X = posX + 50/2;
                        int s1Y = posY;
                        int s2X = posX + 50;
                        int s2Y = posY + 50;
                        int s3X = posX;
                        int s3Y = posY + 50;

                        int[] ptsX = {s1X, s2X, s3X};
                        int[] ptsY = {s1Y, s2Y, s3Y};

                        g.fillPolygon(ptsX, ptsY, 3);
                }
                if(this.forme.equals("ETOILE")){

                        int s1X = posX + 50/2;
                        int s1Y = posY;
                        int s2X = posX + 50;
                        int s2Y = posY + 50;
                        g.drawLine(s1X, s1Y, s2X, s2Y);

                        int s3X = posX;
                        int s3Y = posY + 50/3;
                        g.drawLine(s2X, s2Y, s3X, s3Y);

                        int s4X = posX + 50;
                        int s4Y = posY + 50/3;
                        g.drawLine(s3X, s3Y, s4X, s4Y);

                        int s5X = posX;
                        int s5Y = posY + 50;
                        g.drawLine(s4X, s4Y, s5X, s5Y);
                        g.drawLine(s5X, s5Y, s1X, s1Y);
                }

            }

            /**
        * Méthode qui peint le morphing
        * @param g
        */
            private void drawMorph(Graphics g){
```

```
                //On incrémente le tour
                increment++;
                //On regarde si on doit réduire ou non
                if(drawSize >= 50)reduce = true;
                if(drawSize <= 10)reduce = false;

                if(reduce)
                        drawSize = drawSize - getUsedSize();
                else
                        drawSize = drawSize + getUsedSize();

                if(this.forme.equals("ROND")){
                        g.fillOval(posX, posY, drawSize, drawSize);
                }
                if(this.forme.equals("CARRE")){
                        g.fillRect(posX, posY, drawSize, drawSize);
                }
                if(this.forme.equals("TRIANGLE")){

                        int s1X = posX + drawSize/2;
                        int s1Y = posY;
                        int s2X = posX + drawSize;
                        int s2Y = posY + drawSize;
                        int s3X = posX;
                        int s3Y = posY + drawSize;

                        int[] ptsX = {s1X, s2X, s3X};
                        int[] ptsY = {s1Y, s2Y, s3Y};

                        g.fillPolygon(ptsX, ptsY, 3);
                }
                if(this.forme.equals("ETOILE")){

                        int s1X = posX + drawSize/2;
                        int s1Y = posY;
                        int s2X = posX + drawSize;
                        int s2Y = posY + drawSize;
                        g.drawLine(s1X, s1Y, s2X, s2Y);

                        int s3X = posX;
                        int s3Y = posY + drawSize/3;
                        g.drawLine(s2X, s2Y, s3X, s3Y);

                        int s4X = posX + drawSize;
                        int s4Y = posY + drawSize/3;
                        g.drawLine(s3X, s3Y, s4X, s4Y);

                        int s5X = posX;
                        int s5Y = posY + drawSize;
                        g.drawLine(s4X, s4Y, s5X, s5Y);
                        g.drawLine(s5X, s5Y, s1X, s1Y);
                }


        }

        /**
     * Méthode qui retourne le nombre à retrancher (ou ajouter) pour
     le morphing
     * @return res
     */
        private int getUsedSize(){
                int res = 0;
                //Si le nombre de tours est de 10
                //On réinitialise l'incrément et on retourne 1
                if(increment / 10 == 1){
                        increment = 0;
                        res = 1;
                }
                return res;
```

```
    }

    public int getDrawSize(){
        return drawSize;
    }

    public boolean isMorph(){
        return morph;
    }

    public void setMorph(boolean bool){
        this.morph = bool;
        //On réinitialise la taille
        drawSize = 50;
    }

    public void setForme(String form){
        this.forme = form;
    }

    public int getPosX() {
            return posX;
    }

    public void setPosX(int posX) {
            this.posX = posX;
    }

    public int getPosY() {
            return posY;
    }

    public void setPosY(int posY) {
            this.posY = posY;
    }
}
```

Fichier Fenetre.java

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class Fenetre extends JFrame{

        private Panneau pan = new Panneau();
        private JButton bouton = new JButton("Go");
        private JButton bouton2 = new JButton("Stop");
    private JPanel container = new JPanel();
    private JLabel label = new JLabel("Choix de la forme");
    private int compteur = 0;
    private boolean animated = true;
    private boolean backX, backY;
    private int x,y ;
    private Thread t;
```

```
        private JComboBox combo = new JComboBox();

        private JCheckBox morph = new JCheckBox("Morphing");

        public Fenetre(){

                this.setTitle("Animation");
                this.setSize(300, 300);
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setLocationRelativeTo(null);

                container.setBackground(Color.white);
                container.setLayout(new BorderLayout());
                container.add(pan, BorderLayout.CENTER);

                bouton.addActionListener(new BoutonListener());
                bouton2.addActionListener(new Bouton2Listener());
                bouton2.setEnabled(false);

                JPanel south = new JPanel();
                south.add(bouton);
                south.add(bouton2);
                container.add(south, BorderLayout.SOUTH);

                combo.addItem("ROND");
                combo.addItem("CARRE");
                combo.addItem("TRIANGLE");
                combo.addItem("ETOILE");
                combo.addActionListener(new FormeListener());

                morph.addActionListener(new MorphListener());

                JPanel top = new JPanel();
                top.add(label);
                top.add(combo);
                top.add(morph);

                container.add(top, BorderLayout.NORTH);
                this.setContentPane(container);
                this.setVisible(true);

        }


            private void go(){
                    x = pan.getPosX();
                    y = pan.getPosY();
            while(this.animated){

                    //Si le mode morphing est activé, on utilise la
    taille actuelle de la forme
                    if(pan.isMorph())
                    {
                            if(x < 1)backX = false;
                        if(x > pan.getWidth() -
    pan.getDrawSize())backX = true;
                        if(y < 1)backY = false;
                        if(y > pan.getHeight() -
    pan.getDrawSize())backY = true;
                    }
                    //Sinon, comme d'habitude
                    else
                    {
                            if(x < 1)backX = false;
                        if(x > pan.getWidth()-50)backX = true;
                        if(y < 1)backY = false;
                        if(y > pan.getHeight()-50)backY = true;
                    }

                    if(!backX)pan.setPosX(++x);
```

```
                else pan.setPosX(--x);
                if(!backY) pan.setPosY(++y);
                else pan.setPosY(--y);
                pan.repaint();

                try {
                        Thread.sleep(3);
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
        }

        public class BoutonListener implements ActionListener{

                public void actionPerformed(ActionEvent arg0) {
                        animated = true;
                        t = new Thread(new PlayAnimation());
                        t.start();
                        bouton.setEnabled(false);
                        bouton2.setEnabled(true);
                }
        }

        class Bouton2Listener  implements ActionListener{

                public void actionPerformed(ActionEvent e) {
                        animated = false;
                        bouton.setEnabled(true);
                        bouton2.setEnabled(false);
                }
        }

        class PlayAnimation implements Runnable{

                public void run() {
                        go();
                }
        }


        class FormeListener implements ActionListener{

                public void actionPerformed(ActionEvent e) {

pan.setForme(combo.getSelectedItem().toString());
                }
        }

        class MorphListener implements ActionListener{

                public void actionPerformed(ActionEvent e) {
                        //Si la case est cochée, activation du
mode morphing
                        if(morph.isSelected())pan.setMorph(true);
                        //Sinon, rien !
                        else pan.setMorph(false);
                }

        }


}
```

Alors ? Vous en pensez quoi ?
J'aime bien, moi...

Vous voyez que l'utilisation des **`JCheckBox`** est très simple... Je vous propose maintenant d'aller voir un de ses cousins...
Retour en haut

Les boutons radio : l'objet JRadioButton

Le voici, le cousin éloigné...

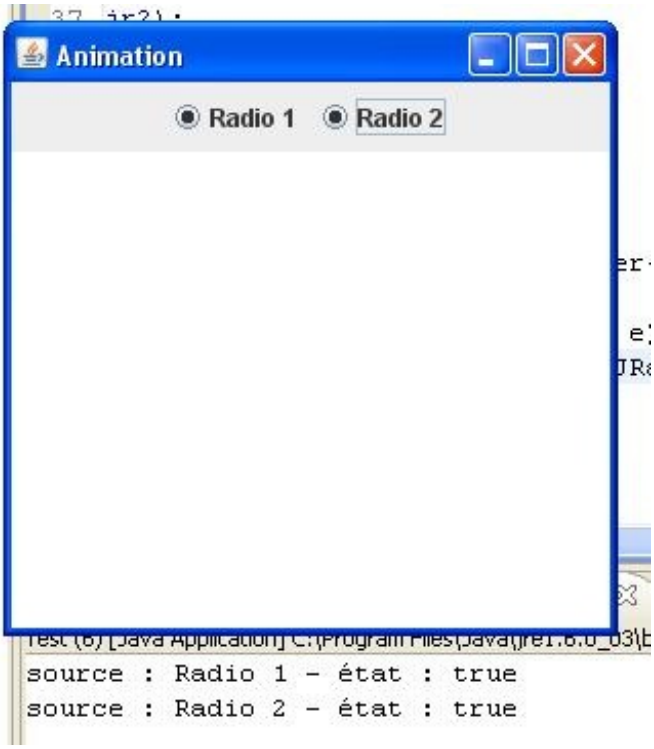
Bon, alors : le principe est de pouvoir avoir deux (ou plus) choix distincts, mais qu'un seul soit sélectionné à la fois... C'est-à-dire que pour un choix ***oui / non*** , le choix ***oui*** étant présélectionné, lorsque nous choisirons ***non***, l'autre choix se désélectionnera tout seul...

L'instanciation se fait de la même manière que pour un **JCheckBox** ; d'ailleurs, nous allons utiliser l'exemple du début de chapitre en remplaçant les cases à cocher par des boutons radio. Voici les codes et le résultat :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JRadioButton;


public class Fenetre extends JFrame {

        private JPanel container = new JPanel();

        private JRadioButton jr1 = new JRadioButton("Radio 1");
        private JRadioButton jr2 = new JRadioButton("Radio 2");

        public Fenetre(){

        this.setTitle("Animation");
        this.setSize(300, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());

        JPanel top = new JPanel();

        jr1.addActionListener(new StateListener());
        jr2.addActionListener(new StateListener());

        top.add(jr1);
        top.add(jr2);

        container.add(top, BorderLayout.NORTH);
        this.setContentPane(container);
        this.setVisible(true);
        }

        class StateListener implements ActionListener{

                public void actionPerformed(ActionEvent e) {
                        System.out.println("source : " +
((JRadioButton)e.getSource()).getText() + " - état : " +
((JRadioButton)e.getSource()).isSelected());
                }

        }
}
```

Résultat :



Vous pouvez voir que ces objets s'utilisent de la même manière.

Euh... Tu nous as dis que seul un choix devrait être sélectionné... Or ici, tes deux radios sont sélectionnés...

Tout à fait... Voilà la différence entre ces deux objets... Pour qu'un seul bouton radio soit sélectionné à la fois, nous devons définir un groupe de boutons, **ButtonGroup**. Dans celui-ci, nous ajouterons nos boutons radio et ainsi seul un bouton sera sélectionné à la fois !

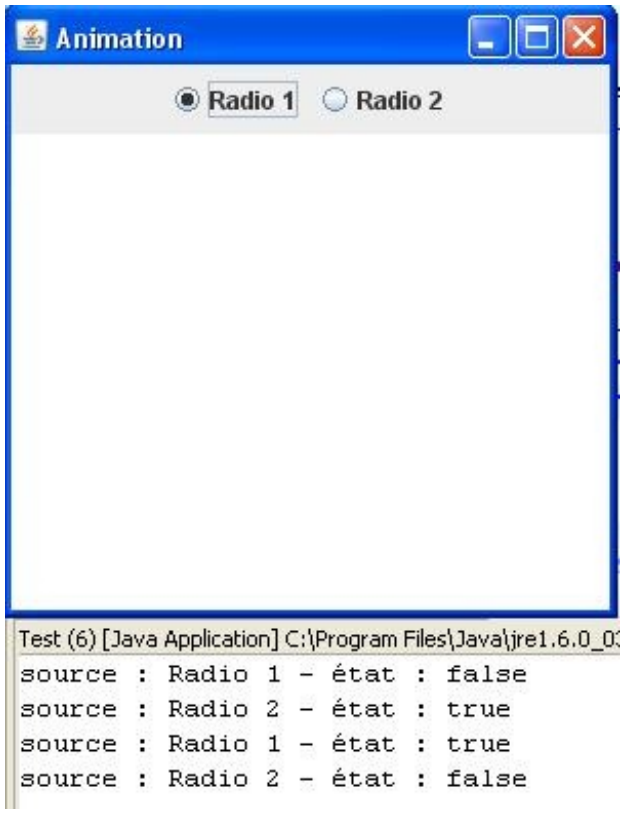
Code :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.ButtonGroup;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JRadioButton;


public class Fenetre extends JFrame {

        private JPanel container = new JPanel();

        private JRadioButton jr1 = new JRadioButton("Radio 1");
        private JRadioButton jr2 = new JRadioButton("Radio 2");
        private ButtonGroup bg = new ButtonGroup();

        public Fenetre(){

        this.setTitle("Animation");
        this.setSize(300, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);
```

```
                container.setBackground(Color.white);
                container.setLayout(new BorderLayout());

                JPanel top = new JPanel();

                jr1.setSelected(true);
                jr1.addActionListener(new StateListener());
                jr2.addActionListener(new StateListener());

                bg.add(jr1);
                bg.add(jr2);

                top.add(jr1);
                top.add(jr2);

                container.add(top, BorderLayout.NORTH);
                this.setContentPane(container);
                this.setVisible(true);
                }

                class StateListener implements ActionListener{

                        public void actionPerformed(ActionEvent e) {
                                System.out.println("source : " +
        jr1.getText() + " - état : " + jr1.isSelected());
                                System.out.println("source : " +
        jr2.getText() + " - état : " + jr2.isSelected());
                        }

                }
        }
```

Résultat :



Une précision, toutefois... Lorsque je vous ai dit que la différence entre ces composants se trouvait dans cette notion de groupe de boutons... Je vous ai un peu induits en erreur .

En fait, j'ai dit ceci car il est plus fréquent d'avoir ce genre de chose sur des boutons de type radio que sur un autre, mais rien ne vous empêche de faire la même chose pour des cases à cocher...
Effectivement, l'objet **`ButtonGroup`** accepte en paramètre un type `AbstractButton` qui est la super-classe de tous les boutons... Vous pouvez donc en tirer les conclusions qui s'imposent !
Retour en haut

Ce qu'il faut retenir

- Les objets **JCheckBox**, **JRadioButton** et **ButtonGroup** sont dans le package **javax.swing**
- Vous pouvez savoir si un de ces composants est sélectionné grâce à la méthode **isSelected()**.
- Cette méthode retourne **true** si l'objet est sélectionné, et **false** dans le cas contraire.
- Vous pouvez restreindre les choix dans un groupe de réponses en utilisant la classe **ButtonGroup**.
- Vous pouvez ajouter des boutons à un groupe de boutons grâce à la méthode **add(AbstractButton button)**.

Allez... Pas besoin de QCM ici... Il n'y a rien de compliqué !

Retour en haut

Je sens que vous êtes de plus en plus à l'aise avec la programmation événementielle !
Continuons donc avec : **les zones de texte**.

Retour en haut

Les champs de texte : l'objet JTextField

Dans ce chapitre, nous allons voir l'objet qui va vous permettre de saisir des informations.
Celui-ci est très simple d'utilisation aussi...

Ne perdons pas de temps, allons-y !

Utilisation

Je pense que vous savez ce qu'il vous reste à faire... Donc, si ce n'est pas encore fait, créez-vous un nouveau projet avec les classes habituelles.

Comme le titre du chapitre l'indique, nous allons utiliser l'objet **`JTextField`**. Comme vous pouvez vous en douter, cet objet a lui aussi les méthodes de redimensionnement, de couleur de police...

De ce fait, je commence donc avec un exemple complet. Regardez et testez ce code :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;


public class Fenetre extends JFrame {

        private JPanel container = new JPanel();
        private JTextField jtf = new JTextField("Valeur par
défaut");
        private JLabel label = new JLabel("Un JTextField");


        public Fenetre(){

        this.setTitle("Animation");
        this.setSize(300, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());

        JPanel top = new JPanel();

        Font police = new Font("Arial", Font.BOLD, 14);
        jtf.setFont(police);
        jtf.setPreferredSize(new Dimension(150, 30));
        jtf.setForeground(Color.BLUE);

        top.add(label);
        top.add(jtf);

        container.add(top, BorderLayout.NORTH);
        this.setContentPane(container);
        this.setVisible(true);
        }
}
```

Ce qui nous donne :

Vous pouvez voir que c'est très simple ! Vous pouvez saisir ce que vous voulez dans cette zone de texte.

Vous pouvez initialiser le contenu avec la méthode `setText(String str)` ou le récupérer avec la méthode `getText()`.

Il existe toutefois un objet très ressemblant à celui-ci, un peu plus étoffé. En fait, cet objet permet d'avoir un `JTextField` formaté pour recevoir un certain type de saisie (date, pourcentage...).

Voyons ça tout de suite...
Retour en haut

Un objet plus restrictif : le JFormattedTextField

Avec ce genre d'objet, vous allez pouvoir vous éviter beaucoup de contrôles et de cast sur le contenu de vos zones de texte...

Si vous avez essayé de récupérer le contenu du **JTextField** utilisé ci-dessus (lors du clic sur un bouton, par exemple...) vous avez dû vous rendre compte que le type de contenu lui était égal...

Mais, un jour sûrement, vous aurez besoin d'une zone de texte qui n'accepte qu'un certain type de donnée... Avec l'objet **JFormattedTextField**, nous nous en rapprochons, mais vous verrez que vous pouvez faire encore mieux !
Cet objet retourne une valeur uniquement si celle-ci correspond à ce que vous lui avez demandé de contenir, je m'explique : si vous voulez que votre zone de texte DOIVE contenir des entiers, ou des dates... c'est possible !

Par contre, ce contrôle se fait lorsque vous quittez le champ en question ! **Vous pouvez saisir des lettres dans un objet n'acceptant que des entiers, mais la méthode `getText()` ne renverra RIEN car le contenu sera effacé si les données ne correspondent pas aux attentes ! !**

Voici un code et deux exemples :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.text.NumberFormat;

import javax.swing.JButton;
import javax.swing.JFormattedTextField;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;


public class Fenetre extends JFrame {

        private JPanel container = new JPanel();
        private JFormattedTextField jtf = new
JFormattedTextField(NumberFormat.getIntegerInstance());
        private JFormattedTextField jtf2 = new
JFormattedTextField(NumberFormat.getPercentInstance());
        private JLabel label = new JLabel("Un JTextField");
        private JButton b = new JButton ("OK");

        public Fenetre(){

        this.setTitle("Animation");
        this.setSize(300, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());

        JPanel top = new JPanel();

        Font police = new Font("Arial", Font.BOLD, 14);
        jtf.setFont(police);
        jtf.setPreferredSize(new Dimension(150, 30));
        jtf.setForeground(Color.BLUE);

        jtf2.setPreferredSize(new Dimension(150, 30));

        b.addActionListener(new BoutonListener());

        top.add(label);
        top.add(jtf);
```

```
                top.add(jtf2);
                top.add(b);

               // container.add(top, BorderLayout.NORTH);
                this.setContentPane(top);
                this.setVisible(true);
                }

                class BoutonListener implements ActionListener{

                        public void actionPerformed(ActionEvent e) {
                                System.out.println("TEXT : jtf " +
        jtf.getText());
                                System.out.println("TEXT : jtf2 " +
        jtf2.getText());
                        }

                }
        }
```

Exemple valide :

Exemple invalide :

Vous pouvez voir que notre objet met aussi en forme la saisie lorsque celle-ci est valide ! Celui-ci sépare les nombres 3 par 3 afin de facilité la lecture...

Voici les types de contenus que vous pouvez utiliser :

- **`NumberFormat`** avec :
  - `getIntegerInstance()`
  - `getPercentInstance()`
  - `getNumberInstance()`
- **`DateFormat`** avec :
  - `getTimeInstance()`
  - `getDateInstance()`
- **`MessageFormat`**

Sans rentrer dans les détails, vous pouvez aussi utiliser un objet **`MaskFormatter`** qui permet d'avoir un format à taille fixe dans votre zone de texte. Ceci est très pratique lorsque vous voulez un numéro de téléphone, un numéro de sécurité sociale... Vous devez définir ce format en paramètre dans le constructeur à l'aide de ***méta-caractères***. Ceux-ci permettent de dire à votre objet **`MaskFormatter`** comment doit être constitué le futur contenu de votre zone de texte. Voici la liste de ces ***méta-caractères*** :

- # indique un chiffre ;
- *'* indique un caractère d'échappement ;
- ***U*** indique une lettre (les minuscules sont changées en majuscules) ;
- ***L*** indique une lettre (les majuscules sont changées en minuscules) ;
- ***A*** indique un chiffre ou une lettre ;
- ***?*** indique une lettre ;
- ***** indique que tous les caractères sont acceptés ;
- ***H*** indique que tous les caractères hexadécimaux sont acceptés (0->9, a->f ou A->F).

L'instanciation d'un tel objet peut lever une **`ParseException`**. Vous devrez donc l'entourer d'un bloc **`try{...}catch(ParseException e){...}`**.

Voici à quoi ressemblerait un format téléphonique :

Code : Java

```
try{
   MaskFormatter tel = new MaskFormatter("## ## ## ## ##");
   //Ou encore
   MaskFormatter tel2 = new MaskFormatter("##-##-##-##-##");
   //Vous pouvez ensuite le passer à votre zone de texte
   JFormattedTextField jtf = new JFormattedTextField(tel2);
   }catch(ParseException e){e.printStackTrace();}
```

Vous pouvez vous rendre compte qu'il n'y a rien de compliqué...
Je vous donne tout de même un exemple de code permettant de saisir un numéro de téléphone français et un numéro de téléphone américain :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.text.ParseException;

import javax.swing.JButton;
import javax.swing.JFormattedTextField;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.text.MaskFormatter;


public class Fenetre extends JFrame {

    private JPanel container = new JPanel();
    private JFormattedTextField jtf;
    private JFormattedTextField jtf2;
    private JLabel label = new JLabel("Téléphone FR ");
    private JLabel label2 = new JLabel("Téléphone USA");
    private JButton b = new JButton ("OK");

    /**
* Constructeur de l'objet
*/
    public Fenetre(){

        this.setTitle("Animation");
        this.setSize(300, 150);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());

        try{
            MaskFormatter tel = new MaskFormatter("##-##-##-##-##");
            MaskFormatter telUSA = new MaskFormatter("###-####");
            jtf = new JFormattedTextField(tel);
            jtf2 = new JFormattedTextField(telUSA);
        }catch(ParseException e){
         e.printStackTrace();
        }


        JPanel top = new JPanel();

        Font police = new Font("Arial", Font.BOLD, 14);
        jtf.setFont(police);
```

```
            jtf.setPreferredSize(new Dimension(150, 30));
            jtf.setForeground(Color.BLUE);

            jtf2.setPreferredSize(new Dimension(150, 30));

            b.addActionListener(new BoutonListener());

            top.add(label);
            top.add(jtf);
            top.add(label2);
            top.add(jtf2);
            top.add(b);

           // container.add(top, BorderLayout.NORTH);
            this.setContentPane(top);
            this.setVisible(true);
        }

            class BoutonListener implements ActionListener{

                    public void actionPerformed(ActionEvent e) {
                            System.out.println("Téléphone FR : " +
jtf.getText());
                            System.out.println("Téléphone USA : " +
jtf2.getText());
                    }

            }
}
```

Vous pouvez constater qu'avec le méta-caractère utilisé avec notre objet **MaskFormatter**, nous sommes obligé de saisir des chiffres !

Et voici le résultat lorsque nous cliquons sur le bouton :

Je ne sais pas pour le numéro américain, mais le numéro de téléphone français est loin d'être un numéro de téléphone valide !

Ah ! je savais que vous alliez remarquer ce petit détail, de taille je vous l'accorde.
Nous voilà confrontés à un problème qui vous hantera tant que vous programmerez : **L'intégrité de vos données !**

Comme démontré ci-dessus, vous pouvez aider le plus possible l'utilisateur sur ce qu'il doit renseigner comme données dans des

champs, vous ne devrez **JAMAIS FAIRE UNE CONFIANCE AVEUGLE EN CELLES-CI !**
C'est simple : dans ma boîte, on part du principe de ne jamais faire confiance à l'utilisateur !

Nous sommes obligés de faire une multitude de contrôles en plus, mais les applications ont le mérite d'être un tant soit peu sécurisées...

Qu'est-ce que nous pouvons faire dans le cas de ta saisie ?

En réalité, beaucoup de choses :

- tester chaque élément de votre numéro ;
- tester le numéro en entier ;
- dans le cas ou vous n'utilisez pas de `MaskFormatter`, vérifier en plus que les saisies soient numériques ;
- utiliser une expression régulière ;
- empêcher la saisie d'un type de caractères ;
- ...

En gros, vous devrez vérifier l'intégrité de vos données et, dans le cas qui nous intéresse, l'intégrité de vos chaînes de caractères, pendant ou après la saisie !
D'ailleurs, c'est ce que je vous propose de faire, pas plus tard que maintenant !

Retour en haut

Contrôlez vos données post-saisie

Afin de voir comment contrôler au mieux vos données, nous allons travailler avec un `JFormattedTextField` acceptant tous types de caractères. Voici donc notre code :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.text.ParseException;

import javax.swing.JButton;
import javax.swing.JFormattedTextField;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.text.MaskFormatter;


public class Fenetre extends JFrame {

    private JPanel container = new JPanel();
    private JFormattedTextField jtf;
    private JFormattedTextField jtf2;
    private JLabel label = new JLabel("Téléphone FR ");
    private JButton b = new JButton ("OK");

    /**
* Constructeur de l'objet
*/
    public Fenetre(){

        this.setTitle("Animation");
        this.setSize(300, 150);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());

        try{
            MaskFormatter tel = new MaskFormatter("**-**-**-**-**");
            jtf = new JFormattedTextField(tel);
        }catch(ParseException e){
         e.printStackTrace();
        }

        JPanel top = new JPanel();

        Font police = new Font("Arial", Font.BOLD, 14);
        jtf.setFont(police);
        jtf.setPreferredSize(new Dimension(150, 30));
        jtf.setForeground(Color.BLUE);

        b.addActionListener(new BoutonListener());

        top.add(label);
        top.add(jtf);
        top.add(b);

        this.setContentPane(top);
        this.setVisible(true);
     }

    class BoutonListener implements ActionListener{
```

```
            public void actionPerformed(ActionEvent e) {
                    System.out.println("Téléphone FR : " +
jtf.getText());
            }
        }
}
```

Maintenant, vous pouvez saisir n'importe quoi dans ce qui devait être un numéro de téléphone.
Il reste tout de même une restriction sur le nombre de caractères que doit prendre le champ, ici 10, mais, mis à part ça, vous êtes libres de saisir ce que vous voulez :

Première approche

Une méthode de contrôle, un peu compliquée au final, consisterait à exploser la chaîne de caractères grâce à la méthode `split(String regex)` et tester les éléments un par un...
Cette méthode, `split(String regex)` , permet de créer un tableau de **`String`** à partir d'une chaîne de caractères en l'explosant par rapport à l'expression régulière passée en paramètre. Un exemple est toujours mieux :

Code : Java

```
String str = "Je-suis-un-ZérO";
String[] tab = str.split("-");//Donne un tableau à 4 entrées
//tab[0] vaut "Je"
//tab[1] vaut "suis"
//tab[2] vaut "un"
//tab[3] vaut "ZérO"
```

Le paramètre de cette méthode n'est pas une chaîne de caractères banale ! Il s'agit en fait d'une expression régulière. Nous allons y venir.

Voici une façon de faire, un peu barbare mais elle fonctionne :

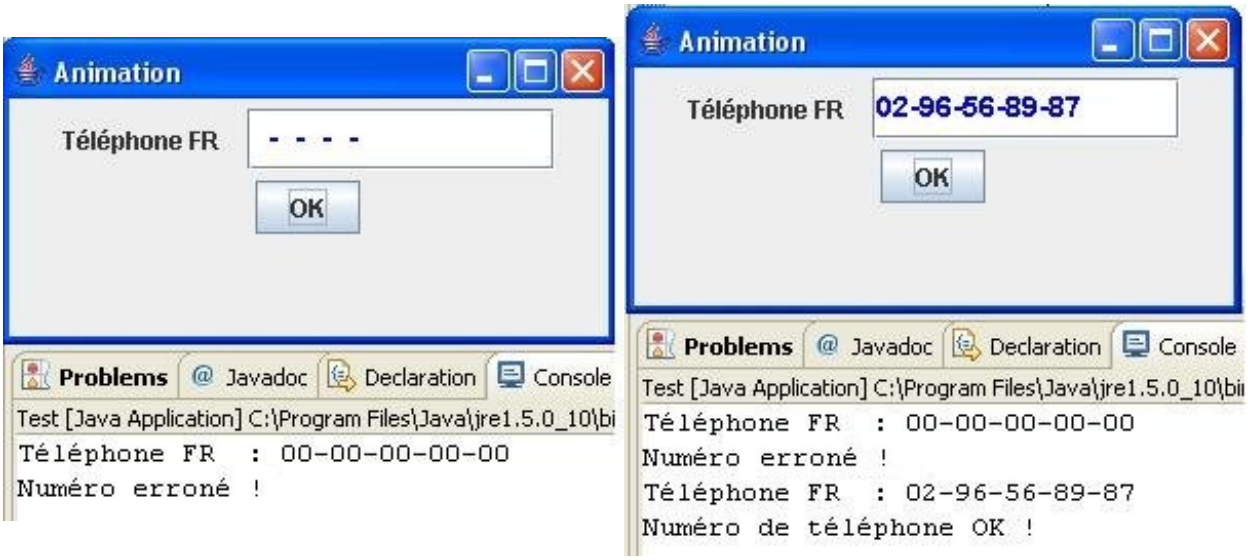
Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.text.ParseException;
```

```
import javax.swing.JButton;
import javax.swing.JFormattedTextField;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.text.MaskFormatter;


public class Fenetre extends JFrame {

    private JPanel container = new JPanel();
    private JFormattedTextField jtf;
    private JFormattedTextField jtf2;
    private JLabel label = new JLabel("Téléphone FR ");
    private JButton b = new JButton ("OK");

    /**
* Constructeur de l'objet
*/
    public Fenetre(){

        this.setTitle("Animation");
        this.setSize(300, 150);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());

        try{
            MaskFormatter tel = new MaskFormatter("**-**-**-**-**");
            jtf = new JFormattedTextField(tel);
        }catch(ParseException e){
         e.printStackTrace();
        }

        JPanel top = new JPanel();

        Font police = new Font("Arial", Font.BOLD, 14);
        jtf.setFont(police);
        jtf.setPreferredSize(new Dimension(150, 30));
        jtf.setForeground(Color.BLUE);

        b.addActionListener(new BoutonListener());

        top.add(label);
        top.add(jtf);
        top.add(b);

        this.setContentPane(top);
        this.setVisible(true);
     }

    class BoutonListener implements ActionListener{
        public void actionPerformed(ActionEvent e) {
                System.out.println("Téléphone FR : " +
jtf.getText());
                String[] tab = jtf.getText().split("-");

                //On contrôle le numéro de téléphone
                //-----------------------------------
                if(!controleData(tab)){
                 System.out.println("Numéro erroné ! ");
                 jtf.setText("");
                }
                else{
                 System.out.println("Numéro de téléphone OK ! ");
                }
        }
```

```
        /**
     * Méthode qui va contrôler la saisie
     * @param data
     * @return Boolean
     */
        private boolean controleData(String[] data){
         int i = 0;

         //On balaye tout le contenu du tableau et on vérifie
         //que les données sont conformes à nos attentes
         while(i < data.length){

          switch(i){

           //Le premier élément doit être numérique et
           //égal à 01 ou 02 ou 03 ou 04 ou 05 ou 06 ou 08 ou 09
           case 0:
            try {
          int j = Integer.parseInt(data[i]);
          if(j < 1 || j > 6 && j != 8 && j != 9)
           return false;
         } catch (NumberFormatException e) {
          return false;
         }

            break;

           //Les autres chiffres doivent être compris entre 00 et 99
     INCLUS
           //Je ne sais pas si ça marche réellement comme ça, mais
     c'est pour l'exemple...
           case 1:
           case 2:
           case 3:
           case 4:
            try {
          int j = Integer.parseInt(data[i]);
          if(j < 0 || j > 99)
           return false;
         } catch (NumberFormatException e) {
          return false;
         }
            break;
          }

          i++;
         }

         return true;
        }
    }
}
```

Ce qui nous donne :



Un peu fastidieux comme façon de contrôler !
Imaginez un peu que vous ayez une multitude de champs à vérifier... Une sacré galère au final !

Allez, dis-nous tout, on te connaît maintenant...

Personnellement, je trouve qu'utiliser des expressions régulières (ou regex) permet plus de souplesse et une économie de code assez conséquente.

On te croit sur parole ! Mais qu'est-ce qu'une regex ?

Bon, je n'ai pas l'intention de réinventer la poudre... Surtout que M@teo a très bien expliqué ça dans deux chapitres de son tuto PHP. Vu que je vais partir du principe que vous connaissez la base des expressions régulières, je vous conseille vivement d'aller faire un tour sur son tuto et, une fois lu, revenez me voir ici-même...

Utiliser des expressions régulières

Comme vous avez pu le constater lors de la lecture du tuto de M@teo, les regex permettent de faire énormément de choses et, dans notre cas, de nous simplifier les contrôles de saisie de notre `JFormattedTextField`.

Maintenant, afin de pouvoir contrôler la saisie, nous allons devoir définir la regex.
Comme mis dans les commentaires du code précédent, je pars du principe qu'un numéro de téléphone est composé comme suit :

- un 0 en tout premier chiffre ;
- un chiffre qui peut être : 1, 2, 3, 4, 5, 6, 8, 9 ;
- ensuite, 4 blocs composés d'un "-" suivi d'un nombre compris entre 00 et 99 inclus.

Si vous avez bien suivi le tuto de M@teo sur les regex, vous devez avoir une regex qui ressemble à ça :

```
#^0[0-689](-[0-9]{2}){2}$#
```

ou à ça :

```
#^0[0-689](-[\d]{2}){2}$#
```

Ces deux regex sont tout à fait correctes pour une application PHP, mais elles ne fonctionneront pas avec une application Java. Ceci pour deux raisons.

- En Java, il n'y a pas besoin de délimiteurs. Vous pouvez donc enlever les deux "#". Ceci concerne les deux regex.
- Ce point, par contre, ne concerne que la deuxième. Le caractère "\", est utilisé comme caractère d'échappement, ceci afin de *dé-spécialiser* des caractères comme "`\n`", "`\r`"... La classe abrégée "`\d`", correspondant à un chiffre, ne

fonctionnera donc pas.

Afin de pouvoir utiliser les classes abrégées dans une regex, il faut faire en sorte que le backslash de la classe abrégée soit interprété comme tel et non comme un caractère d'échappement.

Comment ?

Il faut tout simplement échapper le caractère d'échappement...
Ce qui nous donne : `^0[0-689](-[\\d]{2}){2}$`
Le premier backslash échappe le second, ce qui a pour conséquence que celui-ci est interprété comme un backslash tout ce qu'il y a de plus normal et ainsi que notre classe abrégée fonctionne !

Maintenant, nous sommes parés pour utiliser des regex...
Retour en haut

Utiliser des regex

Avant de nous lancer tête baissée dans l'utilisation des regex en Java, vous devez savoir que vous pouvez procéder de deux façon différentes :

- en utilisant un objet **String** ;
- en utilisant l'API regex qui se trouve dans le package `java.util.regex` .

Les regex et l'objet String

Vous allez voir que c'est simplissime.
Nous avons donc convenu de la regex à utiliser afin de contrôler nos saisies de numéros de téléphone.
Pour mémoire : **`^0[0-689](-[\\d]{2}){4}$`**

Il ne nous reste plus qu'à dire au contenu de notre **`JFormattedTextField`** qu'il doit correspondre à celle-ci.

Cette opération se fait grâce à la méthode `matches(String regex)` , qui renvoie **`true`** si notre chaîne correspond à la regex ou **`false`**, dans le cas contraire.

Voici le code qui met en oeuvre cette démarche :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.text.ParseException;

import javax.swing.JButton;
import javax.swing.JFormattedTextField;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.text.MaskFormatter;


public class Fenetre extends JFrame {

    private JPanel container = new JPanel();
    private JFormattedTextField jtf;
    private JFormattedTextField jtf2;
    private JLabel label = new JLabel("Téléphone FR ");
    private JButton b = new JButton ("OK");

    /**
* Constructeur de l'objet
*/
    public Fenetre(){

        this.setTitle("Animation");
        this.setSize(300, 150);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());

        try{
            MaskFormatter tel = new MaskFormatter("**-**-**-**-**");
            jtf = new JFormattedTextField(tel);
        }catch(ParseException e){
         e.printStackTrace();
        }
```

```
            }

            JPanel top = new JPanel();

            Font police = new Font("Arial", Font.BOLD, 14);
            jtf.setFont(police);
            jtf.setPreferredSize(new Dimension(150, 30));
            jtf.setForeground(Color.BLUE);

            b.addActionListener(new BoutonListener());

            top.add(label);
            top.add(jtf);
            top.add(b);

            this.setContentPane(top);
            this.setVisible(true);
        }

        class BoutonListener implements ActionListener{
            public void actionPerformed(ActionEvent e) {
                    System.out.println("Téléphone FR : " +
jtf.getText());

                    if(jtf.getText().matches("^0[0-689](-
[\\d]{2}){4}$")){
                     System.out.println("Numéro de téléphone OK ! !");
                    }
                    else{
                     System.out.println("Numéro de téléphone PAS OK !
!");
                    }
            }
        }
}
```

Ainsi que deux captures d'écran afin de bien vous montrer le résultat :

Vous pouvez voir que c'est très simple à utiliser...
Je profite de cet aparté sur les regex afin d'introduire une autre méthode : `replaceAll(String regex, String remplacement)` .
Grâce à cette dernière, vous pourrez changer tous les caractères, ou chaînes de caractères correspondant à la regex passée en premier paramètre par la chaîne passée en deuxième paramètre.

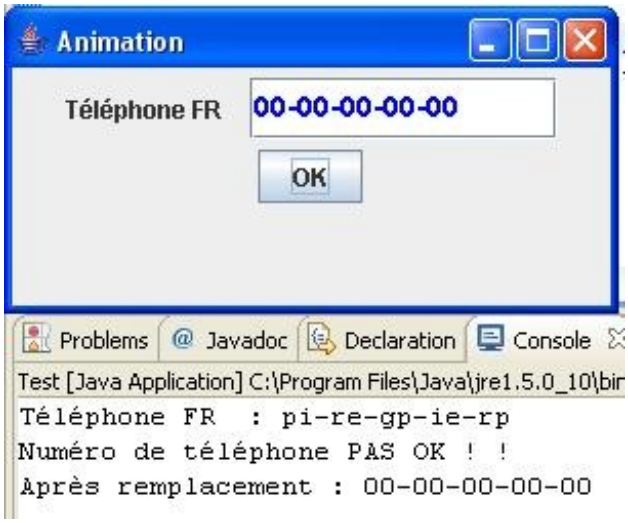
Si nous appliquons ceci à notre exemple, en partant du principe que, si la saisie du numéro de téléphone est erronée, on remplace tous les caractères par des zéros, cela nous donne :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.text.ParseException;

import javax.swing.JButton;
import javax.swing.JFormattedTextField;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.text.MaskFormatter;


public class Fenetre extends JFrame {

    private JPanel container = new JPanel();
    private JFormattedTextField jtf;
    private JFormattedTextField jtf2;
    private JLabel label = new JLabel("Téléphone FR ");
    private JButton b = new JButton ("OK");

    /**
* Constructeur de l'objet
*/
    public Fenetre(){

        this.setTitle("Animation");
        this.setSize(300, 150);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());

        try{
            MaskFormatter tel = new MaskFormatter("**-**-**-**-**");
            jtf = new JFormattedTextField(tel);
        }catch(ParseException e){
         e.printStackTrace();
        }

        JPanel top = new JPanel();

        Font police = new Font("Arial", Font.BOLD, 14);
        jtf.setFont(police);
        jtf.setPreferredSize(new Dimension(150, 30));
        jtf.setForeground(Color.BLUE);

        b.addActionListener(new BoutonListener());

        top.add(label);
        top.add(jtf);
        top.add(b);

        this.setContentPane(top);
        this.setVisible(true);
     }

    class BoutonListener implements ActionListener{
        public void actionPerformed(ActionEvent e) {
                System.out.println("Téléphone FR : " +
jtf.getText());

                if(jtf.getText().matches("^0[0-689](-
```

```
        [\\d]{2}){4}$")){
                            System.out.println("Numéro de téléphone OK ! !");
                           }
                           else{
                            System.out.println("Numéro de téléphone PAS OK !
        !");
                            //Si la saisie est erronée
                            //On remplace tous les caractères alphabétiques par
        des 0
                            String str = jtf.getText().replaceAll("\\w", "0");
                            jtf.setText(str);
                            System.out.println("Après remplacement : " + str);
                           }
                }
            }
        }
```

Et le résultat :



Je pense que cette méthode pourrait vous être utile : on ne sait jamais...

Maintenant, nous allons voir comment avoir le même résultat avec l'API regex.

Les regex et l'API regex

Avec cette méthode, nous allons utiliser deux nouveaux objets :

- un objet **`Pattern`** qui va contenir notre regex et qui va retourner un objet contenant le résultat de la comparaison ;
- l'objet **`Matcher`** qui est le résultat de la comparaison de la regex avec la chaîne à tester.

Vous pourrez voir que ces objets sont très simples à utiliser.
L'utilisation de l'objet **`Pattern`** se fait comme ceci :

```
Pattern pattern = Pattern.compile("^0[0-689](-[\\d]{2}){4}$");
```

Cette instruction déclare et initialise notre objet **`Pattern`**, celui-ci est maintenant prêt à tester des chaînes de caractères !

Le test d'une chaîne par rapport à une regex via l'objet **`Pattern`** se fait grâce à la méthode `matcher(String string)` : il ne s'agit pas de la regex en paramètre, mais de la chaîne à tester !

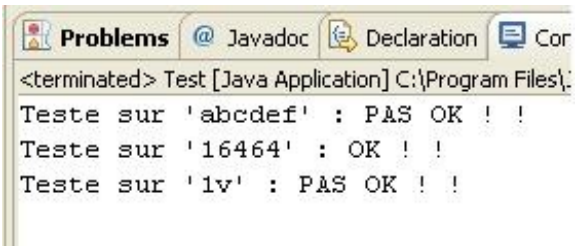
Comme je vous l'ai dit plus haut, la comparaison via l'objet **`Pattern`** renvoie un objet **`Matcher`** qui, lui, contient le résultat du test (vrai ou faux) que nous pourrons récupérer grâce à la méthode `matches()` .

Voici un exemple simple :

Code : Java

```
String[] tab = {"abcdef", "16464","1v"};
//Regex qui vérifie si la chaîne ne contient que des chiffres
Pattern pattern = Pattern.compile("\\d+");

for(String str : tab){
 Matcher matcher = pattern.matcher(str);
 System.out.print("Teste sur '"+str+"' : ");
 //On regarde le résultat
 if(matcher.matches())
  System.out.println("OK ! ! ");
 else
  System.out.println("PAS OK ! ! ");
}
```

Et voilà le résultat :



Rien de plus simple, n'est-ce pas ?

On voit bien que le résultat est le même, mais... l'intérêt ?

Je vais y venir, mais avant de vous expliquer pourquoi il est intéressant de passer par l'objet **`Pattern`**, vous devez savoir que vous pouvez ne pas utiliser l'objet **`Matcher`**.

Je vois bien que vous êtes un peu dans le flou...
Reprenez ce que je vous ai dit plus haut : **l'objet `Pattern` retourne un objet `Matcher`.**
Par conséquent, vous pouvez gagner un peu de mémoire en ne déclarant pas d'objet **`Matcher`** mais en vous servant de celui que vous retourne l'objet **`Pattern`** !

Voilà le code précédent mettant en oeuvre cette démarche :

Code : Java

```
String[] tab = {"abcdef", "16464","1v"};
//Regex qui vérifie si la chaîne ne contient que des chiffres
Pattern pattern = Pattern.compile("\\d+");

for(String str : tab){
 System.out.print("Teste sur '"+str+"' : ");
 //On regarde le résultat, et plus besoin d'instancier un objet
Matcher
 if(pattern.matcher(str).matches())
  System.out.println("OK ! ! ");
 else
  System.out.println("PAS OK ! ! ");
}
```

Je ne vous mets pas de capture d'écran car elle est identique à la précédente !

Tu ne voudrais pas nous expliquer ça ?

Bien sûr...

En fait, repensez à la pile d'exécution lorsque nous avons abordé les threads.
Ici, c'est la même chose. L'instruction `pattern.matcher(str).matches()` se découpe en deux.
Lors de l'exécution, la JVM va lire cette ligne, elle voit qu'il y a plusieurs appels de méthode : par conséquent, elle va invoquer celle qui doit être exécutée en premier, faire ce qu'elle a à faire, puis passer à la suivante...

Voilà un schéma résumant la situation :

La flèche indique le sens dans lequel la JVM va lire l'instruction et l'exécuter.

- Elle va lire le `pattern.matcher(str)` qui, comme je vous l'ai déjà dit, retourne un objet **`Matcher`**. Étape 1.
- Ensuite, elle va exécuter la méthode `matches()` qui est une méthode de l'objet **`Matcher`**. Étape 2.

Lors de l'étape 2, c'est comme si vous aviez un objet **`Matcher`** à la place de l'instruction correspondant à l'étape 1... La méthode `matches()` peut donc être invoquée !

Ainsi vous gagnez en objets, en lignes de codes et en mémoire...

Maintenant, la réponse à la question que vous vous posez :

Pourquoi utiliser l'objet `Pattern` alors que l'objet `String` gère les regex ?

En fait, les deux méthodes sont équivalentes...
C'est vrai que dans notre exemple, nous ne contrôlons qu'un champ. Mais ce ne sera peut-être pas toujours le cas...

Imaginez-vous en train de développer un progiciel de gestion avec, sur une de ses IHM, 35 champs de saisie qui doivent contenir des codes spécifiques à une norme... La solution des regex semble la plus optimisée mais **vous n'allez pas répéter la regex pour tous les contrôles de tous les champs ! !**
Le jour où votre chef va vous demander de mettre à jour ladite expression car un nouveau code vient de faire son apparition, vous allez sûrement oublier un ou plusieurs champs !

Le fait d'utiliser un objet **`Pattern`**, dans ce cas, permet de centraliser la donnée qui va vous servir à contrôler vos champs et, au lieu de faire X modifications, vous n'avez qu'à changer l'objet **`Pattern`**.

Mais il y a une autre alternative

Vous pouvez aussi stocker votre regex dans un objet de type **`String`** et utiliser ce dernier dans tous vos contrôles, en utilisant la méthode `matches(String regex)`. Le but final étant de centraliser les données dont vous vous servirez pour faire vos contrôles et que celles-ci soient facilement modifiables sans risque d'oubli.

En bref, ces deux méthodes sont équivalentes.
Je vous ai un peu induits en erreur, mais il était important que vous connaissiez l'API regex.

Vous devez savoir tout de même que lorsque vous utilisez la méthode `matches(String regex)` de l'objet **`String`**, celui-ci fait appel à l'objet **`Pattern`** dans cette méthode...

De même, lorsque vous utilisez la méthode `replaceAll(String regex, String remplacement)` , celle-ci invoque l'expression `Pattern.compile(regex).matcher(str).replaceAll(repl)` .

Pour finir sur l'utilisation des regex

Vous pouvez utiliser la méthode qui vous convient, mais gardez en tête qu'il faut que vos contrôles soient facilement modifiables !

Bon, vous venez de voir comment on peut gérer les saisies après les avoir tapées. Maintenant, je vous propose de voir comment intercepter les saisies des utilisateurs avant que votre composant ne soit affecté de quelque valeur que ce soit !

Retour en haut

Contrôle du clavier : l'interface KeyListener

Tout est dans le titre de cette sous-partie !
Vous connaissez déjà :

- l'interface **MouseListener** qui interagit avec votre souris ;
- l'interface **ActionListener** qui interagit lors d'un clic sur un composant ;
- l'interface **ItemListener** qui écoute les événements sur une liste déroulante.

Voici à présent l'interface **KeyListener**.
Comme dit dans le titre, celle-ci va vous permettre d'intercepter les événements clavier lorsqu'on :

- presse une touche ;
- relâche une touche ;
- tape sur une touche.

Vous savez ce qu'il vous reste à faire : créer un implémentation de cette interface dans notre projet.
Créez une classe interne implémentant cette interface et utilisez l'astuce d'Eclipse pour générer les méthodes à implémenter.

Vous constatez que celle-ci a trois méthodes :

- **keyPressed**(**KeyEvent** event) : appelée lorsqu'on presse une touche ;
- **keyReleased**(**KeyEvent** event) : appelée lorsqu'on relâche une touche. C'est à ce moment que le composant se voit affecter la valeur de la touche ;
- **keyTyped**(**KeyEvent** event) : appelée entre les deux méthodes citées ci-dessus.

Comme vous devez vous en douter, l'objet **KeyEvent** va nous permettre d'obtenir des informations sur les touches qui ont été utilisées... Parmi celles-ci, nous allons utiliser :

- **getKeyCode()** : retourne le code de la touche ;
- **getKeyChar()** : retourne le caractère correspondant à la touche.

Vous pouvez aussi savoir si certaines touches de contrôle ont été utilisées (SHIFT, CTRL...), connaître le composant à l'origine de l'événement... Nous n'en parlerons pas ici mais ce genre d'informations sont faciles à trouver : **Google**.

Pour des raisons de simplicité, nous n'allons pas utiliser de **JFormattedTextField** mais un **JTextField** sans **MaskFormatter**.
Ainsi, nous n'aurons pas à nous préoccuper des "**-**" de notre champ.

Pour commencer, nous allons voir dans quel ordre se passent les événements clavier.
Voici le code source que nous allons utiliser, il est presque identique aux précédents, rassurez-vous :

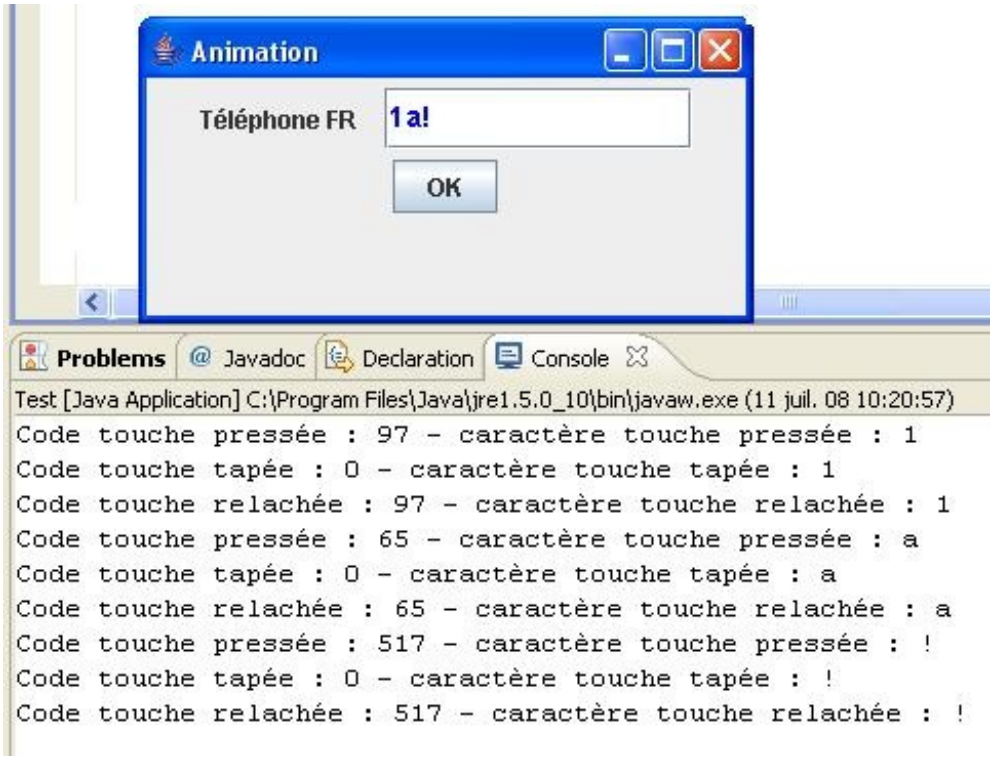
Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.text.ParseException;
import java.util.regex.Pattern;

import javax.swing.JButton;
import javax.swing.JFormattedTextField;
import javax.swing.JFrame;
import javax.swing.JLabel;
```

```
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.text.MaskFormatter;


public class Fenetre extends JFrame {

    private JPanel container = new JPanel();
    private JTextField jtf;

    private JLabel label = new JLabel("Téléphone FR ");
    private JButton b = new JButton ("OK");
    //Création de l'objet pattern dont nous allons nous servir pour
    //tester le contenu de notre champ
    private Pattern regex;

    /**
* Constructeur de l'objet
*/
    public Fenetre(){

     //On initialise notre pattern
     this.regex  = Pattern.compile("^0[0-689](-[\\d]{2}){4}$");

        this.setTitle("Animation");
        this.setSize(300, 150);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());

        jtf = new JTextField();
        JPanel top = new JPanel();

        Font police = new Font("Arial", Font.BOLD, 14);
        jtf.setFont(police);
        jtf.setPreferredSize(new Dimension(150, 30));
        jtf.setForeground(Color.BLUE);
        //On ajoute l'écouteur à notre composant
        jtf.addKeyListener(new ClavierListener());

        b.addActionListener(new BoutonListener());

        top.add(label);
        top.add(jtf);
        top.add(b);

        this.setContentPane(top);
        this.setVisible(true);
     }

    class BoutonListener implements ActionListener{
        public void actionPerformed(ActionEvent e) {
                System.out.println("Téléphone FR : " +
jtf.getText());
                if(regex.matcher(jtf.getText()).matches()){
                 System.out.println("Numéro de téléphone OK ! !");
                 String str = jtf.getText().replaceAll("\\d", "X");
                 System.out.println("Après remplacement : " + str);
                }
                else{
                 System.out.println("Numéro de téléphone PAS OK !
!");
                 //Si la saisie est erronée
                 //On remplace tous les caractères alphabétiques par
des 0
                 String str = jtf.getText().replaceAll("\\w", "0");
                 jtf.setText(str);
                 System.out.println("Après remplacement : " + str);
```

```
                    }
                }
            }

            /**
         * Implémentataion de l'interface KeyListener
         * @author CHerby
         */
            class ClavierListener implements KeyListener{

          public void keyPressed(KeyEvent event) {
           System.out.println("Code touche pressée : " + event.getKeyCode()
        +
                " - caractère touche pressée : " + event.getKeyChar());
          }

          public void keyReleased(KeyEvent event) {
           System.out.println("Code touche relâchée : " + event.getKeyCode()
        +
             " - caractère touche relâchée : " + event.getKeyChar());

          }

          public void keyTyped(KeyEvent event) {
           System.out.println("Code touche tapée : " + event.getKeyCode() +
             " - caractère touche tapée : " + event.getKeyChar());
          }
            }
        }
```

Voici un petit jeu d'essai de ce code :



C'est vrai que les événements vont tellement vite que n'avez pas le temps de voir le défilement.
Vous pouvez ajouter une pause à la fin de chaque méthode de l'implémentation afin de mieux voir l'ordre d'exécution, comme ceci :

Code : Java

```
    class ClavierListener implements KeyListener{
```

```
    public void keyPressed(KeyEvent event) {
     System.out.println("Code touche pressée : " + event.getKeyCode()
+
           " - caractère touche pressée : " + event.getKeyChar());
     pause();
    }

    public void keyReleased(KeyEvent event) {
     System.out.println("Code touche relâchée : " + event.getKeyCode()
+
       " - caractère touche relâchée : " + event.getKeyChar());
     pause();
    }

    public void keyTyped(KeyEvent event) {
     System.out.println("Code touche tapée : " + event.getKeyCode() +
       " - caractère touche tapée : " + event.getKeyChar());
     pause();
    }
      }

      private void pause(){
       try {
     Thread.sleep(1000);
    } catch (InterruptedException e) {
     // TODO Auto-generated catch block
     e.printStackTrace();
    }
      }
```

Maintenant, vous pouvez voir dans quel ordre les événements du clavier sont gérés.
En premier lorsqu'on presse la touche, en second lorsque celle-ci est tapée et enfin relâchée.

Dans le cas qui nous intéresse, nous voulons que lorsque l'utilisateur saisit un caractère non autorisé, celui-ci soit retiré automatiquement de la zone de saisie. Pour cela, nous allons faire un traitement spécifique dans la méthode **keyReleased(KeyEvent event)**.

Si vous avez fait beaucoup de tests de touches, vous avez dû remarquer que les codes des touches correspondant aux chiffres du pavé numérique sont compris entre **96** et **105** !

À partir de là, c'est simple : il vous suffit de supprimer le caractère tapé de la zone de saisie si le code de celui-ci n'est pas compris dans cette tranche de code.
Mais voilà, un problème se pose avec cette méthode : pour celles et ceux qui ont un PC portable, sans pavé numérique, la saisie sera impossible alors que vous pouvez avoir des chiffres en faisant **MAJ** + (**&** ou **é** ou **'** ou **(** ou encore **-** ...).

À cause de ce souci, nous allons opter pour une autre méthode. Nous allons créer une méthode ayant comme type de renvoi un booléen et qui va se charger de nous dire si la saisie est numérique ou non. Comment ? Tout simplement en faisant un `Integer.parseInt(value);`, le tout enveloppé dans un **try{...} catch(NumberFormatException ex){}**.
Si nous essayons de convertir un caractère "a" en entier, l'exception sera levée et nous retournerons FAUX, et VRAI dans le cas contraire...

Attention : la méthode **parseInt()** prendra un **String** en paramètre ! La méthode **getKeyChar()**, elle, nous renvoie un **char**... Il faudra penser à faire la conversion...

Voici notre implémentation quelque peu modifiée :

Code : Java

```
class ClavierListener implements KeyListener{

  public void keyReleased(KeyEvent event) {

   if(!isNumeric(event.getKeyChar())){

jtf.setText(jtf.getText().replace(String.valueOf(event.getKeyChar()),
""));
    }
  }

  //Inutile de redéfinir ces méthodes
  //Nous n'en avons plus l'utilité !
  public void keyPressed(KeyEvent event) {}
  public void keyTyped(KeyEvent event) {}

  /**
* Retourne vrai si le paramètre est numérique
* Retourne Faux dans le cas contraire
* @param carac
* @return Boolean
*/
  private boolean isNumeric(char carac){
      try {
    Integer.parseInt(String.valueOf(carac));
   } catch (NumberFormatException e) {
    return false;
   }
   return true;
     }
    }
```

Vous pouvez voir que les lettres simples sont désormais interdites à la saisie => **Mission accomplie !**

Les caractères spéciaux comme "ô", "ï"... ne sont pas pris en charge par cette méthode... Par conséquent, leur saisie reste possible. Mais c'est à ça que sert notre contrôle avec la regex.

Par contre, je ne sais pas pour vous mais, le fait d'avoir deux méthodes sans corps me dérange un peu...

On peut éviter ce genre de chose ?

Comment ? Puisque nous devons redéfinir toutes les méthodes de l'interface !

Tout à fait. Il existe une classe, **KeyAdapter**, que vous pouvez étendre (par là comprenez : créez une classe héritée) et ne redéfinir que la méthode qui nous intéresse, et donc ADIEU aux deux méthodes vides !

Vous pouvez bien entendu créer un classe interne héritée de **KeyAdapter** et redéfinir la méthode **keyReleased(KeyEvent event)** mais je vais en profiter pour vous montrer une autre méthode.

Utiliser les classes anonymes

Il n'y a rien de compliqué dans cette manière de faire, mais je me rappelle avoir été déconcerté lorsque je l'ai vue pour la première fois...

En fait, les classes anonymes sont le plus souvent utilisées pour la gestion d'événements ponctuels, là où créer une classe pour un seul traitement est trop lourd...

Notre exemple est très bien pour les classes anonymes : nous n'avons qu'un champ et une redéfinition de méthode. Maintenant, adieu à l'implémentation que vous avez codée tout à l'heure, nous allons dire à notre **JTextField** qu'une instance d'une classe anonyme va l'écouter. Attention les yeux, ça risque de piquer un peu :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.util.regex.Pattern;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;


public class Fenetre extends JFrame {

    private JPanel container = new JPanel();
    private JTextField jtf;

    private JLabel label = new JLabel("Téléphone FR ");
    private JButton b = new JButton ("OK");
    //Création de l'objet pattern dont nous allons nous servir pour
    //tester le contenu de notre champ
    private Pattern regex;

    /**
* Constructeur de l'objet
*/
    public Fenetre(){

     //On initialise notre pattern
     this.regex  = Pattern.compile("^0[0-689](-[\\d]{2}){4}$");

        this.setTitle("Animation");
        this.setSize(300, 150);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());

        jtf = new JTextField();
        JPanel top = new JPanel();

        Font police = new Font("Arial", Font.BOLD, 14);
        jtf.setFont(police);
        jtf.setPreferredSize(new Dimension(150, 30));
        jtf.setForeground(Color.BLUE);

        //*********************************************
        //Voilà notre classe anonyme
        //*********************************************
        jtf.addKeyListener(new KeyAdapter(){

         public void keyReleased(KeyEvent event) {
       System.out.println("keyReleased dans une classe anonyme");
       if(!isNumeric(event.getKeyChar())){

jtf.setText(jtf.getText().replace(String.valueOf(event.getKeyChar()),
""));
        }
      }
```

```
        private boolean isNumeric(char carac){
            try {
          int i =Integer.parseInt(String.valueOf(carac));
         } catch (NumberFormatException e) {
          return false;
         }
         return true;
           }

          });
          //*******************************************

          b.addActionListener(new BoutonListener());

          top.add(label);
          top.add(jtf);
          top.add(b);

          this.setContentPane(top);
          this.setVisible(true);
      }

     class BoutonListener implements ActionListener{
         public void actionPerformed(ActionEvent e) {
                  System.out.println("Téléphone FR : " +
jtf.getText());
                  if(regex.matcher(jtf.getText()).matches()){
                   System.out.println("Numéro de téléphone OK ! !");
                   String str = jtf.getText().replaceAll("\\d", "X");
                   System.out.println("Après remplacement : " + str);
                  }
                  else{
                   System.out.println("Numéro de téléphone PAS OK !
!");
                   //Si la saisie est erronée
                   //On remplace tous les caractères alphabétiques par
des 0
                   String str = jtf.getText().replaceAll("\\w", "0");
                   jtf.setText(str);
                   System.out.println("Après remplacement : " + str);
                  }
         }
     }
}
```

Ce code a le même effet que le précédent : la seule chose qui change, c'est qu'au lieu d'avoir une implémentation de l'interface **KeyListener** ou d'avoir une classe interne héritée de **KeyAdapter**, nous utilisons une classe anonyme au moment où nous définissons l'écouteur pour notre composant.

Décortiquons tout ça...
Nous avons toujours notre instruction `jtf.addKeyListener();`, sauf qu'au lieu de lui donner une instance habituelle, nous créons une classe qui redéfinit la méthode qui nous intéresse. Ceci en faisant :

Code : Java

```
new KeyAdapter(){
 //Redéfinition de la classe
};
```

De ce fait, vous pouvez aussi créer une instance de type **KeyAdapter** en utilisant une classe interne comme implémentation :

Code : Java

```
KeyAdapter kAdapter = new KeyAdapter(){
 //Redéfinissions de la classe
};
jtf.addKeyListener(kAdapter);
```

Attention : vous avez dû le remarquer mais je préfère le dire, dans ce type de déclaration, le "**;**" final se trouve **après l'accolade fermante de la classe anonyme ! !**

L'une des particularités de cette façon de faire, c'est que cet écouteur n'écoutera que ce composant !

Pourquoi on appelle ça une classe ***anonyme*** ?

C'est simple : le fait de procéder de cette manière revient à créer une classe héritée sans être obligés de créer de façon explicite ladite classe.
L'héritage se fait automatiquement, en fait, le code ci-dessus reviendrait à faire :

Code : Java

```
class Fenetre extends JFrame{

  //...
  jtf.addKeyListener(new KeyAdapterBis());

  //...

   public class KeyAdapterBis extends Keyadapter{

        public void keyReleased(KeyEvent event) {
       System.out.println("keyReleased dans une classe anonyme");
       if(!isNumeric(event.getKeyChar())){

jtf.setText(jtf.getText().replace(String.valueOf(event.getKeyChar()),
""));
       }
      }

      private boolean isNumeric(char carac){
          try {
        int i =Integer.parseInt(String.valueOf(carac));
       } catch (NumberFormatException e) {
        return false;
       }
       return true;
         }

   }
}
```

Mais la classe créée n'a pas de nom ! L'héritage se fait de façon tacite. On bénéficie donc de tous les avantages de la classe mère en ne redéfinissant que la méthode qui nous intéresse.

Vous devez savoir aussi que les classes anonymes peuvent être utilisées pour implémenter des interfaces. Ce code est tout aussi équivalent aux précédents :

Code : Java

```
jtf.addKeyListener(new KeyListener(){
```

```
            public void keyReleased(KeyEvent event) {
          System.out.println("keyReleased dans une classe anonyme");
          if(!isNumeric(event.getKeyChar())){

    jtf.setText(jtf.getText().replace(String.valueOf(event.getKeyChar()),
    ""));
          }
        }

       private boolean isNumeric(char carac){
           try {
         int i =Integer.parseInt(String.valueOf(carac));
         } catch (NumberFormatException e) {
         return false;
         }
         return true;
           }

              //Méthode de l'interface a redéfinir
              public void keyPressed(KeyEvent e) {}
     public void keyTyped(KeyEvent e) {}

        });
```

Les classes anonymes sont soumises aux mêmes lois que les classes ***normales*** :

- utilisation des méthodes non redéfinies de la classe mère ;
- OBLIGATION de redéfinir TOUTES LES MÉTHODES d'une interface ;
- OBLIGATION de redéfinir les méthodes abstraites d'une classe abstraite.

Cependant, elles ont des restrictions de par leur essence et par là, je veux dire leur rôle et leur but :

- ces classes ne peuvent pas être déclarées **abstract** !
- elles ne peuvent pas non plus être **static** ;
- elles ne peuvent pas définir de constructeur ;
- elles sont automatiquement déclarées **final** : impossible de dériver de cette classe, donc héritage impossible !

Encore une chose avant de terminer ce chapitre sur le JTextField : il existe encore deux objets fonctionnant de la même manière :

- le **JPasswordField** : utilisé pour les saisies de mots de passe ;
- le **JTextArea** : utilisé, lui, pour les saisies multilignes.

Essayez-les, vous verrez que leur utilisation est très simple.

Bon, après toutes ces émotions, je crois qu'un petit topo s'impose...

Retour en haut

Ce qu'il faut retenir

- Par défaut, un `JTextField` accepte tous types de caractères.
- Un `JFormattedTextField` est, pour simplifier, un `JTextField` plus restrictif.
- On peut restreindre la saisie d'un de ces objets en utilisant l'objet `MaskFormatter`.
- Pour contrôler les événements clavier, l'utilisation d'une implémentation de l'interface `KeyListener` est de mise.
- Vous pouvez utiliser une classe dérivée de `KeyAdapter` à la place d'une implémentation de `KeyListener`.
- Une classe anonyme est propre à un objet.
- Une classe anonyme est automatiquement déclarée `final` : donc pas d'héritage possible.
- Vous pouvez utiliser les regex avec l'objet `String` ou avec l'objet `Pattern`.

Retour en haut

Vous avez vu que cet objet est très simple à utiliser...
La plupart des composants que vous pouvez utiliser dans un formulaire de base ont été vus.
Je vous propose donc de faire un détour vers ce qu'on appelle **les Applets**.

Retour en haut

Les applets

Dans ce chapitre, vous apprendrez tout ce qu'il y a savoir sur les applets :

- ce qu'elles sont ;
- comment les créer ;
- comment les intégrer dans une page web ;
- faire interagir votre applet via du Javascript ;
- appeler du Javascript via votre applet ;
- ...

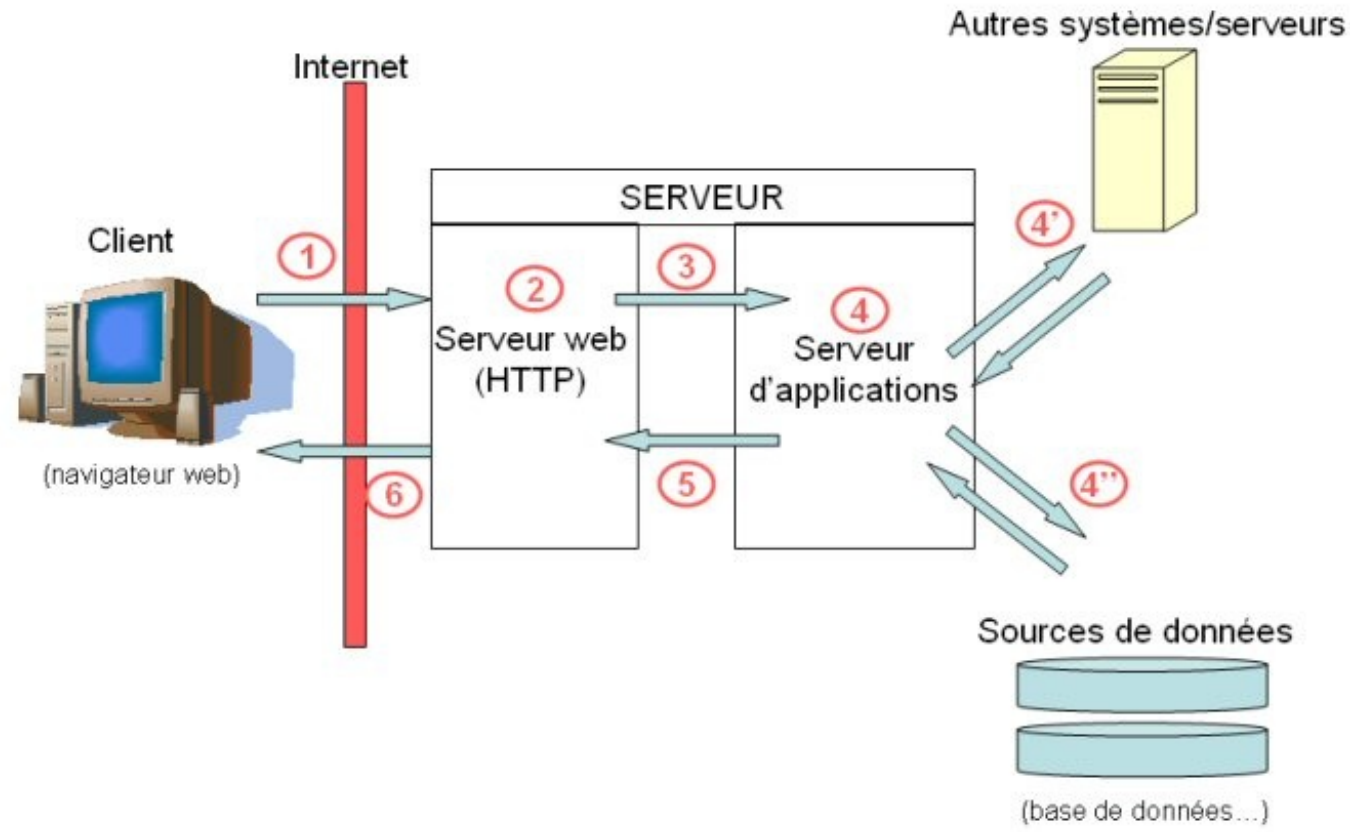
Par contre, afin de suivre ce chapitre dans les meilleures conditions, je vous conseille d'avoir des bases en Javascript.
Pour ce faire, vous pouvez utiliser ce tuto.
Vous pouvez le lire jusqu'au chapitre 2 de la partie 2, au moins !

Je sens que vous êtes impatients de commencer, alors allons-y !

Les applets : quésaco ?

Un applet est une application Java insérée dans une page web.
Ce genre d'application est appelée *application client* car elle est utilisée par celui qui appelle la page web (le client), et non par celui qui la génère (le serveur).

Il ne faut pas confondre un applet et une application J2EE (maintenant JEE). Pour bien comprendre à quel niveau se situe l'applet dans le cycle de vie d'une page web, un petit schéma s'impose. Ce schéma représente le cycle de vie d'une page web dite dynamique, c'est-à-dire que celle-ci contient du code interprété par le serveur (Java, PHP, ASP...) qui est ici une page contenant du code Java :



- **(1) Le client émet une requête** (saisie d'une URL, clic sur un lien...) pour demander une ressource au serveur. Exemple : ***http://www.monserveur.com/tuto.do***. Il ne sait pas ici si la réponse qui va lui parvenir est statique (page HTML simple) ou dynamique (générée par une application web). Dans notre cas, il s'agit d'une application répondant à l'adresse "***tuto.do***" sur le serveur "***monserveur.com***".
- **(2)** Côté serveur, **c'est le serveur web (exemple : Apache) qui traite les requêtes HTTP** entrantes. Il traite donc toutes les requêtes, qu'elles demandent une ressource statique ou dynamique. Seulement, un serveur HTTP ne sait répondre qu'aux requêtes visant des ressources statiques. **Il ne peut que renvoyer des pages HTML, des images, des applets... existantes**.
- **(3)** Ainsi, si le serveur HTTP s'aperçoit que la requête reçue est destinée au serveur d'applications, il la lui transmet. Les deux serveurs sont reliés par un canal, nommé "***connecteur***".
- **(4) Le serveur d'applications (exemple : Tomcat ! Serveur d'applications Java) reçoit la requête à son tour**. Il est, lui, en mesure de la traiter. Il exécute donc le morceau d'application (la *servlet*) auquel est destinée la requête, en fonction de l'URL. Cette opération est effectuée à partir de la configuration du serveur. La servlet est donc invoquée, et le serveur lui fournit notamment deux objets Java exploitables : un représentant la requête, l'autre représentant la réponse. **La servlet peut maintenant travailler, et générer la réponse à la demande**. Cela peut passer par la consultation de sources de données, comme des bases de données (4'' sur le schéma). Ou bien par l'interrogation d'autres serveurs ou systèmes (4' sur le schéma), l'environnement Java web permettant de se connecter à de nombreux systèmes.
- **(5) Une fois sa réponse générée, le serveur d'applications la renvoie, par le connecteur, au serveur web**. Celui-ci la récupère comme s'il était lui-même allé chercher une ressource statique. Il a simplement délégué la récupération de la réponse, et celle-ci a été générée, mais ce n'est plus le problème.
- **(6) La réponse est dorénavant du simple code HTML**, compréhensible par un navigateur. Le serveur HTTP peut donc retourner la réponse au client. Celle-ci contient toutes les ressources nécessaires (feuilles Javascript, feuilles CSS, applet Java, images...).

Les servlets sont en fait des classes Java. Celles-ci peuvent être des classes Java programmées par un développeur ou une page JSP (page web contenant du code Java) compilée en servlet par le serveur d'application avant traitement par celui-ci. Nous reviendrons sur tous ces points plus tard, nous n'avons pas besoin d'en savoir plus pour le moment...

Ceci est un résumé du cycle de vie d'une page web faite avec la technologie J2EE. Je me doute que vous devez avoir des sueurs froides, mais ne vous inquiétez pas, nous reverrons tout ça plus en détail lorsque nous aborderons le développement web en Java...
Pour le moment, tout ce que vous avez besoin de savoir c'est qu'**un applet est une ressource utilisée par votre navigateur, tout comme une image : à la différence que là, il s'agit d'un programme qui va s'exécuter sur votre page !**

Les ressources utilisées par votre navigateur pour charger et utiliser un applet sont chargées au chargement de la page, après que le serveur web ait renvoyé la réponse à votre requête.
Ces ressources sont dans le code source HTML de la page et le navigateur charge tout ce dont il a besoin pour afficher la page comme le développeur l'a souhaité (images, feuilles CSS, applet...).

Vous pouvez voir un aperçu de notre animation version applet par ici.
Vous constaterez que le programme fonctionne parfaitement, comme s'il s'agissait d'une application fenêtrée, mais là, notre programme Java s'exécute sur une page web !

Attention : il se peut que votre navigateur n'autorise pas l'exécution des applets Java !
Pour remédier à ce problème, vous devez allez dans les options internet : ***menu Outils > Options*** dans l'onglet **Contenu** : cochez "***autoriser le Java***", sous Firefox.
Sous IE 7, faites : ***Outils > Options internet***, dans l'onglet "***content***", cochez la case "***utiliser JRE 1.X.XX pour les applets***" où X.XX correspond à la version de votre JRE installé sur votre machine.

Maintenant, vous savez distinguer une application client d'une application serveur et donc, vous ne devrez plus faire l'amalgame entre applet et servlet !
Retour en haut

Votre première applet

Il est temps maintenant de faire votre première applet.
Vous allez voir que c'est très simple et que ça ressemble beaucoup à ce que vous avez fait jusque-là. En fait, c'est quasiment identique à une exception près :
un applet n'a pas de constructeur mais elle utilise la méthode `init()` de la super-classe `Applet` du package `awt` ou `JApplet` du package `swing`.

Codage de l'applet

Nous allons faire un applet avec un code minimal, disons un label et un bouton. Lors du clic sur `bouton`, nous afficherons le nombre de clics effectués. Rien de bien méchant. Créez un nouveau projet avec une classe `FirstApplet` héritée de `JApplet`.

Voici le code source de votre première applet :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JLabel;


public class FirstApplet extends JApplet {

 private JLabel label = new JLabel();
 private JButton bouton = new JButton("Cliquez");
 private int count = 0;
 /**
* Méthode d'initialisation de l'applet
* C'est cette méthode qui fait office de constructeur
*/
 public void init(){
  this.setSize(300, 80);

  //On centre le texte du JLabel et on écrit en bleu...
  label.setHorizontalAlignment(JLabel.CENTER);
  //C'est plus zoli.
  label.setForeground(Color.blue);

  //Allez, une classe anonyme... Just for the fun ;)
  this.bouton.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent arg0) {
    label.setText("Vous avez cliqué " + (++count) + " fois sur le
bouton");
   }
  });

  //On ajoute nos composants
  this.getContentPane().add(bouton, BorderLayout.SOUTH);
  this.getContentPane().add(label, BorderLayout.NORTH);
  //Et le tour est joué !
 }


}
```

Avant de vous lancer dans le test de cette première applet, vous devez savoir tout de même que, mis à part quelques méthodes comme `setTitle("Animation")` , `setLocationRelativeTo(null)` et quelques autres propres aux objets

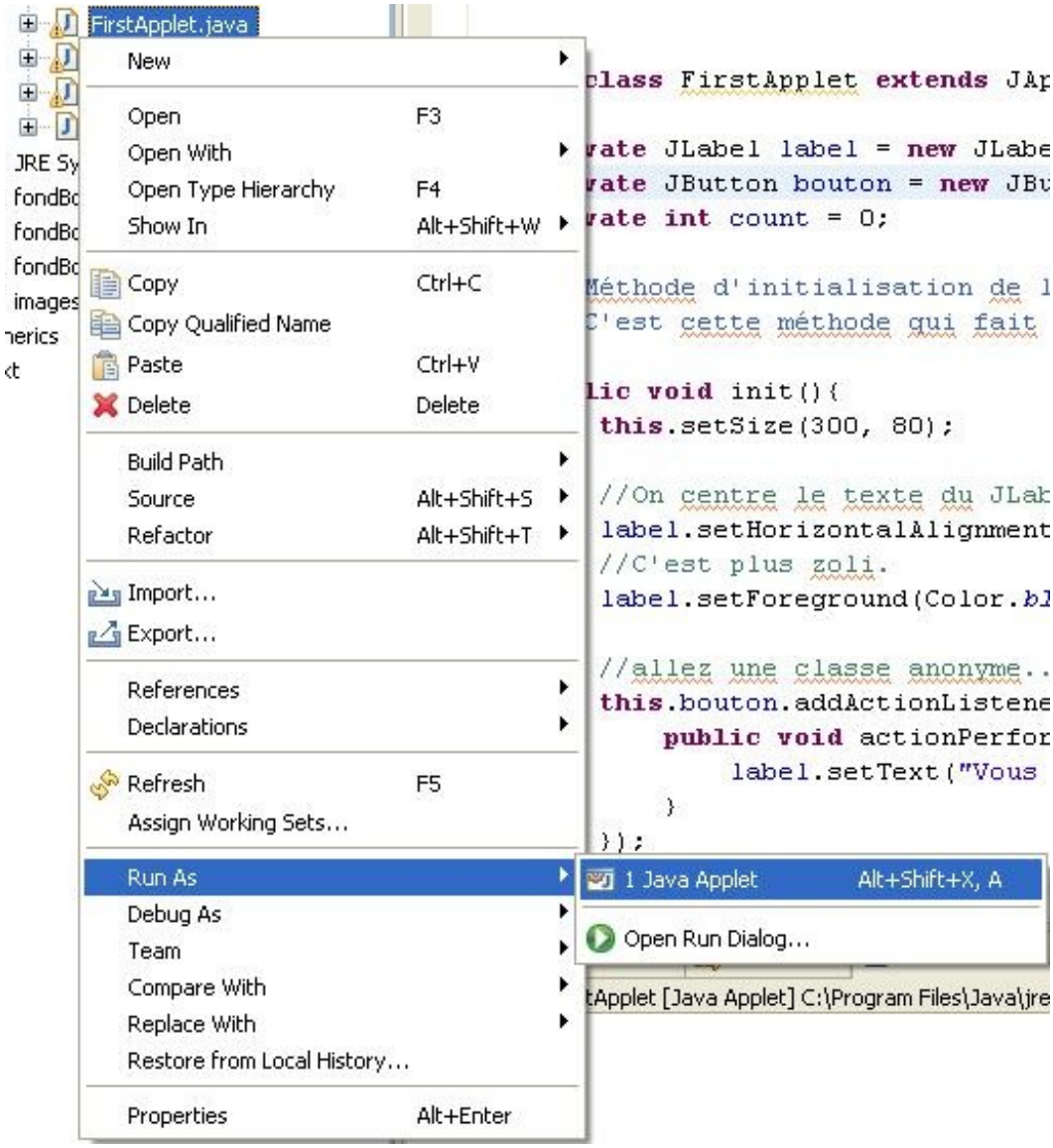
JFrame, les applets s'utilisent de la même manière. Bien sûr, avec la méthode `public void init()` à la place d'un constructeur !

Maintenant, avant d'intégrer votre programme dans une page web, vous devez vous assurer que celui-ci fonctionne correctement.

Comment on fait ça ? Tu nous a dit que les applets étaient utilisées dans des pages web...

Je maintiens, mais Eclipse vous propose un moyen d'exécuter votre classe comme un applet.

Pour faire cela, faites un clic droit sur votre fichier puis choisissez "**Run as**" et enfin choisissez "**Java Applet**", comme ceci :



Vous pouvez voir, ébahis, le résultat de votre applet :

Vous avez un indicateur vous signalant si votre applet est lancé ou non. Si vous voyez le message : "***Applet not inited***", ça veut dire qu'une erreur s'est glissée dans votre code et que la JVM n'a pas pu initialiser l'applet !

Insertion dans une page HTML

Pour que votre navigateur sache que la ressource à utiliser est un applet Java, vous devez utiliser la balise HTML **`<applet></applet>`** .
Celle-ci peut prendre plusieurs attributs et vous pouvez même passer des paramètres à votre applet grâce à cette balise.

Voici la liste des paramètres que peut prendre la balise **`<applet></applet>`** :

- `name="FirstAnimation"` : nom de l'applet, ici `FirstAnimation` . Nous allons vite voir l'intérêt de cet attribut ;
- `width="300px"` : largeur de l'applet affiché, ici, 300 pixels ;
- `height="300px"` : hauteur de l'applet affiché ;
- `codebase="class/"` : l'URL de base pour l'applet, c'est-à-dire l'endroit où le navigateur peut trouver les fichiers .class ; ici, les fichiers sont dans le dossier ***class*** à côté de votre fichier HTML ;
- `code="FirstAnimation.class"` : fichier de classe de l'applet. Celui où se trouve la méthode `init()` ;
- `archive="plugin.jar"` : identifie les ressources à pré-charger (.jar, images...) ; ici, nous pré-chargeons une archive Java appelée plugin.jar ;
- `alt="Please Wait..."` : affiche un texte au cours du chargement ;
- `hspace="10px"` : espacement horizontal entre l'applet et un autre contenu (`div` ou autre bloc HTML...) ;
- `vspace="20px"` : idem que précédemment mais pour l'espacement vertical.

Pfiou... Il y en a des attributs pour cette balise...
Voici donc un exemple de balise applet :

Code : HTML

```
<applet name="FirstAnimation" codebase="class/"
code="FirstAnimation.class"
        height="300px" width="300px" archive="plugin.jar">
   <param name="message" value="Message pour les ZérOs">
</applet>
```

Heu...
On comprend bien tout mais, qu'est-ce que c'est que ce truc : **`<param`** `name="message" value="Message pour les ZérOs">` ?

Je vous ai dit que vous pouviez passer des paramètres à votre applet. Eh bien c'est comme ceci que nous allons nous y prendre !
Ceci veut dire que nous pourrons utiliser la méthode `getParameter(String paramName);` qui va nous renvoyer un `String` correspondant à l'attribut `value` de la balise. Ici, on aurait `this.getParameter("message")//Retourne : Message pour les ZérOs.`

Tenez, nous allons essayer ça ! Ajoutez ceci dans votre méthode `init()` : `System.out.println("Paramètre passé via la balise <param> : " + this.getParameter("message"));` .

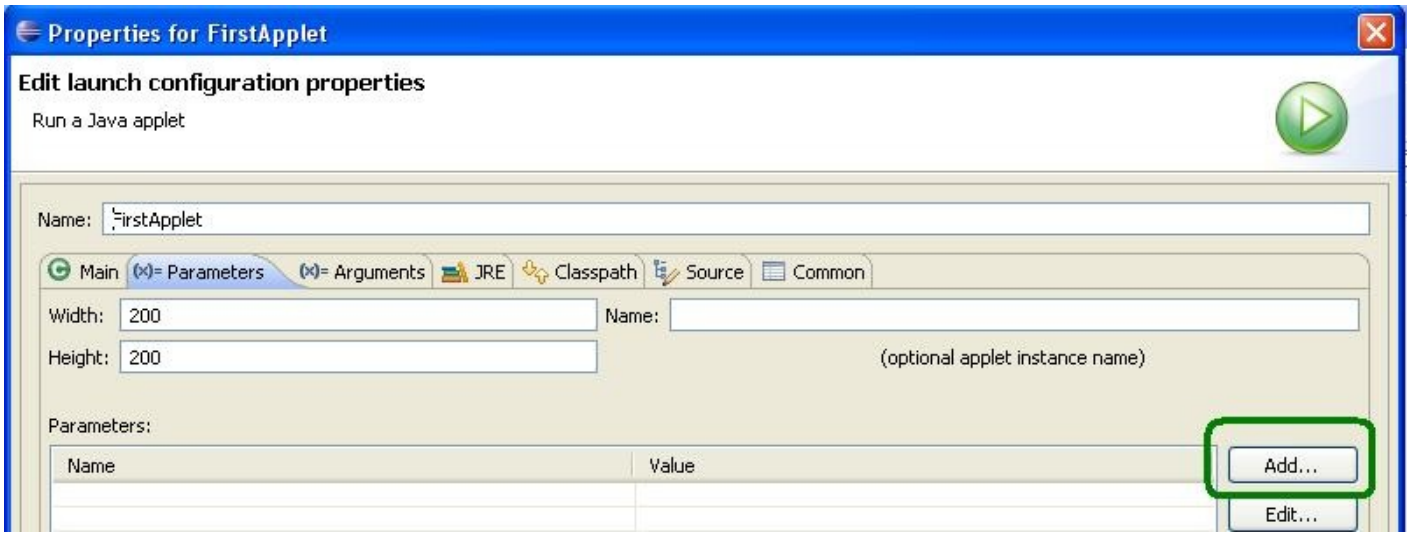
Lancez votre applet et :

Ah oui ! Si vous ne spécifiez pas de paramètre pour votre applet, ledit paramètre vaut **null**.

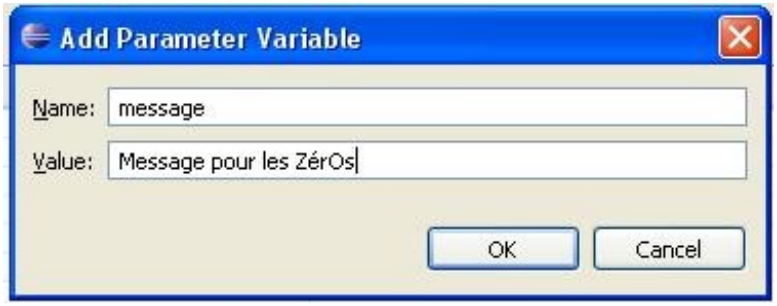
Voici comment on procède pour spécifier un paramètre pour votre application. Déjà, faites un clic droit sur votre fichier puis allez dans **Propriétés**.
Ensuite, cliquez sur "**Run/Debug settings**" puis sur le fichier correspondant à votre applet et enfin sur "**Edit**", comme ceci :

Dans la fenêtre dans laquelle vous êtes maintenant, choisissez l'onglet "**parameter**".

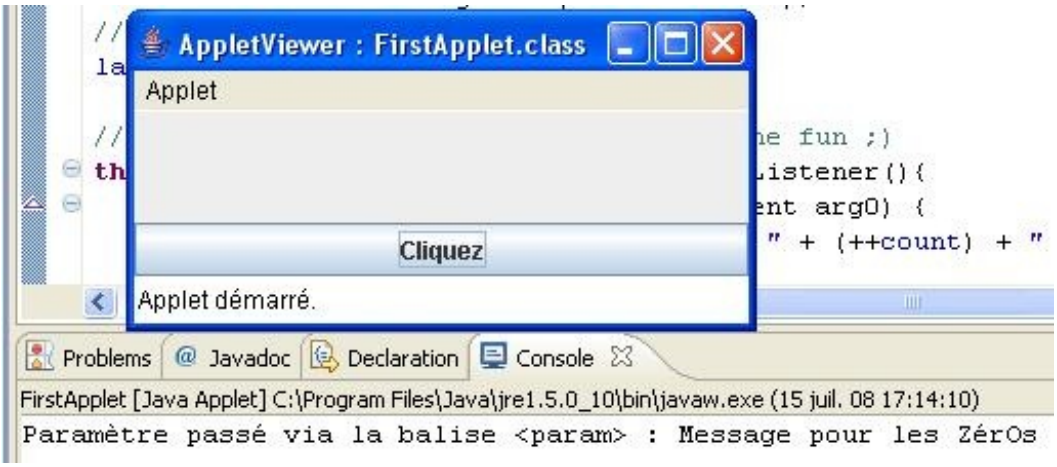
Vous arrivez enfin à l'endroit où vous pouvez ajouter des paramètres. Cliquez sur "**Add**", comme ceci :



Et enfin, renseignez le nom de votre paramètre ainsi que sa valeur :



Cliquez sur "**Ok**" sur toutes les fenêtres que vous avez ouvertes et relancez votre applet, le paramètre s'affiche enfin !



Vous pouvez maintenant incorporer votre applet dans une page HTML.
Voici le code de ladite page :

Code : HTML

```
<html>

   <body style="margin:auto;">

 <div style="width:320px;margin:auto;margin-top:100px;border:5px
solid black">
  <applet codebase="class/" code="FirstApplet.class" height="80"
width="300" hspace="10" vspace="10">
   <param name="message" value="Message pour les ZérOs">
```

```
    </applet>
  </div>

    </body>
</html>
```

J'ai créé ce code et enregistré le fichier contenant ce code HTML sur mon bureau, j'ai donc dû déplacer mes fichiers .class - oui, vous ne rêvez pas, j'ai dit : **mes fichiers .class** - dans un dossier, que j'ai appelé "***class***" pour l'occasion (cf. paramètre codebase de l'applet)...

J'ai récupéré mes fichiers .class dans le répertoire ***/bin*** de votre projet, et vous pouvez voir que vous avez **FirstApplet.class** et **FirstApplet$1.class** dans le cas où vous avez exactement le même code que moi.

À quoi ça correspond ?

En fait, **FirstApplet.class** correspond à la compilation de votre classe **`FirstApplet`** et **FirstApplet$1.class** correspond à la compilation de votre classe anonyme ! Pas de nom pour cette classe, donc la JVM remplace le nom par "***$1***". Si vous aviez utilisé une classe interne, appelons-la `BoutonListener` et si vous compiliez le code, vous auriez toujours **FirstApplet.class**, mais vous auriez eu le fichier **FirstApplet$BoutonListener.class**.

Donc, si vous avez créé votre page web ailleurs que dans votre dossier contenant votre projet, vous devrez déplacer tous les fichiers .class commençant par **`FirstApplet`** et toutes les autres ressources que votre applet utilise (images, autres classes, archives Java...).

Vous pouviez aussi créer votre page web dans le dossier de votre projet et spécifier comme codebase "**`bin/`**", dossier contenant vos .class dans le projet d'Eclipse... C'est à votre bon vouloir !

Maintenant que toute la lumière est faite sur ce point, vous pouvez aller voir votre première applet : ici.

Il faut que vous sachiez que, si Eclipse est assez laxiste pour lancer l'applet même si le paramètre "**`message`**" n'est pas renseigné, votre navigateur, enfin la JVM de votre navigateur sera moins conciliante... **Si le paramètre est manquant, l'applet plantera !**

Voilà, vous venez de faire votre première applet ! Alors, heureux ?

Nota Bene

Avant de continuer, vous devez savoir une dernière chose, ceci ne concerne pas directement Java mais la balise **`<applet></applet>`** .
En fait, depuis la sortie de HTML 4.0, la balise **`<applet></applet>`** est dépréciée par le W3C, c'est-à-dire que cet organisme préconise l'utilisation de la balise **`<object></object>`** .

Ceci en grande partie à cause de IE qui gérait le Java avec sa propre JVM (version 1.1, c'est vieux...) et non celle de Sun Microsystems (bientôt 1.7...). Il faut, afin que la balise **`<applet></applet>`** fonctionne correctement sous IE, avoir installé un environnement Java et s'assurer que IE utilise celui-ci pour interpréter du Java... (cf. plus haut).

Je ne détaillerai pas l'utilisation de cette balise vu que Sun Microsystems recommande l'utilisation de la balise **`<applet></applet>`** ... Voici tout de même un exemple d'utilisation de la balise **`<object></object>`** :

Code : HTML

```
<!-- L'Utilisation des commentaires conditionnels propres à IE sont
à utiliser -->
<!-- car même si IE requiert l'utilisation de cette balise, il ne
l'interprète pas comme les autres -->
<!--[if IE]>
```

```
        <object classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
        width="300" height="80" name="FirstApplet">
        <param name="java_code" value="FirstApplet.class" />
        <param name="java_codebase" value="class/" />
        <param name="type" value="application/x-java-applet;version=1.5" />

        <![endif]-->
        <p>
        <object classid="java:FirstApplet.class"
                  codebase="class"
                  type="application/x-java-applet"
                  width="300" height="80">
            <param name="code" value="FirstApplet" />
            <!-- Safari a besoin de ces paramètres -->
            <param name="JAVA_CODEBASE" value="class" />
            <param name="BGCOLOR" value="000000" />
            <param name="TEXTCOLOR" value="FF0000" />
            <param name="TEXT" value="Test :-)" />
            <param name="SPEED" value="250" />
            <param name="RANDOMCOLOR" value="1" />
            alt : <a href="class/FirstApplet.class">FirstApplet.class</a>
          </object>
        </p>
```

Il est grand temps de voir comment notre page web peut interagir avec notre applet...
Go !
Retour en haut

Interaction page web - applet

C'est à ce moment précis que vous aurez besoin des bases de Javascript.
Eh oui, la communication page web - applet se fait via un script Javascript !

En tout premier lieu, nous allons créer une méthode qui fait exactement la même chose que l'action lors du clic sur le bouton de notre applet, mais qui n'est jamais appelée dans notre applet...
C'est cette méthode que nous allons utiliser via Javascript :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JLabel;


public class FirstApplet extends JApplet {

 private JLabel label = new JLabel();
 private JButton bouton = new JButton("Cliquez");
 private int count = 0;

 /**
* Méthode d'initialisation de l'applet
* C'est cette méthode qui fait office de constructeur
*/
 public void init(){
  this.setSize(300, 80);

  System.out.println("Paramètre passé via la balise <param> : " +
this.getParameter("message"));
  //On centre le texte du JLabel et on écrit en bleu...
  label.setHorizontalAlignment(JLabel.CENTER);
  //C'est plus zoli.
  label.setForeground(Color.blue);

  //Allez, une classe anonyme... Just for the fun ;)
  this.bouton.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent arg0) {
    label.setText("Vous avez cliqué " + (++count) + " fois sur le
bouton");
   }
  });

  //On ajoute nos composants
  this.getContentPane().add(bouton, BorderLayout.SOUTH);
  this.getContentPane().add(label, BorderLayout.NORTH);
  //Et le tour est joué !
 }

 /**
* Méthode utilisée par Javascript pour mettre à jour
* Celle-ci doit être public afin que Javascript puisse y avoir accè
s
* Le JLabel
*/
 public void doJavascript(){
  System.out.println("Méthode appelée par javascript ! ");
  label.setText("Vous avez cliqué " + (++count) + " fois sur le
bouton");
 }

}
```

Rien de bien sorcier pour l'instant... Maintenant, nous allons voir comment appeler une méthode d'un applet Java dans un script Javascript. Pour ce faire, nous allons rajouter un simple bouton HTML sur notre applet précédent avec, lors du clic sur le bouton, l'appel à la fonction de l'applet.

Voici le code source HTML de cette page :

Code : HTML

```
<html>

 <body style="margin:auto;">

  <div style="width:100%;text-align:center;background-
color:#aef15b;"> </div>
  <div style="width:100%;text-align:center;background-
color:#aef15b;">
   <input type="button" value="Lancer la méthode de l'applet"
onClick="document.applets['first'].doJavascript();"/>
  </div>
  <div style="width:100%;text-align:center;background-
color:#aef15b;"> </div>


  <div style="width:320px;margin:auto;margin-top:20px;border:5px
solid black">
   <applet codebase="class/" code="FirstApplet.class" height="80"
width="300" hspace="10" vspace="10" name="first" id="firstApplet">
    <param name="message" value="Message pour les ZérOs">
   </applet>
  </div>

 </body>
</html>
```

Vous pouvez voir le résultat : ici.

Voyons un peu comment cela fonctionne.

Votre navigateur charge l'applet.
Vous avez spécifié que lorsque nous cliquerons sur le bouton (événement onClick) nous exécuterons un morceau de code Javascript : `document.applets['first'].doJavascript();` .
À l'appel de cette instruction, le script se positionne sur l'applet portant le nom **`first`** (notez que vous pouvez aussi utiliser un index, commençant par 0, pour utiliser des applets via Javascript : ici, `document.applets[0].doJavascript()` est équivalent) sur notre document et appelle la méthode **`doJavascript()`** déclarée dans cette applet. Celle-ci met à jour le **`JLabel`** à l'intérieur de l'applet, et le tour est joué !

Ici, la méthode que nous appelons ne retourne aucune valeur, mais vous pouvez utiliser une méthode retournant un type de donnée et l'affecter à une variable Javascript !

Nous allons aborder maintenant un point intéressant, la communication applet - page web !
Retour en haut

Interaction applet - page web

Dans ce cas, la communication se fait dans l'autre sens, c'est-à-dire que c'est votre applet qui va mettre à jour votre page web ! Ceci se fait toujours grâce à du Javascript, sauf que maintenant, c'est notre applet qui va invoquer une méthode Javascript...

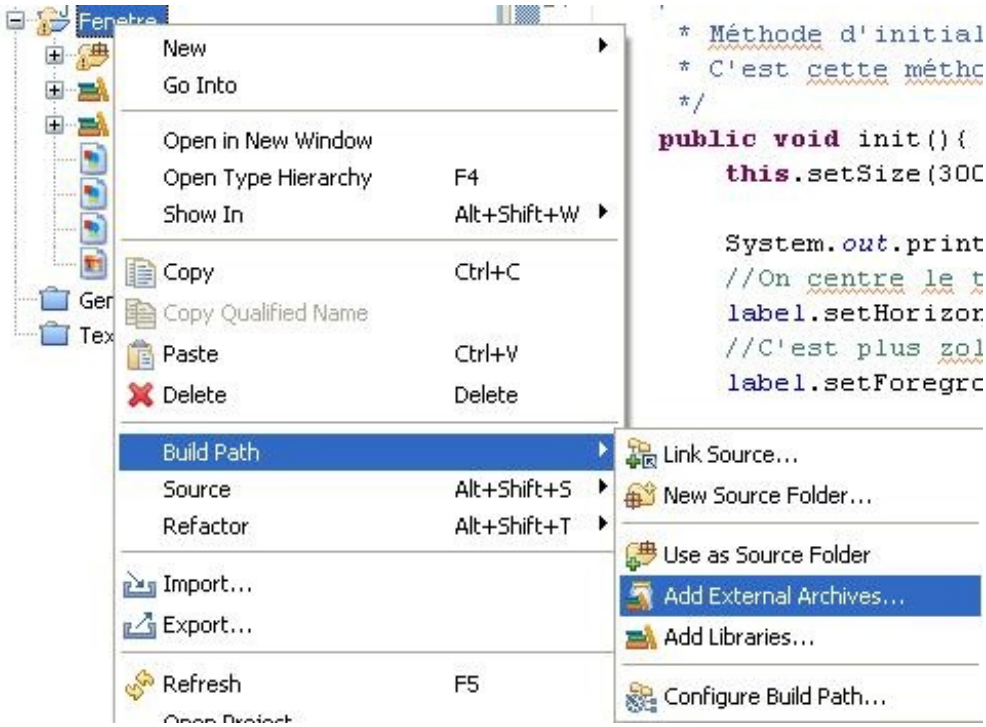
Pour réussir ce tour de force, nous allons avoir besoin d'un objet particulier, un `JSObject`, non présent d'office dans les ressources disponibles Java. Nous allons donc devoir utiliser une bibliothèque externe, l'ajouter à notre projet, recompiler le programme et ajouter cette nouvelle ressource dans la déclaration de notre applet dans la page web.

Où peut-on trouver cet objet, alors ?

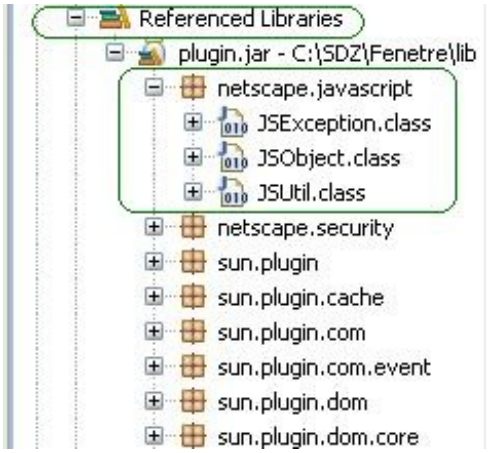
Il est dans votre JRE, et là, je veux dire dans le dossier **/lib** de votre JRE.
L'objet en question se trouve dans l'archive plugin.jar qui se trouve, je le répète, dans le dossier **/lib** de votre JRE (ou JDK).
Copiez cette archive dans le dossier contenant votre projet et mettez-le dans un dossier **/lib** que vous créerez. Vous devez maintenant avoir trois dossiers dans votre projet :

- **/src** : contient les fichiers sources de votre applet ;
- **/bin** : contient les fichiers .class de votre applet ;
- **/lib** : contient l'archive **plugin.jar**.

Maintenant, nous allons dire à Eclipse qu'il peut utiliser la nouvelle ressource afin que nous puissions utiliser les classes qu'elle contient.
Pour ce faire, faites un clic droit sur votre projet, puis dans "**Build Path**" choisissez "**Add External Archives**" comme ceci :



Allez maintenant dans le dossier **/lib** de votre projet et double cliquez sur **plugin.jar**. Vous devez voir que, maintenant, la ressource externe fait partie de votre projet : vous pouvez utiliser les classes que contient l'archive **plugin.jar**, et même voir son contenu :



L'objet tant convoité se trouve dans le package **`netscape.javascript`**. Oui, vous ne rêvez pas et il s'agit bien du même netscape que vous connaissez : le navigateur web ! Ce sont eux qui ont développé ces classes...

Bon, nous sommes parés à utiliser l'objet **`JSObject`** mais avant, nous allons mettre à jour notre page web afin que notre applet puisse écrire quelque part...

Nous allons donc retirer notre bouton HTML pour le remplacer par un `div` vide. C'est dans ce dernier que notre applet va écrire via du Javascript.
Il nous faut aussi une méthode Javascript pour écrire dans notre `div` : rien de difficile... Alors, qu'attendons-nous ?

Voici le code de notre page web :

Code : HTML

```
<html>

 <body style="margin:auto;">

  <script type="text/javascript">
   function affiche(str){
    document.getElementById('label').innerHTML = str;
   }
  </script>

  <div style="width:100%;text-align:center;background-
color:#aef15b;"> </div>
  <div style="width:100%;text-align:center;background-
color:#aef15b;color:white;font-weight:bold;font-size:1.2em;"
id="label"> </div>
  <div style="width:100%;text-align:center;background-
color:#aef15b;"> </div>


  <div style="width:320px;margin:auto;margin-top:20px;border:5px
solid black">
   <applet  mayscript="true" codebase="class3/"
archive="lib/plugin.jar" code="FirstApplet.class" height="80"
width="300" hspace="10" vspace="10" name="first">
    <param name="message" value="Message pour les ZérOs">
   </applet>
  </div>

 </body>
</html>
```

N'oubliez surtout pas l'attribut **`mayscript`** dans votre applet : sans celui-ci, votre applet ne sera pas habilitée à utiliser l'objet **`JSObject`** !

Et voici le code de votre applet :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JLabel;

import netscape.javascript.JSException;
import netscape.javascript.JSObject;


public class FirstApplet extends JApplet {

 private JLabel label = new JLabel();
 private JButton bouton = new JButton("Cliquez");
 private int count = 0;
 private JSObject jso;
 /**
* Méthode d'initialisation de l'applet
* C'est cette méthode qui fait office de constructeur
*/
 public void init(){
  setSize(300, 80);

  //On initialise l'objet

  try {
   jso = JSObject.getWindow(this);
  } catch (JSException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
  }

  System.out.println("Paramètre passé via la balise <param> : " +
this.getParameter("message"));
  //On centre le texte du JLabel et on écrit en bleu...
  label.setHorizontalAlignment(JLabel.CENTER);
  //C'est plus zoli.
  label.setForeground(Color.blue);

  //allez une classe anonyme... Just for the fun ;)
  this.bouton.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent arg0) {
    String str = "Vous avez cliqué " + (++count) + " fois sur le
bouton";
    label.setText(str);

    //On appelle la fonction Javascript
    //ceci peut lever une exception
    try {
     //On appelle la méthode affiche en lui passant en paramètre
     //un tableau
                        jso.call("affiche", new String[]
{String.valueOf(str)});
                }
                catch (Exception ex) {
                        ex.printStackTrace();
                }
   }
  });
```

```
        //On ajoute nos composants
        this.getContentPane().add(bouton, BorderLayout.SOUTH);
        this.getContentPane().add(label, BorderLayout.NORTH);
        //Et le tour est joué !
      }

    }
```

Pour plus de dynamisme, vous pouvez passer le nom de la méthode Javascript à utiliser avec un paramètre de la balise applet... Comme ça, vous n'êtes pas obligés de changer le code source de l'applet si le nom de votre méthode Javascript change !

Et le résultat est exactement celui escompté.

Dans ce code, il n'y a rien de compliqué...
L'objet s'utilise facilement, il suffit de ne pas oublier de gérer les exceptions et c'est bon.

Avant de terminer ce chapitre, vous devez savoir que les applets n'ont pas le droit de tout faire sur une page web !

Retour en haut

Applets et sécurité

En fait, au même titre que Javascript, les applets Java n'ont pas le droit d'accéder à la machine du client. Pour faire simple, ils sont confinés dans le navigateur web.
Et heureusement ! Vous imaginez un peu toutes les dérives si ce genre de script ou de programme pouvait naturellement avoir accès à votre PC ? Là, on pourrait devenir parano et il vaudrait mieux...

- Est-ce que ce script n'est pas en train d'effacer un fichier de configuration ?
- Il me semble que cette applet est en train d'accéder à un répertoire sensible...
- ...

Bref, vous imaginez...

Cependant, il se peut que quelquefois, pour quelques rares cas, votre applet doive accéder à des ressources de votre PC.

Exemple

Dans la boîte dans laquelle je suis actuellement, nous sommes en train de développer une application, format client léger (web), afin de gérer tous les processus industriels de la société, dont la pesée de certains articles avec scan des documents en même temps.
Nous avons donc fait un applet qui s'occupe de faire tout ça mais pour communiquer avec les ports COM et le scanner, nous avons dû **signer notre applet**.

Eh... Quoi ?

Nous avons signé notre applet, c'est-à-dire que nous avons créé un certificat que nous avons attribué à notre applet et que l'utilisateur DOIT soit accepter, soit refuser au chargement de la page : ce certificat stipule que l'applet peut accéder à des ressources de sa machine, et lui demande s'il veut lui faire confiance .

Il n'est pas très pertinent de parler de la façon de signer une applet : au pire, si vous avez vraiment besoin de ça, Google est votre ami...

Vous savez tout de même que les applets n'ont pas tous les droits sur une page web, au même titre que Javascript.

Vous avez vu pas mal de choses, mine de rien, dans ce chapitre.
On va faire un tour sur le topo habituel, suivi d'un petit QCM...
Retour en haut

Ce que vous devez retenir

- Les applets peuvent dériver de `java.awt.Applet` ou de `javax.swing.JApplet`.
- Les applets n'ont pas de constructeur mais une méthode `init()`.
- En gros, les applets se comportent comme des applications fenêtrées.
- Vous pouvez communiquer de votre page web vers votre applet grâce à l'instruction Javascript : `document.applets`.
- La communication inverse se fait via l'objet `JSObject` qui se trouve dans l'archive **plugin.jar** dans le dossier **/lib** de votre JRE (ou JDK).
- Par défaut, **les applets n'ont pas accès aux ressources de la machine client**.
- Pour accéder à la machine du client, **vous devrez signer votre applet !**

Retour en haut

Maintenant, je pense que vous ne pourrez plus mélanger :

- Javascript et Java ;
- les applets et J2EE.

Sinon, vous êtes IMPARDONNABLES !

Après cette petite dérive, je pense que nous pouvons continuer notre tour d'horizon de l'API **`swing`** !
En avant pour **les boîtes de dialogues** !

Retour en haut

Les boîtes de dialogue

Les boîtes de dialogue, c'est sûr, vous connaissez !
Cependant, afin d'être sûr que nous parlons de la même chose, voici un petit descriptif.

Une boîte de dialogue est une mini-fenêtre pouvant servir à plusieurs choses :

- afficher une information (message d'erreur, warning...) ;
- demander une validation, une réfutation ou une annulation ;
- demander à l'utilisateur de saisir une information dont le système a besoin ;
- ...

Vous pouvez voir que ces dernières peuvent servir à beaucoup de choses.
Il faut toutefois les utiliser avec parcimonie : il est, au final, assez pénible qu'une application ouvre une boîte de dialogue à chaque notification, car toute boîte ouverte doit être fermée !

Pour ce point je vous laisse seuls juges de leur utilisation... Et si nous commencions ?

Les boîtes d'information

L'objet que nous allons utiliser tout au long de ce chapitre n'est autre que le **JOptionPane**.
Objet assez complexe au premier abord, mais fort pratique.

Voici à quoi ressemblent des boîtes de dialogues "*informatives*" :

Ces boîtes n'ont pas vocation à participer à de quelconques traitements, elles affichent juste un message pour l'utilisateur.

Voici le code utilisé pour obtenir ces boîtes :

Code : Java

```
//Déclaration des objets
 JOptionPane jop1, jop2, jop3;

//Boîte du message d'information
//------------------------------
jop1 = new JOptionPane();
jop1.showMessageDialog(null, "Message informatif", "Information",
JOptionPane.INFORMATION_MESSAGE);

//Boîte du message préventif
//------------------------------
jop2 = new JOptionPane();
jop2.showMessageDialog(null, "Message préventif", "Attention",
JOptionPane.WARNING_MESSAGE);

//Boîte du message d'erreur
//------------------------------
jop3 = new JOptionPane();
jop3.showMessageDialog(null, "Message d'erreur", "Erreur",
JOptionPane.ERROR_MESSAGE);
```

Ces trois boîtes ne s'affichent pas en même temps...

Pourquoi ça ?

Tout simplement parce qu'en Java, mais aussi dans les autres langages, les boîtes de dialogue sont dites **modales**.
Ceci signifie que lorsque qu'une boîte fait son apparition, celle-ci bloque toute interaction possible avec un autre composant que ceux présents sur la boîte. Ceci, tant que l'utilisateur n'a pas mis fin au dialogue !

Même en créant un thread différent par boîte de dialogue, vous ne pourrez pas toutes les voir !
Les 3 boîtes seront créées, mais les boîtes s'affichant au centre de la fenêtre, vous ne pourrez en voir que deux sur trois, en

déplaçant la première.

Comme le montre le résultat de ce code, vous ne pourrez pas déplacer la deuxième boîte à cause de la modalité des boîtes :

Code : Java

```
import javax.swing.JOptionPane;

public class Test {

 public static void main(String[] args) {

  Thread t = new Thread(new Runnable(){
   public void run(){
    JOptionPane jop1 = new JOptionPane();
    jop1.showMessageDialog(null, "Message informatif",
"Information", JOptionPane.INFORMATION_MESSAGE);
   }
  });

  Thread t2 = new Thread(new Runnable(){
   public void run(){
    JOptionPane jop2 = new JOptionPane();
    jop2.showMessageDialog(null, "Message préventif", "Attention",
JOptionPane.WARNING_MESSAGE);
   }
  });

  Thread t3 = new Thread(new Runnable(){
   public void run(){
    JOptionPane jop3 = new JOptionPane();
    jop3.showMessageDialog(null, "Message d'erreur", "Erreur",
JOptionPane.ERROR_MESSAGE);
   }
  });

  t.start();
  t2.start();
  t3.start();
 }
}
```

Ce qui nous donnerait (tout dépend de l'ordonnanceur, vu qu'il y a trois thread...) :

Maintenant, voyons de plus près la façon de construire un tel objet. Ici nous avons utilisé la méthode : `showMessageDialog(Component parentComponent, String message, String title, int`

`messageType);` .

- **`Component`parentComponent** : correspond au composant parent ; ici, il n'y en a aucun, nous mettons donc **`null`**.
- **`String`message** : permet de renseigner le message à afficher dans la boîte de dialogue.
- **`String`title** : permet de donner un titre à notre objet.
- **`int`messageType** : permet de savoir s'il s'agit d'un message d'information, de prévention ou d'erreur. Vous avez sans doute remarqué que, mis à part le texte et le titre, seul ce champ change entre les trois objets !

Il existe deux autres méthodes `showMessageDialog()` pour cet objet : une qui prend deux paramètres en moins (le titre et le type de message), et une qui prend un paramètre en plus (l'icône à utiliser).

Je pense qu'il est inutile de détailler la méthode avec les paramètres en moins, mais voici des exemples de boîtes avec des icônes définies par nos soins...

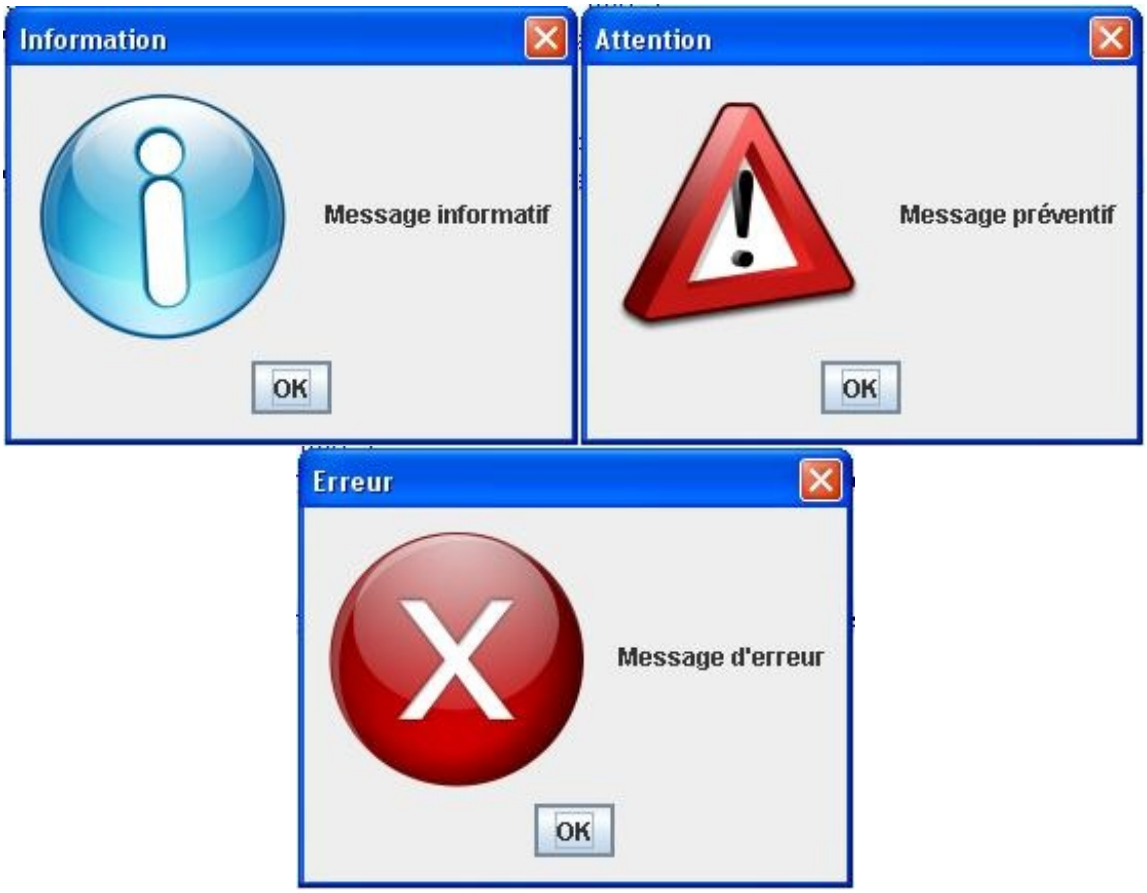
Code : Java

```
import javax.swing.ImageIcon;
import javax.swing.JOptionPane;


public class Test {

 public static void main(String[] args) {

  JOptionPane jop1, jop2, jop3;

  jop1 = new JOptionPane();
  ImageIcon img = new ImageIcon("images/information.png");
  jop1.showMessageDialog(null, "Message informatif", "Information",
JOptionPane.INFORMATION_MESSAGE, img);

  jop2 = new JOptionPane();
  img = new ImageIcon("images/warning.png");
  jop2.showMessageDialog(null, "Message préventif", "Attention",
JOptionPane.WARNING_MESSAGE, img);

  jop3 = new JOptionPane();
  img = new ImageIcon("images/erreur.png");
  jop3.showMessageDialog(null, "Message d'erreur", "Erreur",
JOptionPane.ERROR_MESSAGE, img);

 }
}
```

Les images ont été trouvées sur Google puis rangées dans un dossier "images" à la racine du projet Eclipse ! Je vous invite à télécharger vos propres images et de faire vos propres tests...

Vous constaterez aussi l'utilisation de l'objet **`ImageIcon`** qui va lire le fichier image à l'emplacement spécifié dans son constructeur.

Voici le résultat obtenu :



Ce type de boîtes est très utile pour notifier à l'utilisateur qu'un traitement s'est terminé ou qu'une erreur est survenue...
L'exemple le plus simple qui me vient en tête serait une division par zéro :

Code : Java

```
import javax.swing.JOptionPane;

public class Test {

 public static void main(String[] args) {

  int i = 100, j = 0;

  try{
   System.out.println("Résultat = " + (i/j));
  }catch(ArithmeticException ex){
   JOptionPane  jop3 = new JOptionPane();
   jop3.showMessageDialog(null, "Division par zéro détecté !",
"Erreur fatale ! ! ", JOptionPane.ERROR_MESSAGE);
  }
 }
}
```

Les types de boîtes

Voici les types de boîtes que vous pouvez utiliser (valable pour tout ce qui suit), triés par ordre alphabétique s'il vous plaît... :

- JOptionPane.ERROR_MESSAGE
- JOptionPane.INFORMATION_MESSAGE
- JOptionPane.PLAIN_MESSAGE
- JOptionPane.QUESTION_MESSAGE

- `JOptionPane.WARNING_MESSAGE`

Je pense que vous devez mieux voir l'utilité de telles boîtes de dialogues...
Nous allons donc poursuivre avec les boîtes de confirmation.
Retour en haut

Les boîtes de confirmation

Comme leur nom l'indique, ces dernières permettent de valider, de réfuter ou d'annuler une décision.
Nous utiliserons toujours l'objet **JOptionPane** mais, cette fois, avec la méthode `showConfirmDialog()` . Cette méthode retourne un entier qui correspond à l'option que vous aurez choisie dans cette boîte :

- Yes ;
- No ;
- Cancel.

Comme exemple, nous pouvons prendre notre animation comme nous l'avons laissée la dernière fois.
Nous pourrions utiliser une boîte de confirmation lorsque nous cliquerons sur l'un des boutons contrôlant l'animation (`Go` ou `Stop`).

Pour ceux qui n'auraient pas conservé leur projet, voici la classe **Panneau** :

Secret (cliquez pour afficher)

Code : Java

```
import java.awt.Color;
import java.awt.Font;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;

import javax.swing.JPanel;

public class Panneau extends JPanel {

        private int posX = -50;
    private int posY = -50;
    private int drawSize = 50;
    //boolean pour le mode morphing et pour savoir si la taille
doit réduire
    private boolean morph = false, reduce = false;;
    private String forme = "ROND";
    //Le compteur de rafraîchissements
    private int increment = 0;

    public void paintComponent(Graphics g){
            g.setColor(Color.white);
            g.fillRect(0, 0, this.getWidth(), this.getHeight());
            g.setColor(Color.red);
            //Si le mode morphing est activé, on peint le
morphing
            if(this.morph)
                drawMorph(g);
            //sinon, mode normal
            else
                draw(g);
    }

    private void draw(Graphics g){

        if(this.forme.equals("ROND")){
                g.fillOval(posX, posY, 50, 50);
        }
        if(this.forme.equals("CARRE")){
                g.fillRect(posX, posY, 50, 50);
        }
        if(this.forme.equals("TRIANGLE")){

                int s1X = posX + 50/2;
                int s1Y = posY;
                int s2X = posX + 50;
                int s2Y = posY + 50;
                int s3X = posX;
```

```
                int s3Y = posY + 50;

                int[] ptsX = {s1X, s2X, s3X};
                int[] ptsY = {s1Y, s2Y, s3Y};

                g.fillPolygon(ptsX, ptsY, 3);
            }
            if(this.forme.equals("ETOILE")){

                int s1X = posX + 50/2;
                int s1Y = posY;
                int s2X = posX + 50;
                int s2Y = posY + 50;
                g.drawLine(s1X, s1Y, s2X, s2Y);

                int s3X = posX;
                int s3Y = posY + 50/3;
                g.drawLine(s2X, s2Y, s3X, s3Y);

                int s4X = posX + 50;
                int s4Y = posY + 50/3;
                g.drawLine(s3X, s3Y, s4X, s4Y);

                int s5X = posX;
                int s5Y = posY + 50;
                g.drawLine(s4X, s4Y, s5X, s5Y);
                g.drawLine(s5X, s5Y, s1X, s1Y);
            }

        }

        /**
     * Méthode qui peint le morphing
     * @param g
     */
        private void drawMorph(Graphics g){
            //On incrémente le tour
            increment++;
            //On regarde si on doit réduire ou non
            if(drawSize >= 50)reduce = true;
            if(drawSize <= 10)reduce = false;

            if(reduce)
                drawSize = drawSize - getUsedSize();
            else
                drawSize = drawSize + getUsedSize();

            if(this.forme.equals("ROND")){
                g.fillOval(posX, posY, drawSize, drawSize);
            }
            if(this.forme.equals("CARRE")){
                g.fillRect(posX, posY, drawSize, drawSize);
            }
            if(this.forme.equals("TRIANGLE")){

                int s1X = posX + drawSize/2;
                int s1Y = posY;
                int s2X = posX + drawSize;
                int s2Y = posY + drawSize;
                int s3X = posX;
                int s3Y = posY + drawSize;

                int[] ptsX = {s1X, s2X, s3X};
                int[] ptsY = {s1Y, s2Y, s3Y};

                g.fillPolygon(ptsX, ptsY, 3);
            }
            if(this.forme.equals("ETOILE")){

                int s1X = posX + drawSize/2;
```

```
                int s1Y = posY;
                int s2X = posX + drawSize;
                int s2Y = posY + drawSize;
                g.drawLine(s1X, s1Y, s2X, s2Y);

                int s3X = posX;
                int s3Y = posY + drawSize/3;
                g.drawLine(s2X, s2Y, s3X, s3Y);

                int s4X = posX + drawSize;
                int s4Y = posY + drawSize/3;
                g.drawLine(s3X, s3Y, s4X, s4Y);

                int s5X = posX;
                int s5Y = posY + drawSize;
                g.drawLine(s4X, s4Y, s5X, s5Y);
                g.drawLine(s5X, s5Y, s1X, s1Y);
            }


        }

        /**
    * Méthode qui retourne le nombre à retrancher (ou ajouter) pour
    le morphing
    * @return res
    */
        private int getUsedSize(){
            int res = 0;
            //Si le nombre de tours est de 10
            //On réinitialise l'incrément et on retourne 1
            if(increment / 10 == 1){
                    increment = 0;
                    res = 1;
            }
            return res;
        }

        public int getDrawSize(){
            return drawSize;
        }

        public boolean isMorph(){
            return morph;
        }

        public void setMorph(boolean bool){
            this.morph = bool;
            //On réinitialise la taille
            drawSize = 50;
        }

        public void setForme(String form){
            this.forme = form;
        }

        public int getPosX() {
                return posX;
        }

        public void setPosX(int posX) {
                this.posX = posX;
        }

        public int getPosY() {
                return posY;
        }

        public void setPosY(int posY) {
                this.posY = posY;
```

```
        }
    }
```

Voici le code de notre classe **Fenetre** :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;

public class Fenetre extends JFrame{

 private Panneau pan = new Panneau();
 private JButton bouton = new JButton("Go");
 private JButton bouton2 = new JButton("Stop");
    private JPanel container = new JPanel();
    private JLabel label = new JLabel("Choix de la forme");
    private int compteur = 0;
    private boolean animated = true;
    private boolean backX, backY;
    private int x,y ;
    private Thread t;
    private JComboBox combo = new JComboBox();

    private JCheckBox morph = new JCheckBox("Morphing");

    public Fenetre(){

            this.setTitle("Animation");
            this.setSize(300, 300);
            this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            this.setLocationRelativeTo(null);

            container.setBackground(Color.white);
            container.setLayout(new BorderLayout());
            container.add(pan, BorderLayout.CENTER);

            bouton.addActionListener(new BoutonListener());
            bouton2.addActionListener(new Bouton2Listener());

            bouton2.setEnabled(false);

            JPanel south = new JPanel();
            south.add(bouton);
            south.add(bouton2);
            container.add(south, BorderLayout.SOUTH);

            combo.addItem("ROND");
            combo.addItem("CARRE");
            combo.addItem("TRIANGLE");
            combo.addItem("ETOILE");
            combo.addActionListener(new FormeListener());

            morph.addActionListener(new MorphListener());

            JPanel top = new JPanel();
```

```
                top.add(label);
                top.add(combo);
                top.add(morph);

                container.add(top, BorderLayout.NORTH);
                this.setContentPane(container);
                this.setVisible(true);

    }


 private void go(){
  x = pan.getPosX();
  y = pan.getPosY();
        while(this.animated){
         //System.out.println("OK");
         //Si le mode morphing est activé, on utilise la taille
actuelle de la forme
       if(pan.isMorph())
       {
        if(x < 1)backX = false;
              if(x > pan.getWidth() - pan.getDrawSize())backX = true;
              if(y < 1)backY = false;
              if(y > pan.getHeight() - pan.getDrawSize())backY =
true;
       }
       //Sinon, comme d'habitude
       else
       {
           if(x < 1)backX = false;
              if(x > pan.getWidth()-50)backX = true;
              if(y < 1)backY = false;
              if(y > pan.getHeight()-50)backY = true;
       }

          if(!backX)pan.setPosX(++x);
             else pan.setPosX(--x);
             if(!backY) pan.setPosY(++y);
             else pan.setPosY(--y);
             pan.repaint();

             try {
                     Thread.sleep(3);
             } catch (InterruptedException e) {
                     // TODO Auto-generated catch block
                     e.printStackTrace();
             }
        }
 }

 public class BoutonListener implements ActionListener{

  public void actionPerformed(ActionEvent arg0) {

   JOptionPane jop = new JOptionPane();
   int option = jop.showConfirmDialog(null, "Voulez-vous lancer
l'animation ?", "Lancement de l'animation",
JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE);

   if(option == JOptionPane.OK_OPTION)
   {
    animated = true;
    t = new Thread(new PlayAnimation());
    t.start();
    bouton.setEnabled(false);
    bouton2.setEnabled(true);
   }
  }
 }
```

```
 class Bouton2Listener  implements ActionListener{

  public void actionPerformed(ActionEvent e) {

      JOptionPane jop = new JOptionPane();
   int option = jop.showConfirmDialog(null, "Voulez-vous arrêter
l'animation ?", "Arrêt de l'animation",
JOptionPane.YES_NO_CANCEL_OPTION, JOptionPane.QUESTION_MESSAGE);

   if(option != JOptionPane.NO_OPTION && option !=
JOptionPane.CANCEL_OPTION && option != JOptionPane.CLOSED_OPTION)
   {
    animated = false;
    bouton.setEnabled(true);
    bouton2.setEnabled(false);
   }
  }
 }

 class PlayAnimation implements Runnable{

  public void run() {
   go();
  }
 }


 class FormeListener implements ActionListener{

  public void actionPerformed(ActionEvent e) {
   pan.setForme(combo.getSelectedItem().toString());
  }
 }

 class MorphListener implements ActionListener{

  public void actionPerformed(ActionEvent e) {
   //Si la case est cochée, activation du mode morphing
   if(morph.isSelected())pan.setMorph(true);
   //Sinon rien !
   else pan.setMorph(false);
  }
 }
}
```

L'une des instructions intéressantes se trouve ici :

Code : Java

```
JOptionPane jop = new JOptionPane();
int option = jop.showConfirmDialog(null, "Voulez-vous lancer
l'animation ?", "Lancement de l'animation",
JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE);

if(option == JOptionPane.OK_OPTION)
{
 animated = true;
 t = new Thread(new PlayAnimation());
 t.start();
 bouton.setEnabled(false);
 bouton2.setEnabled(true);
}
```

Et l'autre, là :

Code : Java

```
JOptionPane jop = new JOptionPane();
int option = jop.showConfirmDialog(null, "Voulez-vous arrêter
l'animation ?", "Arrêt de l'animation",
JOptionPane.YES_NO_CANCEL_OPTION, JOptionPane.QUESTION_MESSAGE);

if(option != JOptionPane.NO_OPTION && option !=
JOptionPane.CANCEL_OPTION && option != JOptionPane.CLOSED_OPTION)
{
 animated = false;
 bouton.setEnabled(true);
 bouton2.setEnabled(false);
}
```

Voyons ce qu'il se passe ici :

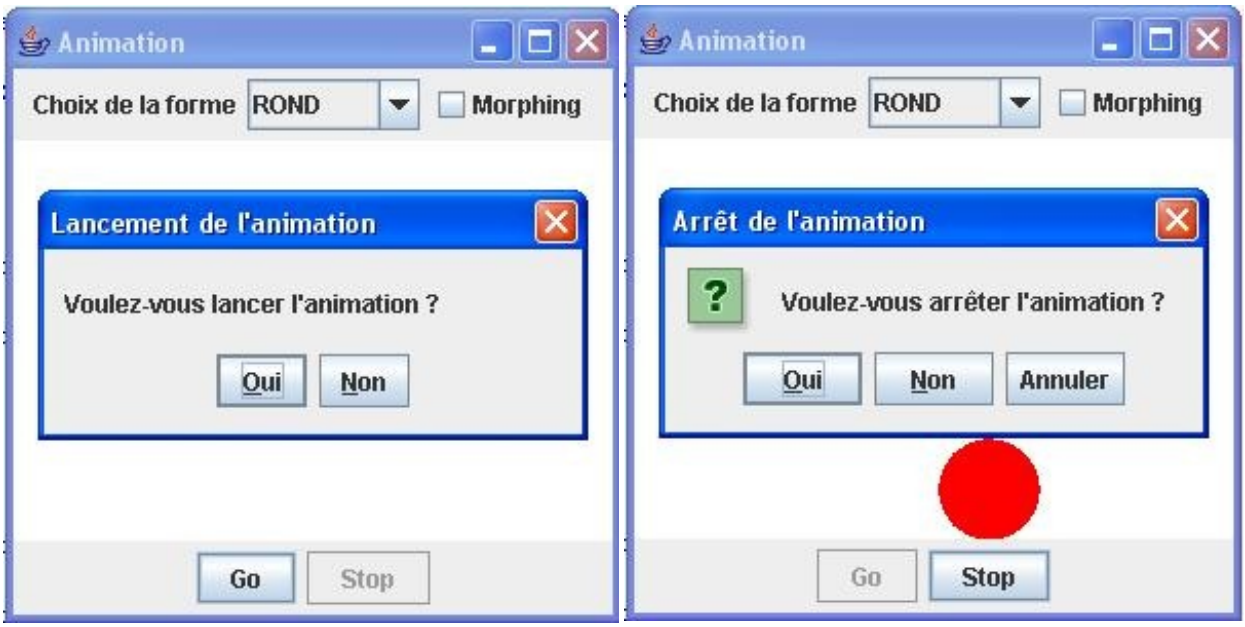
- nous initialisons notre objet **JOptionPane** : rien d'étonnant ;
- par contre, au lieu d'afficher la boîte, nous affectons le résultat que renvoie la méthode `showConfirmDialog()` dans une variable de type **int** ;
- nous nous servons de cette variable afin de savoir quel bouton a été cliqué (oui ou non).

En fait, lorsque vous cliquez sur l'un des deux boutons présents sur cette boîte, vous pouvez affecter ici deux valeurs de type **int** :

- la valeur correspondant à l'entier `JOptionPane.OK_OPTION` vaut 0 (`JOptionPane.YES_OPTION` à la même valeur) ;
- la valeur correspondant à l'entier `JOptionPane.NO_OPTION` vaut 1 ;
- la valeur correspondant à l'entier `JOptionPane.CANCEL_OPTION` pour la boîte apparaissant lors du clic sur "**Stop**" vaut 2 ;
- la valeur correspondant à l'entier `JOptionPane.CLOSED_OPTION` pour la même boîte que ci-dessus, vaut -1.

En comparant la valeur de notre entier avec l'une des deux options, nous pouvons en déduire quel bouton a été cliqué et donc agir en conséquence !

Voici deux screenshots du résultat obtenu :



Vous commencez à maîtriser les `JOptionPane`, on dirait...
Si on continuait ?...

Retour en haut

Les boîtes de saisie

Je suis sûr que vous avez deviné ce à quoi ces boîtes peuvent servir...
Oui, tout à fait, nous allons pouvoir faire des saisies dans ces boîtes ! Et même encore mieux... Nous pourrons même avoir une boîte de dialogue qui propose des choix dans une liste déroulante.

Je sens que vous êtes pressés de commencer. Alors allons-y.
Vous savez déjà que nous allons utiliser l'objet **JOptionPane**, et les plus curieux d'entre vous ont sûrement dû jeter un oeil aux autres méthodes proposés par cet objet...

Ici, nous allons utiliser la méthode `showInputDialog(Component parent, String message, String title, int messageType)` .

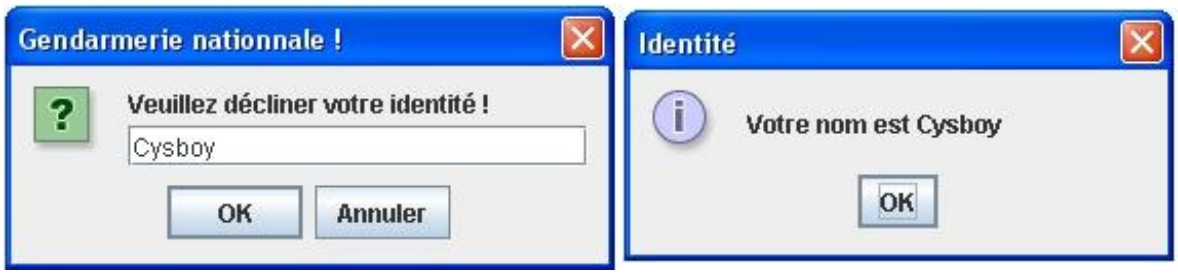
Voici un code mettant en oeuvre cette méthode :

Code : Java

```
import javax.swing.JOptionPane;

public class Test {

 public static void main(String[] args) {
  JOptionPane jop = new JOptionPane(), jop2 = new JOptionPane();
  String nom = jop.showInputDialog(null, "Veuillez décliner votre
identité !", "Gendarmerie nationale !",
JOptionPane.QUESTION_MESSAGE);
  jop2.showMessageDialog(null, "Votre nom est " + nom, "Identité",
JOptionPane.INFORMATION_MESSAGE);
 }
}
```

Vous pouvez constater que cette méthode retourne une chaîne de caractères !

Voici le résultat :



Rien d'extraordinaire...

Maintenant, voyons comment on incorpore une liste dans une boîte de ce genre... Vous allez voir, c'est simplissime !

Code : Java

```
import javax.swing.JOptionPane;

public class Test {

 public static void main(String[] args) {
  String[] sexe = {"masculin", "féminin", "indéterminé"};
  JOptionPane jop = new JOptionPane(), jop2 = new JOptionPane();
```

```
        String nom = (String)jop.showInputDialog(null,
                "Veuillez décliner votre sexe !",
                "Gendarmerie nationale !",
                JOptionPane.QUESTION_MESSAGE,
                null,
                sexe,
                sexe[2]);
        jop2.showMessageDialog(null, "Votre sexe est " + nom, "Etat
civil", JOptionPane.INFORMATION_MESSAGE);
    }
}
```

Ce code nous donne :

Nous avons utilisé la méthode avec des paramètres en plus, les voici :

- les quatre premiers, vous connaissez maintenant ;
- le deuxième `null` correspond à l'icône que vous souhaitez passer ;
- ensuite, vous devez passer un tableau de `String` afin de remplir la combo dans la boîte ;
- le dernier paramètre correspond à la valeur à sélectionner par défaut.

Cette méthode retourne aussi un objet, mais de type `Object`, comme si vous récupériez la valeur directement dans la combo !

Donc pensez à faire un cast.

Voici maintenant une variante de ce que vous venez de voir : nous allons utiliser ici la méthode `showOptionDialog()` . Celle-ci fonctionne à peu près comme la méthode précédente sauf qu'elle prend un paramètre de plus et que le type de retour n'est pas un objet, mais un entier !

Ce type de boîte propose, au lieu d'une combo, un choix de boutons correspondant aux éléments passés en paramètres (tableau de String) ; elle prend aussi une valeur par défaut mais retourne l'indice de l'élément dans la liste, et non l'élément lui-même.

Je pense que vous vous y connaissez assez pour comprendre le code suivant :

Code : Java

```
import javax.swing.JOptionPane;

public class Test {

 public static void main(String[] args) {
  String[] sexe = {"masculin", "féminin", "indéterminé"};
  JOptionPane jop = new JOptionPane(), jop2 = new JOptionPane();
  int rang = jop.showOptionDialog(null,
          "Veuillez décliner votre sexe !",
          "Gendarmerie nationale !",
          JOptionPane.YES_NO_CANCEL_OPTION,
          JOptionPane.QUESTION_MESSAGE,
          null,
```

```
                sexe,
                sexe[2]);
        jop2.showMessageDialog(null, "Votre sexe est " + sexe[rang], "Etat
civil", JOptionPane.INFORMATION_MESSAGE);
    }
}
```

Ce qui nous donne :

Voilà, vous en avez terminé avec les boîtes de saisie...
Cependant, vous devez vous demander si vous ne pouvez pas ajouter des composants sur ces boîtes.
C'est vrai, vous pourriez avoir besoin de plus de renseignements, on ne sait jamais... Je vous propose donc de vous montrer comment créer vos propres boîtes de dialogue !

En avant, mes ZérOs ! Hissez haut !

Retour en haut

Des boîtes de dialogue personnalisées

Je me doute bien que vous devez être impatients de faire vos propres boîtes de dialogue...
Il est vrai que dans certains cas, vous en aurez besoin, donc allons-y gaiement !

Je vais maintenant vous révéler un secret bien gardé : les boîtes de dialogue héritent de la classe **JDialog**.
Vous avez donc deviné que nous allons créer une classe dérivée de cette dernière.

Commençons par faire un nouveau projet.
Créez une nouvelle classe dans Eclipse, appelons-la **ZDialog** héritée de la classe citée ci-dessus, et mettez-y le code suivant :

Code : Java

```
import javax.swing.JDialog;
import javax.swing.JFrame;


public class ZDialog extends JDialog {

 public ZDialog(JFrame parent, String title, boolean modal){
  //On appelle le construteur de JDialog correspondant
  super(parent, title, modal);
  //On spécifie une taille
  this.setSize(200, 80);
  //La position
  this.setLocationRelativeTo(null);
  //La boîte ne devra pas être redimensionnable
  this.setResizable(false);
  //Enfin on l'affiche
  this.setVisible(true);
  //Tout ceci ressemble à ce que nous faisons depuis le début avec
notre JFrame...
 }
}
```

Maintenant, faisons une classe qui va tester notre **ZDialog** :

Code : Java

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;


public class Fenetre extends JFrame {

 private JButton bouton = new JButton("Appel à la ZDialog");

 public Fenetre(){

  this.setTitle("Ma JFrame");
  this.setSize(300, 100);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setLocationRelativeTo(null);

  this.getContentPane().setLayout(new FlowLayout());
  this.getContentPane().add(bouton);
  bouton.addActionListener(new ActionListener(){

   public void actionPerformed(ActionEvent arg0) {
    ZDialog zd = new ZDialog(null, "Coucou les ZérOs", true);
   }
```

```
        });

        this.setVisible(true);

    }

    public static void main(String[] main){
        Fenetre fen = new Fenetre();
    }

}
```

Voilà le résultat ; bon, c'est un début :

Je pense que vous avez deviné à quoi servaient les paramètres du constructeur... Mais nous allons tout de même les expliciter :

- `JFrame Parent` correspond à l'objet parent ;
- `String title` correspond au titre de notre boîte ;
- `boolean modal` correspond à la modalité. **`true`** : boîte modale, **`false`** : boîte non modale.

Rien de compliqué... Il est donc temps de mettre des composants sur notre objet...
Par contre, vous conviendrez que lorsque nous faisons un tel composant, nous voulons quelque chose de plus qu'une réponse à une question ouverte (oui / non), une chaîne de caractères ou encore un choix dans une liste... Nous en voulons encore plus ! Plusieurs saisies, avec plusieurs listes : **en même temps !**

Mais vous avez vu que nous devrons récupérer les informations choisies dans certains cas, mais pas tous : nous allons donc devoir déterminer les différents cas ainsi que les choses à faire.

Nous partons du fait que notre boîte aura un bouton "***OK***" et "***Annuler***" : dans le cas où l'utilisateur clique sur "***OK***", on récupérera les informations, et si l'utilisateur clique sur "***Annuler***", non.
Tout ceci en tenant compte de la modalité de notre boîte...

D'ailleurs, comment va-t-on faire pour dire à notre boîte de mettre fin au dialogue ?

Tout simplement en utilisant la méthode `setVisible(false);`, cette instruction met fin au dialogue !
Ceci signifie aussi que le dialogue commence au moment où l'instruction `setVisible(true);` est exécutée. De ce fait, nous allons sortir cette instruction du constructeur de notre objet et nous allons la mettre dans une méthode à part.

Code : Java

```
    //Cas où notre ZDialog renverra le contenu
    //D'un JTextField nommé jtf
    public String showZDialog(){
        this.setVisible(true);
        //Le dialogue prend fin
        //SI on a cliqué sur OK, on envoie, sinon on envoie chaîne vide
    !
        return jtf.getText();

    }
```

Il nous reste un dernier point à gérer...

Comment récupérer les informations saisies dans notre boîte depuis notre fenêtre, vu que nous voulons plusieurs informations ?

C'est vrai qu'on ne peut retourner qu'une valeur à la fois...
Mais il peut y avoir plusieurs réponses à cette question.

- Dans le cas où nous n'avons qu'un composant, nous pouvons adapter la méthode **`showZDialog()`** au type de retour du composant utilisé.
- Dans notre cas, nous voulons plusieurs composants, donc plusieurs valeurs, vous pouvez :
  - retourner une collection de valeurs (**`ArrayList`**...) ;
  - faire des accesseurs dans votre **`ZDialog`** ;
  - créer un objet dont le rôle est de collecter les informations de votre boîte et de retourner cet objet...
  - ...

Nous allons opter pour un objet qui collectera les informations et nous retournerons cet objet à la fin de la méthode **`showZDialog()`**.
Avant de nous lancer dans la création de cet objet, nous devons savoir ce que nous allons mettre dans notre boîte...

Nous allons faire une boîte permettant de spécifier les caractéristiques d'un personnage de jeu vidéo :

- son nom : un champ de saisie ;
- son sexe : une combo ;
- sa taille : un champ de saisie ;
- sa couleur de cheveux : une combo ;
- sa tranche d'âge : des radios.

Pour le placement des composants, l'objet **`JDialog`** est exactement identique à un objet **`JFrame`** (**`BorderLayout`** par défaut, ajout de composant au conteneur...).

Nous pouvons donc faire notre objet contenant les informations de notre boîte de dialogue, je l'ai appelé **`ZDialogInfo`** :

Code : Java

```
public class ZDialogInfo {

 private String nom, sexe, age, cheveux, taille;

 public ZDialogInfo(){}
 public ZDialogInfo(String nom, String sexe, String age,
      String cheveux, String taille){
  this.nom = nom;
  this.sexe = sexe;
  this.age = age;
  this.cheveux = cheveux;
  this.taille = taille;
 }

 //-------------------------------------

 public String getNom() {
  return nom;
 }

 public void setNom(String nom) {
  this.nom = nom;
 }
```

```
        //-------------------------------------

        public String getSexe() {
         return sexe;
        }

        public void setSexe(String sexe) {
         this.sexe = sexe;
        }

        //-------------------------------------

        public String getAge() {
         return age;
        }

        public void setAge(String age) {
         this.age = age;
        }

        //-------------------------------------
        public String getCheveux() {
         return cheveux;
        }

        public void setCheveux(String cheveux) {
         this.cheveux = cheveux;
        }

        //-------------------------------------
        public String getTaille() {
         return taille;
        }

        public void setTaille(String taille) {
         this.taille = taille;
        }

        //-------------------------------------

        public String toString(){
         String str;
         if(this.nom != null && this.sexe != null &&
           this.taille != null && this.age != null &&
           this.cheveux != null){
          str = "Description de l'objet InfoZDialog";
          str += "Nom : " + this.nom + "\n";
          str += "Sexe : " + this.sexe + "\n";
          str += "Age : " + this.age + "\n";
          str += "Cheveux : " + this.cheveux + "\n";
          str += "Taille : " + this.taille + "\n";
         }
         else{
          str = "Aucune information !";
         }
         return str;
        }
       }
```

L'avantage avec cette méthode, c'est que nous n'avons pas à nous soucier de savoir si nous avons annulé la saisie ou non : l'objet d'information renverra toujours quelque chose...

Voici le code source de notre boîte perso :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.BorderFactory;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.ButtonGroup;
import javax.swing.JTextField;


public class ZDialog extends JDialog {

 private ZDialogInfo zInfo = new ZDialogInfo();
 private JLabel nomLabel, sexeLabel, cheveuxLabel, ageLabel,
tailleLabel, taille2Label, icon;
 private JRadioButton tranche1, tranche2, tranche3, tranche4;
 private JComboBox sexe, cheveux;
 private JTextField nom, taille;

 /**
* Constructeur
* @param parent
* @param title
* @param modal
*/
 public ZDialog(JFrame parent, String title, boolean modal){
  super(parent, title, modal);
  this.setSize(550, 270);
  this.setLocationRelativeTo(null);
  this.setResizable(false);
  this.setDefaultCloseOperation(JDialog.DO_NOTHING_ON_CLOSE);
  this.initComponent();
 }

 /**
* Méthode appelée pour utiliser la boîte
* @return zInfo
*/
 public ZDialogInfo showZDialog(){
  this.setVisible(true);
  return this.zInfo;
 }

 /**
* Initialise le contenu de la boîte
*/
 private void initComponent(){
  //Icone
  icon = new JLabel(new ImageIcon("images/icone.jpg"));
  JPanel panIcon = new JPanel();
  panIcon.setBackground(Color.white);
  panIcon.setLayout(new BorderLayout());
  panIcon.add(icon);

  //Le nom
  JPanel panNom = new JPanel();
  panNom.setBackground(Color.white);
  panNom.setPreferredSize(new Dimension(220, 60));
  nom = new JTextField();
  nom.setPreferredSize(new Dimension(100, 25));
  panNom.setBorder(BorderFactory.createTitledBorder("Nom du
```

```
personnage"));
    nomLabel = new JLabel("Saisir un nom :");
    panNom.add(nomLabel);
    panNom.add(nom);

    //Le sexe
    JPanel panSexe = new JPanel();
    panSexe.setBackground(Color.white);
    panSexe.setPreferredSize(new Dimension(220, 60));
    panSexe.setBorder(BorderFactory.createTitledBorder("Sexe du
personnage"));
    sexe = new JComboBox();
    sexe.addItem("Masculin");
    sexe.addItem("Féminin");
    sexe.addItem("Indéterminé");
    sexeLabel = new JLabel("Sexe : ");
    panSexe.add(sexeLabel);
    panSexe.add(sexe);

    //L'âge
    JPanel panAge = new JPanel();
    panAge.setBackground(Color.white);
    panAge.setBorder(BorderFactory.createTitledBorder("Age du
personnage"));
    panAge.setPreferredSize(new Dimension(440, 60));
    tranche1 = new JRadioButton("15 - 25 ans");
    tranche1.setSelected(true);
    tranche2 = new JRadioButton("26 - 35 ans");
    tranche3 = new JRadioButton("36 - 50 ans");
    tranche4 = new JRadioButton("+ de 50 ans");
    ButtonGroup bg = new ButtonGroup();
    bg.add(tranche1);
    bg.add(tranche2);
    bg.add(tranche3);
    bg.add(tranche4);
    panAge.add(tranche1);
    panAge.add(tranche2);
    panAge.add(tranche3);
    panAge.add(tranche4);


    //La taille
    JPanel panTaille = new JPanel();
    panTaille.setBackground(Color.white);
    panTaille.setPreferredSize(new Dimension(220, 60));
    panTaille.setBorder(BorderFactory.createTitledBorder("Taille du
personnage"));
    tailleLabel = new JLabel("Taille : ");
    taille2Label = new JLabel(" cm");
    taille = new JTextField("180");
    taille.setPreferredSize(new Dimension(90, 25));
    panTaille.add(tailleLabel);
    panTaille.add(taille);
    panTaille.add(taille2Label);

    //La couleur des cheveux
    JPanel panCheveux = new JPanel();
    panCheveux.setBackground(Color.white);
    panCheveux.setPreferredSize(new Dimension(220, 60));
    panCheveux.setBorder(BorderFactory.createTitledBorder("Couleur de
cheveux du personnage"));
    cheveux = new JComboBox();
    cheveux.addItem("Blond");
    cheveux.addItem("Brun");
    cheveux.addItem("Roux");
    cheveux.addItem("Blanc");
    cheveuxLabel = new JLabel("Cheveux");
    panCheveux.add(cheveuxLabel);
    panCheveux.add(cheveux);
```

```
        JPanel content = new JPanel();
        content.setBackground(Color.white);
        content.add(panNom);
        content.add(panSexe);
        content.add(panAge);
        content.add(panTaille);
        content.add(panCheveux);

        JPanel control = new JPanel();
        JButton okBouton = new JButton("OK");

        okBouton.addActionListener(new ActionListener(){
         public void actionPerformed(ActionEvent arg0) {
          zInfo = new ZDialogInfo(nom.getText(),
      (String)sexe.getSelectedItem(), getAge(),
      (String)cheveux.getSelectedItem() ,getTaille());
          setVisible(false);
         }

         public String getAge(){
          return (tranche1.isSelected()) ? tranche1.getText() :
      (tranche2.isSelected()) ? tranche2.getText() :
      (tranche3.isSelected()) ? tranche3.getText() :
      (tranche4.isSelected()) ? tranche4.getText() : tranche1.getText();
         }

         public String getTaille(){
          return (taille.getText().equals("")) ? "180" : taille.getText();
         }
        });

        JButton cancelBouton = new JButton("Annuler");
        cancelBouton.addActionListener(new ActionListener(){
         public void actionPerformed(ActionEvent arg0) {
          setVisible(false);
         }
        });

        control.add(okBouton);
        control.add(cancelBouton);

        this.getContentPane().add(panIcon, BorderLayout.WEST);
        this.getContentPane().add(content, BorderLayout.CENTER);
        this.getContentPane().add(control, BorderLayout.SOUTH);
       }

      }
```

J'ai ajouté une image, mais vous n'y êtes nullement obligés !
Vous constaterez aussi que j'ai ajouté une bordure à nos `JPanel` afin de faire plus... Zoli...

Et le code source permettant de tester cette boîte :

Code : Java

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
```

```
public class Fenetre extends JFrame {

 private JButton bouton = new JButton("Appel à la ZDialog");

 public Fenetre(){

  this.setTitle("Ma JFrame");
  this.setSize(300, 100);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setLocationRelativeTo(null);

  this.getContentPane().setLayout(new FlowLayout());
  this.getContentPane().add(bouton);
  bouton.addActionListener(new ActionListener(){

   public void actionPerformed(ActionEvent arg0) {
    ZDialog zd = new ZDialog(null, "Coucou les ZérOs", true);
    ZDialogInfo zInfo = zd.showZDialog();
    JOptionPane jop = new JOptionPane();
    jop.showMessageDialog(null, zInfo.toString(), "Informations
personnage", JOptionPane.INFORMATION_MESSAGE);
   }

  });

  this.setVisible(true);

 }

 public static void main(String[] main){
  Fenetre fen = new Fenetre();
 }

}
```

Voici des screenshots obtenus

De la ZDialog

Lorsqu'on valide la saisie

Lorsqu'on annule la saisie

Voilà : nous arrivons à la fin de chapitre, riche en nouveautés...

En route pour le topo habituel et le petit QCM des familles...
Retour en haut

Ce qu'il faut retenir

- Les boîtes de dialogue s'utilisent, sauf boîtes personnalisées, avec l'objet **`JOptionPane`**.
- La méthode **`showMessageDialog()`** permet d'afficher un message informatif.
- La méthode **`showConfirmDialog()`** permet d'afficher une boîte attendant une réponse à une question ouverte (oui / non).
- La méthode citée ci-dessus retourne un entier correspondant au bouton sur lequel vous avez cliqué.
- La méthode **`showInputDialog()`** affiche une boîte attendant une saisie de la part de l'utilisateur ou une sélection dans une liste.
- Cette méthode retourne soit un **`String`** dans le cas d'une saisie, soit un **`Object`** dans le cas d'une liste.
- La méthode **`showOptionDialog()`** affiche une boîte attendant le clic sur une option proposée à l'utilisateur.
- Celle-ci retourne l'indice de l'élément sur lequel vous avez cliqué, **ou un indice négatif pour tous les autres cas !**
- Les boîtes de dialogue sont dites **modales** : aucune interaction hors de la boîte n'est possible tant que celle-ci n'est pas fermée !
- Pour faire une boîte de dialogue personnalisée, vous devez créer une classe héritée de **`JDialog`**.
- Pour les boîtes personnalisées, le dialogue commence lorsque la méthode `setVisible(true)` est invoquée et se termine lorsque la méthode `setVisible(false)` est appelée !

Retour en haut

Chapitre très simple, là aussi, mais je tenais à vous féliciter.

Vous avez dû remarquer que vous avez réellement progressé en Java depuis le début de ce tuto, et vous devez vous rendre compte qu'en définitive, faire des IHM en Java n'est pas si compliqué qu'il n'y paraît... Il y a beaucoup de choses à savoir, mais le plus important n'est pas de tout connaître par coeur, mais de savoir retrouver l'information rapidement !

Trêve de compliments, sinon vous ne travaillerez plus...

Je vous propose donc maintenant de continuer par un chapitre qui risque de vous plaire tout autant : **Les menus**.

Retour en haut

Les menus

Tout le monde sait ce qu'est un menu.
Une barre dans laquelle se trouve une ou plusieurs listes déroulantes, elles-mêmes composées de listes.
Vous avez un exemple sous les yeux, tout en haut de votre navigateur internet.

Dans ce chapitre nous allons voir comment ajouter ce genre de composants dans vos fenêtres, mais aussi comment les utiliser. Nous aborderons aussi la façon d'utiliser ce qu'on appelle des **accélérateurs** : par exemple, sous Firefox, si vous faites CTRL + T, vous ouvrez un nouvel onglet. Ce raccourci est un exemple d'accélérateur.

Au cours de ce chapitre nous verrons aussi les **mnémoniques** : dans la barre de menu de votre navigateur, vous pouvez voir qu'une des lettres de chaque élément de menu est soulignée, ceci signifie que si vous appuyez simultanément sur ALT + <la lettre soulignée>, vous déroulerez le menu correspondant. Voici ce qu'on nomme un mnémonique.

Ce type de menu, dit barre de menus, est le plus courant ; mais nous verrons aussi comment utiliser un menu contextuel.
Vous savez, celui qui apparaît lorsque vous faites un clic droit...

Et en bonus, nous apprendrons aussi à utiliser les barres d'outils.

Allez, assez de blabla ! Let's go.

La barre de menus et les éléments de menu

Vous vous rappelez que j'ai mentionné une `MenuBar` faisant partie de la composition de l'objet **JFrame**.

Le moment est venu pour vous d'utiliser un tel composant : cependant, celui cité ci-dessus appartient au package `java.awt`.
Dans ce chapitre nous utiliserons son homologue, l'objet **JMenuBar**, présent dans le package `javax.swing`.

Un menu simple

Afin de travailler avec des menus, nous allons avoir besoin :

- d'objets **JMenu** : titre global d'un point de menu. Regardez en haut de votre navigateur ;
- d'objets **JMenuItem** : éléments composants nos menus.

Ici, afin de pouvoir utiliser nos futurs menus, nous allons devoir coder des implémentations de l'interface **ActionListener** : vous la connaissez bien, celle-là !
Ces implémentations serviront à écouter les objets **JMenuItem**. Ce sont ces objets qui déclencheront tel ou tel traitement. Les **JMenu**, eux, ont un comportement automatique. Si on clique sur un titre de menu, celui-ci se déroule tout seul et, dans le cas où nous avons un tel objet présent dans un autre **JMenu**, une autre liste se déroulera toute seule !

En bref, nous n'avons à gérer qu'une partie de tous ces objets.
Bon : nous allons pouvoir commencer. Je vous propose donc d'enlever tous les composants (bouton, combo, checkbox) de notre animation et de gérer tout cela par le biais d'un menu.

Avant de nous lancer dans cette tâche, je vais vous montrer un exemple d'utilisation, histoire de vous familiariser. Voici un code d'exemple :

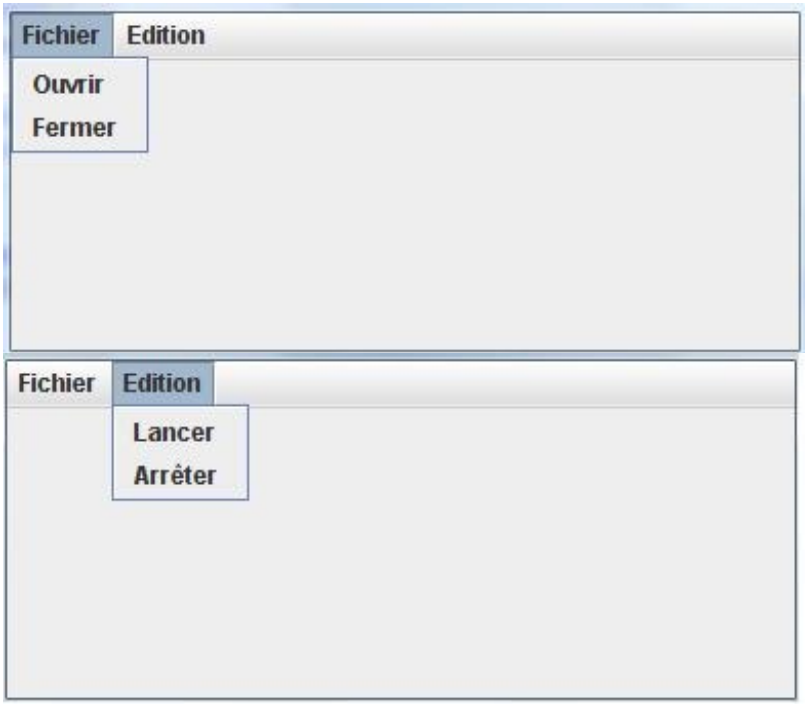
Code : Java

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;


public class ZFenetre extends JFrame {

 private JMenuBar menuBar = new JMenuBar();
 private JMenu test1 = new JMenu("Fichier");
 private JMenu test2 = new JMenu("Edition");

 private JMenuItem item1 = new JMenuItem("Ouvrir");
 private JMenuItem item2 = new JMenuItem("Fermer");
 private JMenuItem item3 = new JMenuItem("Lancer");
 private JMenuItem item4 = new JMenuItem("Arrêter");


 public ZFenetre(){
  this.setSize(400, 200);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setLocationRelativeTo(null);

  //On initialise nos menus
  //-------------------------

  this.test1.add(item1);
  item2.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent arg0) {
    System.exit(0);
   }
  });
```

```
        this.test1.add(item2);

        this.test2.add(item3);
        this.test2.add(item4);

                        //L'ordre d'ajout va déterminer l'ordre
    d'apparition dans le menu de gauche à droite
                        //Le premier ajouté sera tout à gauche de la barre
    de menu et inversement pour le dernier
           this.menuBar.add(test1);
           this.menuBar.add(test2);
           //-------------------------

           this.setJMenuBar(menuBar);
           this.setVisible(true);
         }
    }
```

L'action attachée au **JMenutItem** "***Fermer***" permet de quitter l'application.

Et le résultat de ce code :



Voici notre exemple un peu plus élaboré :

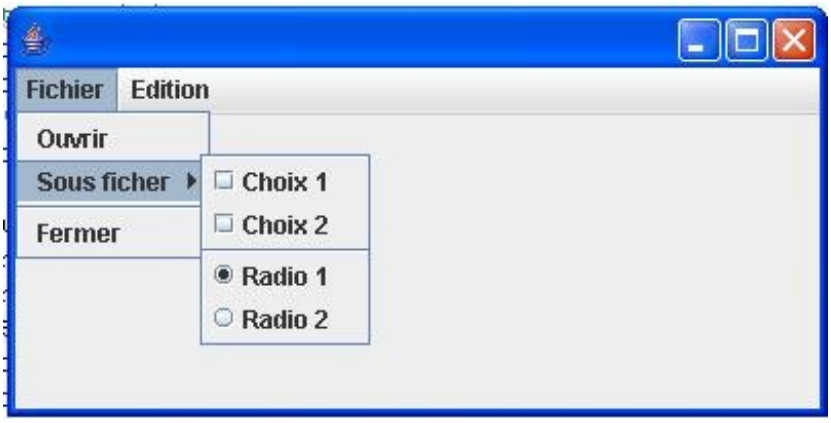
Code : Java

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.ButtonGroup;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JRadioButtonMenuItem;
```

```
public class ZFenetre extends JFrame {

 private JMenuBar menuBar = new JMenuBar();
 private JMenu test1 = new JMenu("Fichier");
 private JMenu test1_2 = new JMenu("Sous ficher");
 private JMenu test2 = new JMenu("Edition");

 private JMenuItem item1 = new JMenuItem("Ouvrir");
 private JMenuItem item2 = new JMenuItem("Fermer");
 private JMenuItem item3 = new JMenuItem("Lancer");
 private JMenuItem item4 = new JMenuItem("Arrêter");

 private JCheckBoxMenuItem jcmi1 = new JCheckBoxMenuItem("Choix 1");
 private JCheckBoxMenuItem jcmi2 = new JCheckBoxMenuItem("Choix 2");

 private JRadioButtonMenuItem jrmi1 = new
JRadioButtonMenuItem("Radio 1");
 private JRadioButtonMenuItem jrmi2 = new
JRadioButtonMenuItem("Radio 2");

 public static void main(String[] args){
  ZFenetre zFen = new ZFenetre();
 }

 public ZFenetre(){
  this.setSize(400, 200);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setLocationRelativeTo(null);

  //On initialise nos menus
  //-------------------------
  this.test1.add(item1);

  //On ajoute les éléments dans notre sous-menu
  this.test1_2.add(jcmi1);
  this.test1_2.add(jcmi2);
  //Ajout d'un séparateur
  this.test1_2.addSeparator();
  //On met nos radios dans un ButtonGroup
  ButtonGroup bg = new ButtonGroup();
  bg.add(jrmi1);
  bg.add(jrmi2);
                //On présélectionne la première radio
  jrmi1.setSelected(true);

  this.test1_2.add(jrmi1);
  this.test1_2.add(jrmi2);

  //Ajout du sous-menu dans notre menu
  this.test1.add(this.test1_2);
  //Ajout d'un séparateur
  this.test1.addSeparator();
  item2.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent arg0) {
    System.exit(0);
   }
  });
  this.test1.add(item2);

  this.test2.add(item3);
  this.test2.add(item4);

                //L'ordre d'ajout va déterminer l'ordre
d'apparition dans le menu de gauche à droite
                //Le premier ajouté sera tout à gauche de la barre
de menu et inversement pour le dernier
   this.menuBar.add(test1);
   this.menuBar.add(test2);
```

```
        //-------------------------

        this.setJMenuBar(menuBar);
        this.setVisible(true);
      }
    }
```

Et voilà le rendu de ce code :



Vous pouvez voir qu'il n'y a rien de difficile à créer un menu. Je vous propose donc d'en créer un pour notre animation. Nous allons faire ceci petit à petit. Nous gèrerons les événements ensuite. Pour le moment, nous allons avoir besoin :

- d'un menu **animation**, pour lancer, arrêter (par défaut à `setEnabled(false)`) ou quitter l'animation ;
- d'un menu **forme**, afin de savoir quel type de forme utiliser (sous-menu + une radio par forme) et si celle-ci est en mode morphing (case à cocher) ;
- d'un menu **à propos**, avec un zoli "?" qui va ouvrir une boîte de dialogue.

N'effacez surtout pas les implémentations pour les événements, retirez seulement les composants utilisant les implémentations et créez votre menu !

Vous serez sans doute obligés de mettre quelques instructions en commentaire... Mais nous y reviendrons.

Je vous laisse faire, vous devriez y arriver sans problème... Prenez votre temps, réfléchissez, et allez-y !

Voici le code que vous devriez avoir (ou un code s'en approchant) :

Code : Java

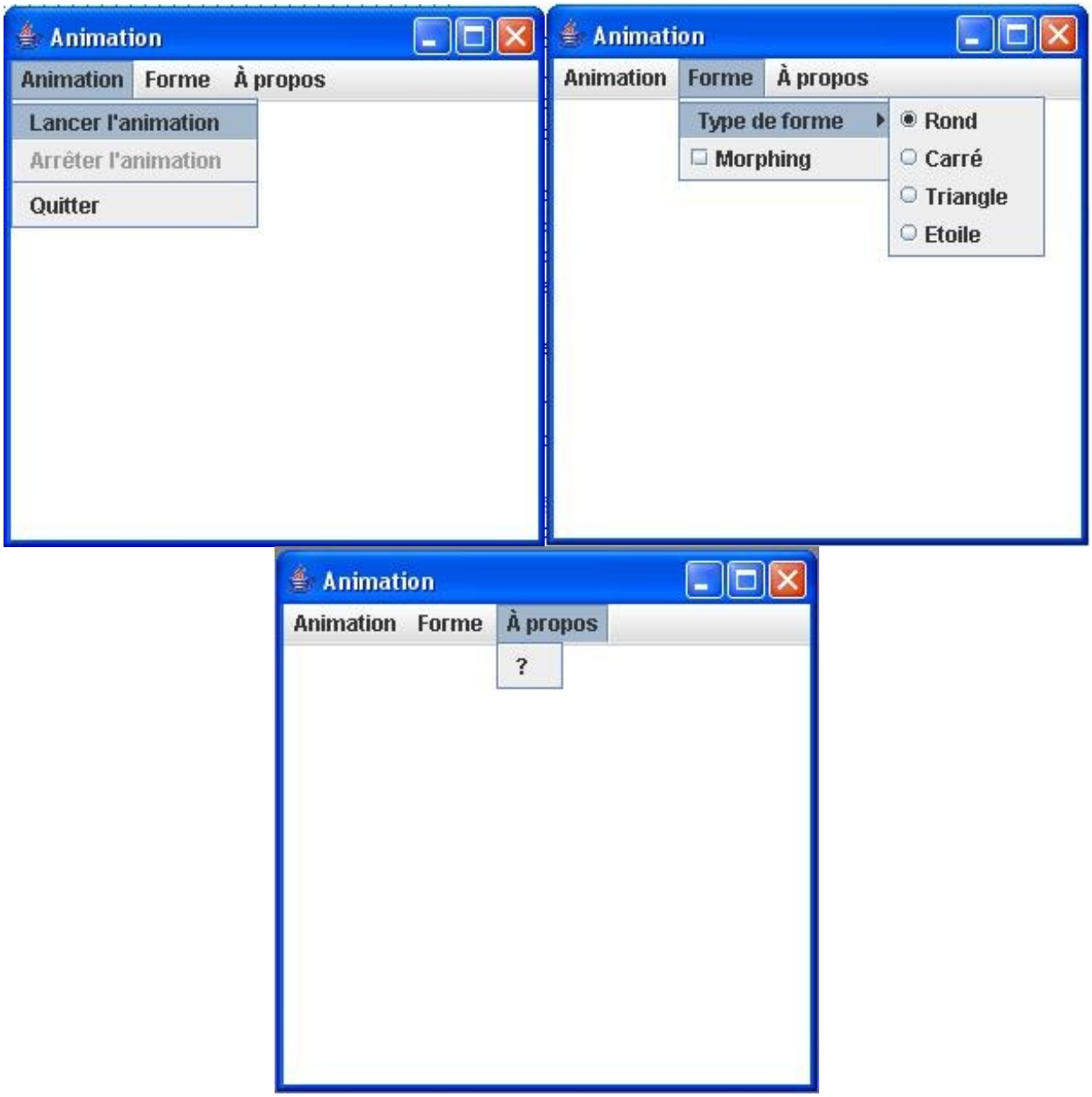
```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JRadioButtonMenuItem;
```

```
public class Fenetre extends JFrame{

 private Panneau pan = new Panneau();
    private JPanel container = new JPanel();
    private int compteur = 0;
    private boolean animated = true;
    private boolean backX, backY;
    private int x,y ;
    private Thread t;

    private JMenuBar menuBar = new JMenuBar();

    private JMenu animation = new JMenu("Animation"),
      forme = new JMenu("Forme"),
      typeForme = new JMenu("Type de forme"),
      aPropos = new JMenu("À propos");

    private JMenuItem lancer = new JMenuItem("Lancer l'animation"),
        arreter = new JMenuItem("Arrêter l'animation"),
        quitter = new JMenuItem("Quitter"),
        aProposItem = new JMenuItem("?");

    private JCheckBoxMenuItem morph = new
JCheckBoxMenuItem("Morphing");
    private JRadioButtonMenuItem carre = new
JRadioButtonMenuItem("Carré"),
           rond = new JRadioButtonMenuItem("Rond"),
           triangle = new JRadioButtonMenuItem("Triangle"),
           etoile = new JRadioButtonMenuItem("Etoile");

    private ButtonGroup bg = new ButtonGroup();

    public Fenetre(){

            this.setTitle("Animation");
            this.setSize(300, 300);
            this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            this.setLocationRelativeTo(null);

            container.setBackground(Color.white);
            container.setLayout(new BorderLayout());
            container.add(pan, BorderLayout.CENTER);

            this.setContentPane(container);
            this.initMenu();
            this.setVisible(true);

    }

    private void initMenu(){
     //Menu animation
     animation.add(lancer);
     arreter.setEnabled(false);
     animation.add(arreter);
     animation.addSeparator();
        //Pour quitter l'application
     quitter.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent event){
       System.exit(0);
      }
     });
     animation.add(quitter);

     //Menu forme

     bg.add(carre);
     bg.add(triangle);
     bg.add(rond);
     bg.add(etoile);
```

```
        typeForme.add(rond);
        typeForme.add(carre);
        typeForme.add(triangle);
        typeForme.add(etoile);

        rond.setSelected(true);

        forme.add(typeForme);
        forme.add(morph);

        //menu à propos
        aPropos.add(aProposItem);

        //Ajout des menus dans la barre de menus
        menuBar.add(animation);
        menuBar.add(forme);
        menuBar.add(aPropos);

        //Ajout de la barre de menus sur la fenêtre
        this.setJMenuBar(menuBar);
       }

    private void go(){
     x = pan.getPosX();
     y = pan.getPosY();
           while(this.animated){
            //System.out.println("OK");
            //Si le mode morphing est activé, on utilise la taille
actuelle de la forme
         if(pan.isMorph())
         {
          if(x < 1)backX = false;
                if(x > pan.getWidth() - pan.getDrawSize())backX = true;
                if(y < 1)backY = false;
                if(y > pan.getHeight() - pan.getDrawSize())backY =
true;
         }
         //Sinon, comme d'habitude
         else
         {
             if(x < 1)backX = false;
                if(x > pan.getWidth()-50)backX = true;
                if(y < 1)backY = false;
                if(y > pan.getHeight()-50)backY = true;
         }

            if(!backX)pan.setPosX(++x);
               else pan.setPosX(--x);
               if(!backY) pan.setPosY(++y);
               else pan.setPosY(--y);
               pan.repaint();

               try {
                       Thread.sleep(3);
               } catch (InterruptedException e) {
                       // TODO Auto-generated catch block
                       e.printStackTrace();
               }
           }
    }

    public class BoutonListener implements ActionListener{

     public void actionPerformed(ActionEvent arg0) {

      JOptionPane jop = new JOptionPane();
      int option = jop.showConfirmDialog(null, "Voulez-vous lancer
l'animation ?", "Lancement de l'animation",
JOptionPane.YES NO OPTION, JOptionPane.PLAIN MESSAGE);
```

```
        if(option == JOptionPane.OK_OPTION)
        {
         lancer.setEnabled(false);
         arreter.setEnabled(true);
         animated = true;
         t = new Thread(new PlayAnimation());
         t.start();
        }
       }
      }

      class Bouton2Listener  implements ActionListener{

       public void actionPerformed(ActionEvent e) {

        JOptionPane jop = new JOptionPane();
        int option = jop.showConfirmDialog(null, "Voulez-vous arrêter
l'animation ?", "Arrêt de l'animation",
JOptionPane.YES_NO_CANCEL_OPTION, JOptionPane.QUESTION_MESSAGE);

        if(option != JOptionPane.NO_OPTION && option !=
JOptionPane.CANCEL_OPTION && option != JOptionPane.CLOSED_OPTION)
        {
         animated = false;
         //On remplace nos boutons par nos JMenuItem
         lancer.setEnabled(true);
         arreter.setEnabled(false);
        }
       }
      }

      class PlayAnimation implements Runnable{

       public void run() {
        go();
       }
      }


      class FormeListener implements ActionListener{

       public void actionPerformed(ActionEvent e) {

        //On commente cette ligne pour l'instant
        //**************************************
        //pan.setForme(combo.getSelectedItem().toString());
       }
      }

      class MorphListener implements ActionListener{

       public void actionPerformed(ActionEvent e) {
        //Si la case est cochée, activation du mode morphing
        if(morph.isSelected())pan.setMorph(true);
        //Sinon rien !
        else pan.setMorph(false);
       }
      }
     }
```

Ce que vous devez obtenir :



Vous pouvez remarquer que notre IHM est beaucoup plus propre...

Il ne nous reste plus qu'à faire interagir nos menus avec notre animation ! Rien de plus simple, il suffit de dire à nos **MenuItem** qu'une implémentation les écoutent. En fait, cela revient à faire comme si nous cliquions sur des boutons, à l'exception des cases à cocher et des radios où, là, nous pouvons utiliser une implémentation d'**ActionListener** ou d'**ItemListener** : nous utiliserons la première.

Afin que l'application fonctionne bien, j'ai apporté deux modifications mineures dans la classe **Panneau** :

- ajout d'une instruction dans une condition, celle-ci :

Code : Java

```
//J'ai rajouté : || this.forme.equals("CARRÉ")
if(this.forme.equals("CARRE") || this.forme.equals("CARRÉ")){
 g.fillRect(posX, posY, 50, 50);
}
```

Pour accepter les deux orthographe !

- ajout d'un **toUpperCase()** dans :

Code : Java

```
    public void setForme(String form){
        this.forme = form.toUpperCase();
    }
```

Afin d'être sûr que le test sur cette chaîne de caractères soit sur des majuscules...

Voici le code de notre animation avec un beau menu pour tout contrôler :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.ButtonGroup;
import javax.swing.ImageIcon;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JRadioButtonMenuItem;

public class Fenetre extends JFrame{

 private Panneau pan = new Panneau();
    private JPanel container = new JPanel();
    private int compteur = 0;
    private boolean animated = true;
    private boolean backX, backY;
    private int x,y ;
    private Thread t;

    private JMenuBar menuBar = new JMenuBar();

    private JMenu animation = new JMenu("Animation"),
      forme = new JMenu("Forme"),
      typeForme = new JMenu("Type de forme"),
      aPropos = new JMenu("À propos");

    private JMenuItem lancer = new JMenuItem("Lancer l'animation"),
        arreter = new JMenuItem("Arrêter l'animation"),
        quitter = new JMenuItem("Quitter"),
        aProposItem = new JMenuItem("?");

    private JCheckBoxMenuItem morph = new
JCheckBoxMenuItem("Morphing");
    private JRadioButtonMenuItem carre = new
JRadioButtonMenuItem("Carré"),
            rond = new JRadioButtonMenuItem("Rond"),
            triangle = new JRadioButtonMenuItem("Triangle"),
            etoile = new JRadioButtonMenuItem("Etoile");

    private ButtonGroup bg = new ButtonGroup();

    public Fenetre(){

            this.setTitle("Animation");
            this.setSize(300, 300);
            this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            this.setLocationRelativeTo(null);
```

```
                container.setBackground(Color.white);
                container.setLayout(new BorderLayout());
                container.add(pan, BorderLayout.CENTER);

                this.setContentPane(container);
                this.initMenu();
                this.setVisible(true);

        }

        private void initMenu(){
         //Menu animation
         //***************************

         //Ajout du listener pour lancer l'animation
         lancer.addActionListener(new StartAnimationListener());
         animation.add(lancer);

         //Ajout du listener pour arrêter l'animation
         arreter.addActionListener(new StopAnimationListener());
         arreter.setEnabled(false);
         animation.add(arreter);

         animation.addSeparator();
         quitter.addActionListener(new ActionListener(){
          public void actionPerformed(ActionEvent event){
           System.exit(0);
          }
         });
         animation.add(quitter);

         //Menu forme

         bg.add(carre);
         bg.add(triangle);
         bg.add(rond);
         bg.add(etoile);

         //On crée un nouvel écouteur, inutile de créer 4 instances
    différentes
         FormeListener fl = new FormeListener();
         carre.addActionListener(fl);
         rond.addActionListener(fl);
         triangle.addActionListener(fl);
         etoile.addActionListener(fl);

         typeForme.add(rond);
         typeForme.add(carre);
         typeForme.add(triangle);
         typeForme.add(etoile);

         rond.setSelected(true);

         forme.add(typeForme);

         //Ajout du listener pour le morphing
         morph.addActionListener(new MorphListener());
         forme.add(morph);

         //menu à propos

         //Ajout de ce que doit faire le "?"
         aProposItem.addActionListener(new ActionListener(){
       public void actionPerformed(ActionEvent arg0) {
         JOptionPane jop = new JOptionPane();
         ImageIcon img = new ImageIcon("images/cysboy.gif");

         String mess = "Merci ! \n J'espère que vous vous amusez bien !
    \n";
         mess += "Je crois qu'il est temps d'ajouter des accélérateurs et
```

```
        des mnémoniques dans tout ça...\n";
            mess += "\n Allez, GO les ZérOs !";

            jop.showMessageDialog(null, mess, "À propos",
        JOptionPane.INFORMATION_MESSAGE, img);

           }
             });
             aPropos.add(aProposItem);

             //Ajout des menus dans la barre de menus
             menuBar.add(animation);
             menuBar.add(forme);
             menuBar.add(aPropos);

             //Ajouit de la barre de menus sur la fenêtre
             this.setJMenuBar(menuBar);
            }

         private void go(){
          x = pan.getPosX();
          y = pan.getPosY();
                while(this.animated){
                 //System.out.println("OK");
                 //Si le mode morphing est activé, on utilise la taille
        actuelle de la forme
              if(pan.isMorph())
              {
               if(x < 1)backX = false;
                     if(x > pan.getWidth() - pan.getDrawSize())backX = true;
                     if(y < 1)backY = false;
                     if(y > pan.getHeight() - pan.getDrawSize())backY =
        true;
              }
              //Sinon, comme d'habitude
              else
              {
                  if(x < 1)backX = false;
                     if(x > pan.getWidth()-50)backX = true;
                     if(y < 1)backY = false;
                     if(y > pan.getHeight()-50)backY = true;
              }

                 if(!backX)pan.setPosX(++x);
                    else pan.setPosX(--x);
                    if(!backY) pan.setPosY(++y);
                    else pan.setPosY(--y);
                    pan.repaint();

                    try {
                            Thread.sleep(3);
                    } catch (InterruptedException e) {
                            // TODO Auto-generated catch block
                            e.printStackTrace();
                    }
                }
         }

         /**
        * Ecouteur du menu Lancer
        * @author CHerby
        */
         public class StartAnimationListener implements ActionListener{

          public void actionPerformed(ActionEvent arg0) {

           JOptionPane jop = new JOptionPane();
           int option = jop.showConfirmDialog(null, "Voulez-vous lancer
        l'animation ?", "Lancement de l'animation",
        JOptionPane.YES NO OPTION, JOptionPane.PLAIN MESSAGE);
```

```
        if(option == JOptionPane.OK_OPTION)
        {
         lancer.setEnabled(false);
         arreter.setEnabled(true);
         animated = true;
         t = new Thread(new PlayAnimation());
         t.start();
        }
       }
      }

      /**
     * Ecouteur du menu Quitter
     * @author CHerby
     */
      class StopAnimationListener  implements ActionListener{

       public void actionPerformed(ActionEvent e) {

        JOptionPane jop = new JOptionPane();
        int option = jop.showConfirmDialog(null, "Voulez-vous arrêter
    l'animation ?", "Arrêt de l'animation",
    JOptionPane.YES_NO_CANCEL_OPTION, JOptionPane.QUESTION_MESSAGE);

        if(option != JOptionPane.NO_OPTION && option !=
    JOptionPane.CANCEL_OPTION && option != JOptionPane.CLOSED_OPTION)
        {
         animated = false;
         //On remplace nos boutons par nos JMenuItem
         lancer.setEnabled(true);
         arreter.setEnabled(false);
        }
       }
      }

      /**
     * Lance le thread.
     * @author CHerby
     */
      class PlayAnimation implements Runnable{
       public void run() {
        go();
       }
      }

      /**
     * Ecoute les menus forme
     * @author CHerby
     */
      class FormeListener implements ActionListener{
       public void actionPerformed(ActionEvent e) {
        pan.setForme(((JRadioButtonMenuItem)e.getSource()).getText());
       }
      }

      /**
     * Ecoute le menu Morphing
     * @author CHerby
     */
      class MorphListener implements ActionListener{
       public void actionPerformed(ActionEvent e) {
        //Si la case est cochée, activation du mode morphing
        if(morph.isSelected())pan.setMorph(true);
        //Sinon rien !
        else pan.setMorph(false);
       }
      }
     }
```

J'ai modifié le nom des implémentations et j'ai ajouté mon avatar du SDZ dans le dialogue du menu "À propos".
Mettez une image si vous avez envie...

Comme je l'ai dit dans le dialogue du menu "**À propos**", je crois qu'il est temps de mettre des raccourcis clavier dans tout ceci ! Vous êtes prêts ?

Les raccourcis clavier

En fait, ceci est très simple aussi. Pour ajouter un accélérateur sur un **JMenu**, nous appelerons la méthode `setAccelerator();` et pour ajouter un mnémonique sur un **JMenuItem**, nous invoquerons la méthode `setMnemonic();`.

Attribuons le mnémonique '**A**' pour le menu "**Animation**", le mnémonique '**F**' pour le menu "**Forme**" et enfin '**P**' pour "**À propos**". Vous allez voir, c'est très simple : il vous suffit d'invoquer la méthode `setMnemonic(char mnemonic);` sur le **JMenu** que vous désirez.

Ce qui nous donne, dans notre cas :

Code : Java

```
private void initMenu(){
      //Menu animation
      //***************************

      //Ajout du listener pour lancer l'animation
      lancer.addActionListener(new StartAnimationListener());
      animation.add(lancer);

      //Ajout du listener pour arrêter l'animation
      arreter.addActionListener(new StopAnimationListener());
      arreter.setEnabled(false);
      animation.add(arreter);

      animation.addSeparator();
      quitter.addActionListener(new ActionListener(){
       public void actionPerformed(ActionEvent event){
        System.exit(0);
       }
      });
      animation.add(quitter);

      //Menu forme

      bg.add(carre);
      bg.add(triangle);
      bg.add(rond);
      bg.add(etoile);

      //On crée un nouvel écouteur, inutile de créer 4 instances
différentes
      FormeListener fl = new FormeListener();
      carre.addActionListener(fl);
      rond.addActionListener(fl);
      triangle.addActionListener(fl);
      etoile.addActionListener(fl);

      typeForme.add(rond);
      typeForme.add(carre);
      typeForme.add(triangle);
      typeForme.add(etoile);
```

```
            rond.setSelected(true);

            forme.add(typeForme);

            //Ajout du listener pour le morphing
            morph.addActionListener(new MorphListener());
            forme.add(morph);

            //menu à propos

            //Ajout de ce que doit faire le "?"
            aProposItem.addActionListener(new ActionListener(){
          public void actionPerformed(ActionEvent arg0) {
            JOptionPane jop = new JOptionPane();
            ImageIcon img = new ImageIcon("images/cysboy.gif");

            String mess = "Merci ! \n J'espère que vous vous amusez bien !
\n";
            mess += "Je crois qu'il est temps d'ajouter des accélérateurs et
des mnémoniques dans tout ça...\n";
            mess += "\n Allez, GO les ZérOs !";

            jop.showMessageDialog(null, mess, "À propos",
JOptionPane.INFORMATION_MESSAGE, img);

          }
            });
            aPropos.add(aProposItem);

            //Ajout des menus dans la barre de menus ET AJOUT DE MNEMONICS
! !
            animation.setMnemonic('A');
          menuBar.add(animation);

          forme.setMnemonic('F');
          menuBar.add(forme);

          aPropos.setMnemonic('P');
          menuBar.add(aPropos);
            //Ajouit de la barre de menus sur la fenêtre
            this.setJMenuBar(menuBar);
          }
```

Nous avons à présent les lettres correspondantes soulignées dans nos menus, et mieux encore, si vous tapez **ALT + <la lettre>** : le menu correspondant se déroule !

Voici ce que j'obtiens :

Concernant les mnémoniques, vous devez savoir que vous pouvez aussi en mettre sur les objets **JMenuItem**.
Il faut que vous sachiez aussi qu'il existe une autre façon d'ajouter un mnémonique sur un **JMenu** mais **UNIQUEMENT SUR UN JMenu** : en passant le mnémonique en deuxième paramètre du constructeur de l'objet, comme ceci `JMenu menu = new JMenu("Fichier", 'F');//Ici, ce menu aura le mnémonique F .`

Oui, je sais, c'est simple et même très simple. Pour ajouter des accélérateurs, c'est quasiment pareil sauf que nous devrons utiliser un nouvel objet : **KeyStroke**.
Cet objet permet de connaître l'identité de la touche utilisée ou à utiliser. De ce fait, c'est grâce à cet objet que nous allons pouvoir construire des combinaisons de touches pour nos accélérateurs !

Nous allons commencer par attribuer un simple caractère comme accélérateur pour notre **JMenuItem** "**Lancer**", ceci en utilisant la méthode `getKeyStroke(char caracter);` de l'objet **KeyStroke**.

Rajoutez cette ligne de code au début de la méthode `initMenu()` :

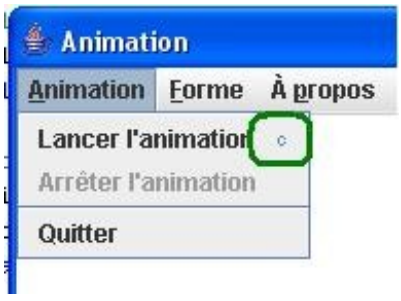
Code : Java

```
//Cette instruction ajoute l'accélérateur 'c' à notre objet
lancer.setAccelerator(KeyStroke.getKeyStroke('c'));
```

Vous aurez besoin de ces packages : `javax.swing.KeyStroke` et `java.awt.event.ActionEvent`

Testez votre application et vous devriez vous rendre compte qu'un petit '**c**' est apparu à côté du menu "**Lancer**".

Voyez plutôt :



Et, si vous appuyez sur la touche 'c' de votre clavier, celle-ci a le même effet qu'un clic sur le menu "**Lancer**" !

Attention : si vous mettez le caractère 'C', vous serez obligés d'appuyer simultanément sur **SHIFT + c** ou alors d'avoir la touche **MAJ** activée !

Le principe est bien, cependant, imaginez aussi que, maintenant, votre touche 'c' vous demandera systématiquement le lancement de votre animation !
C'est l'une des raisons pour laquelle les accélérateurs sont, en général, des combinaisons de touches du genre **CTRL + c** ou encore **CTRL + SHIFT + S**.

Pour réussir à faire ceci, nous allons utiliser une autre méthode `getKeyStroke()` : celle-ci prendra non pas le caractère de notre touche, mais son code, ainsi qu'une ou plusieurs touches faisant la combinaison !

Pour obtenir le code d'une touche, nous utiliserons l'objet **KeyEvent**, qui stocke tous les codes de touches !

Dans le code qui suit, je crée un accélérateur **CTRL + L** pour le menu "**Lancer**" et un accélérateur **CTRL + SHIT + A** pour le menu "**Arrêter**" :

Code : Java

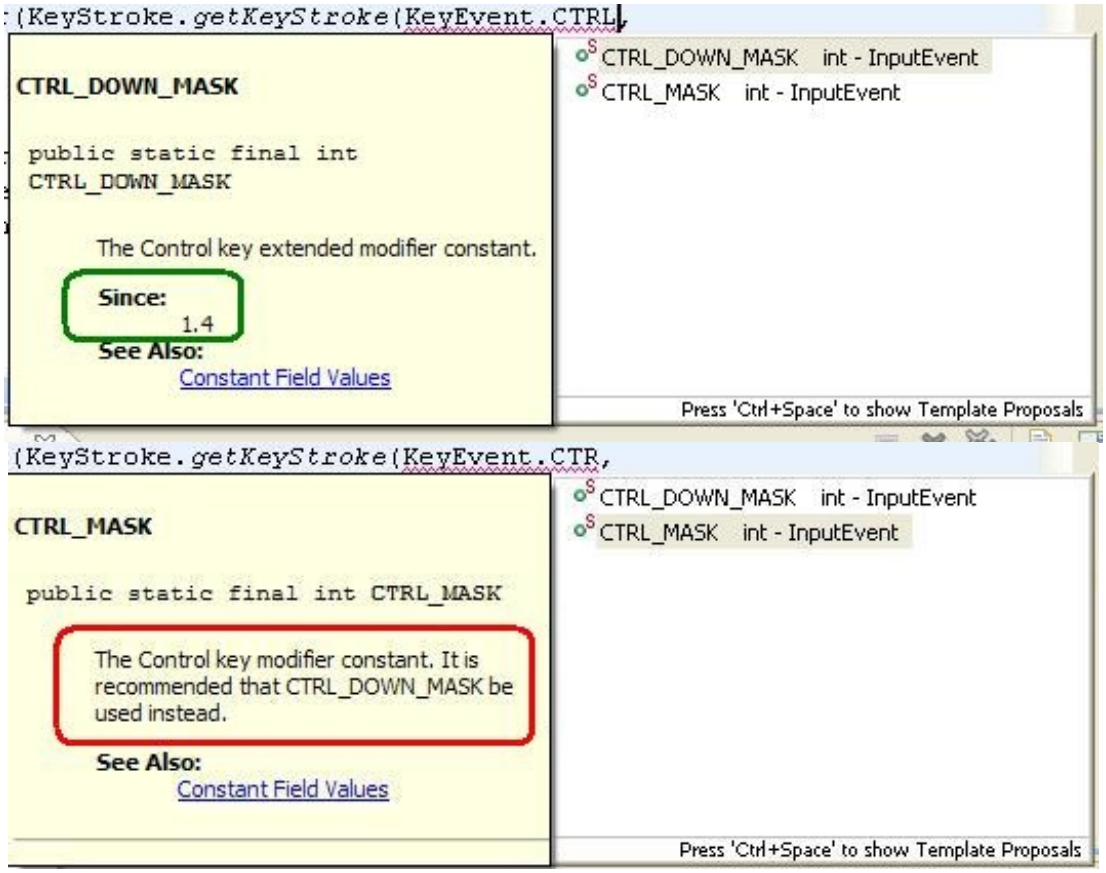
```
lancer.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_L,
                                             KeyEvent.CTRL_MASK));
     animation.add(lancer);

     //Ajout du listener pour arrêter l'animation
     arreter.addActionListener(new StopAnimationListener());
     arreter.setEnabled(false);
     arreter.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_A,

KeyEvent.CTRL_DOWN_MASK + KeyEvent.SHIFT_DOWN_MASK));
     animation.add(arreter);
```

Et ceci nous donne :

À ce moment-ci, j'imagine que vous devez être perturbés par ceci : `KeyEvent.VK_L` et les appels du même genre.

En fait, la classe **KeyEvent** répertorie tous les codes de toutes les touches du clavier. Une grande majorité sont sous la forme **VK_***<le caractère ou le nom de la touche>*, lisez ceci comme : **V**alue of **K**ey *<nom de la touche>*.
Mis à part certaines touches de contrôle comme **CTRL, ALT, SHIFT**... vous pouvez retrouver facilement le code d'une touche grâce à cet objet !

Ensuite, vous avez dû remarquer que lorsque vous avez voulu taper `KeyEvent.CTRL_DOWN_MASK`, Eclipse vous propose quasiment la même chose :



Vous pouvez aisément voir qu'Eclipse vous dit que la version **CTRL_DOWN_MASK** est la plus récente et qu'il est vivement conseillé de l'utiliser !

Maintenant que vous savez comment créer des mnémoniques et des accélérateurs, mettez-en où vous voulez ! Ceci dépend de vous...

Vous voilà donc avec un zoli menu avec des mnémoniques et des accélérateurs.
Il est donc temps de voir comment créer un menu contextuel !
Retour en haut

Faire un menu contextuel

Vous allez vous rendre compte que vous avez déjà fait le plus dur...
Nous allons seulement utiliser un autre objet : un **JPopupMenu**. Dans lequel nous allons mettre nos **JMenuItem** ou / et **JMenu**.

Il va falloir tout de même dire à notre menu contextuel comment s'afficher et surtout où, mais vous verrez que c'est très simple...

Maintenant que vous commencez à bien connaître les principes de bases de la programmation événementielle, nous allons aller plus vite pour apprendre de nouvelles choses !

Les points importants pour notre menu contextuel

- Les actions à effectuer, dans le cas d'actions identiques au menu, doivent être les mêmes : nous devrons donc créer des objets globaux aux deux menus.
- Le menu contextuel ne doit s'afficher que dans la zone où l'animation s'exécute, et non sur le menu !
- Il ne doit s'afficher que lorsqu'on fait un clic droit, et uniquement ça !

Nous allons mettre, dans notre menu contextuel, les actions "**Lancer l'animation**" et "**Arrêter l'animation**" ainsi que deux nouveautés :

- pouvoir changer la couleur du fond de notre animation ;
- avoir la possibilité de changer la couleur de notre forme !

Avant d'implémenter les deux nouvelles fonctionnalités, nous allons travailler sur les deux premières.
Ce qui signifie que, lorsque nous lancerons l'animation, nous devrons mettre les deux menus "**Lancer l'animation**" à l'état `setEnabled(false);` et les deux menus "**Arrêter l'animation**" à l'état `setEnabled(true);`, et inversement pour l'arrêt.

Comme je vous l'ai dit plus haut, nous allons utiliser le même objet écouteur pour le traitement des deux menus, nous allons devoir créer une véritable instance de ces objets et avertir que ces objets écoutent non seulement le menu du haut, mais aussi le menu contextuel.
Nous avons parfaitement le droit de faire ça : **plusieurs écouteurs peuvent écouter un composant et plusieurs composants peuvent avoir le même écouteur !**

Vous êtes presque prêts à créer votre menu contextuel, il ne vous manque que trois informations :

- comment dire à notre panneau d'afficher le menu contextuel ? Et où ?
- comment lui spécifier qu'il doit le faire uniquement sur le clic droit ?

Le déclenchement de l'affichage de la popup doit se faire sur un clic de souris, vous connaissez une interface qui gère ce type d'événement : l'interface **MouseListener**. Nous allons donc dire à notre panneau qu'un écouteur du type de cet interface va l'écouter !

Tout comme lors du chapitre sur les zones de saisie, il existe une classe qui contient toutes les méthodes de la dite interface : la classe **MouseAdapter**.
Vos pouvez utiliser celle-ci pour créer une implémentation afin de ne redéfinir que la méthode dont vous avez besoin !
C'est cette solution que nous allons utiliser.
Quelle méthode doit-on redéfinir ? **mouseClicked()** ?

Si vous voulez, mais je pensais plutôt à **mouseReleased()**, pour une raison simple que vous ne devez sûrement pas connaître : ces deux événements sont quasiment identiques, cependant, dans un certain cas, seul l'événement **mouseClicked()** sera appelé. Il s'agit du cas où vous cliquez sur une zone, que vous déplacez votre souris tout en maintenant le clic et que vous relâchez le bouton de la souris ensuite.

C'est pour cette raison que je préfère utiliser la méthode **mouseReleased()**. Ensuite, pour spécifier où afficher le menu contextuel, la classe **JPopupMenu** possède une méthode `show(Component invoker, int x, int y);`.

- `Component invoker` : désigne l'objet invoquant le menu contextuel, dans notre cas, notre instance de Panneau.
- `int x`: coordonnée X du menu.
- `int y`: Coordonnée Y du menu.

Souvenez-vous que vous pouvez déterminer les coordonnées de la souris grâce à l'objet passé en paramètre de la méthode `mouseReleased(MouseEvent event)`.

Je suis sûr que vous savez comment vous y prendre pour dire au menu contextuel de s'afficher, il ne vous manque plus qu'à détecter le clic droit. Et là, l'objet **MouseEvent** va vous sauver la mise !
En effet, cet objet possède une méthode `isPopupTrigger()` qui renvoie vrai s'il s'agit d'un clic droit.

Vous avez tous les éléments en mains pour faire votre menu contextuel, rappelez-vous que nous ne gérons pas tout de suite les nouvelles fonctionnalités...

Quelques instants de réflexion... Vous avez fini ? Nous pouvons comparer nos codes ?

Secret (cliquez pour afficher)

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

import javax.swing.ButtonGroup;
import javax.swing.ImageIcon;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JPopupMenu;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.KeyStroke;

public class Fenetre extends JFrame{

 private Panneau pan = new Panneau();
    private JPanel container = new JPanel();
    private int compteur = 0;
    private boolean animated = true;
    private boolean backX, backY;
    private int x,y ;
    private Thread t;



    //*******************************************************
    // La déclaration pour le menu de la JMenuBar
    //*******************************************************
    private JMenuBar menuBar = new JMenuBar();

    private JMenu animation = new JMenu("Animation"),
```

```
            forme = new JMenu("Forme"),
            typeForme = new JMenu("Type de forme"),
            aPropos = new JMenu("À propos");

        private JMenuItem  lancer = new JMenuItem("Lancer
    l'animation"),
                arreter = new JMenuItem("Arrêter l'animation"),
                quitter = new JMenuItem("Quitter"),
                aProposItem = new JMenuItem("?");

        private JCheckBoxMenuItem morph = new
    JCheckBoxMenuItem("Morphing");
        private JRadioButtonMenuItem  carre = new
    JRadioButtonMenuItem("Carré"),
                rond = new JRadioButtonMenuItem("Rond"),
                triangle = new JRadioButtonMenuItem("Triangle"),
                etoile = new JRadioButtonMenuItem("Etoile");

        private ButtonGroup bg = new ButtonGroup();

        //*************************************************************
        // La déclaration pour le menu contextuel
        //*************************************************************
        private JPopupMenu jpm = new JPopupMenu();
        private JMenu background = new JMenu("Couleur de fond");
        private JMenu couleur = new JMenu("Couleur de la forme");

        private JMenuItem launch = new JMenuItem("Lancer
    l'animation");
        private JMenuItem stop = new JMenuItem("Arrêter l'animation");
        private JMenuItem  rouge = new JMenuItem("Rouge"),
             bleu = new JMenuItem("Bleu"),
             vert = new JMenuItem("Vert"),
             rougeBack = new JMenuItem("Rouge"),
             bleuBack = new JMenuItem("Bleu"),
             vertBack = new JMenuItem("Vert");

        //*************************************************************
        // ON CRÉE DES LISTENER GLOBAUX
        //*************************************************************
        private StopAnimationListener stopAnimation = new
    StopAnimationListener();
        private StartAnimationListener startAnimation = new
    StartAnimationListener();


        //*************************************************************

        public Fenetre(){

                this.setTitle("Animation");
                this.setSize(300, 300);
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setLocationRelativeTo(null);

                container.setBackground(Color.white);
                container.setLayout(new BorderLayout());

                //On initialise le menu stop
                stop.setEnabled(false);
                //On affecte les écouteurs
                stop.addActionListener(stopAnimation);
          launch.addActionListener(startAnimation);

          //On crée et on passe l'écouteur pour afficher le menu
    contextuel
          //Création d'une implémentation de MouseAdapter
          //avec redéfinition de la méthode adéquate
                pan.addMouseListener(new MouseAdapter(){
                 public void mouseReleased(MouseEvent event){
```

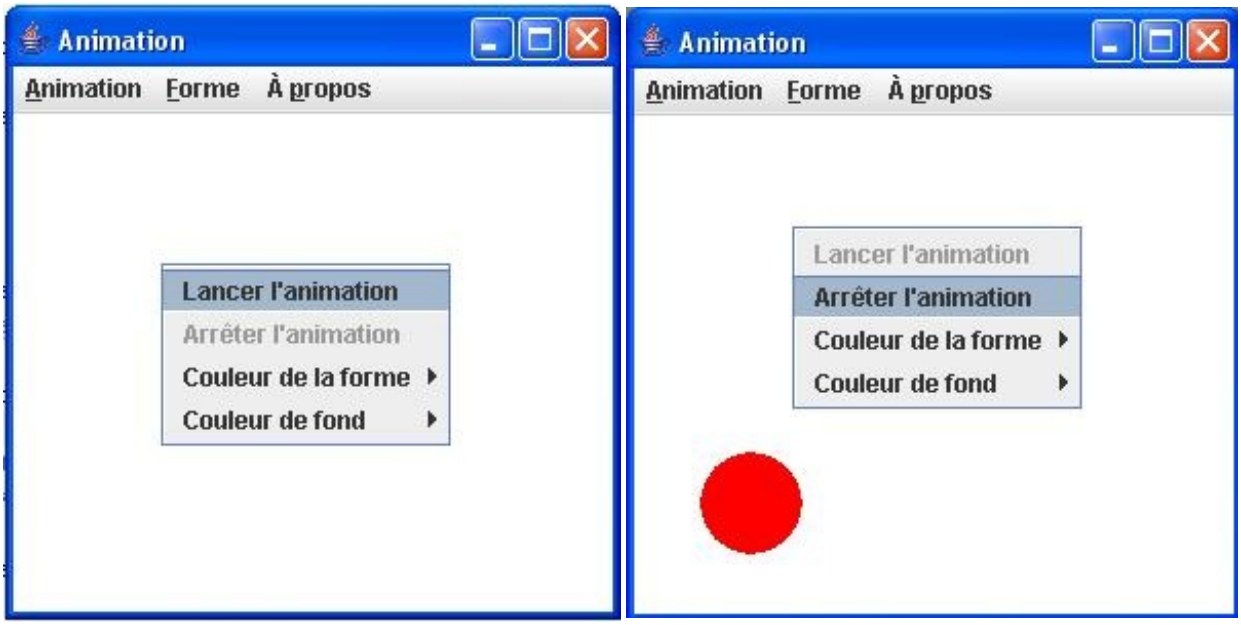
```
                //Seulement s'il s'agit d'un clic droit
                //Vous pouvez aussi utiliser cette méthode pour
détecter le clic droit
                        //if(event.getButton() ==
MouseEvent.BUTTON3)
                        if(event.isPopupTrigger())
                {
                 background.add(rougeBack);
                 background.add(bleuBack);
                 background.add(vertBack);

                 couleur.add(rouge);
                 couleur.add(bleu);
                 couleur.add(vert);

                 jpm.add(launch);
                 jpm.add(stop);
                 jpm.add(couleur);
                 jpm.add(background);

                 //La méthode qui va afficher le menu
                 jpm.show(pan, event.getX(), event.getY());
                }
               }
              });

              container.add(pan, BorderLayout.CENTER);

              this.setContentPane(container);
              this.initMenu();
              this.setVisible(true);

    }

    private void initMenu(){
     //Menu animation
     //***************************

     //Ajout du listener pour lancer l'animation
     //ATTENTION LE LISTENER EST GLOBAL ! ! ! !
     //-----------------------------------------------
     lancer.addActionListener(startAnimation);
     //On attribut l'accélerateur c
     lancer.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_L,
                KeyEvent.CTRL_MASK));
     animation.add(lancer);

     //Ajout du listener pour arrêter l'animation
     //LISTENER A CHANGER ICI AUSSI
     //-----------------------------------------------
     arreter.addActionListener(stopAnimation);
     arreter.setEnabled(false);
     arreter.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_A,
                KeyEvent.CTRL_DOWN_MASK +
KeyEvent.SHIFT_DOWN_MASK));
     animation.add(arreter);

     animation.addSeparator();
     quitter.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent event){
       System.exit(0);
      }
     });
     animation.add(quitter);

     //Menu forme

     bg.add(carre);
     bg.add(triangle);
     bg.add(rond);
```

```
        bg.add(etoile);

        //On crée un nouvel écouteur, inutile de créer 4 instances
différentes
        FormeListener fl = new FormeListener();
        carre.addActionListener(fl);
        rond.addActionListener(fl);
        triangle.addActionListener(fl);
        etoile.addActionListener(fl);

        typeForme.add(rond);
        typeForme.add(carre);
        typeForme.add(triangle);
        typeForme.add(etoile);

        rond.setSelected(true);

        forme.add(typeForme);

        //Ajout du listener pour le morphing
        morph.addActionListener(new MorphListener());
        forme.add(morph);

        //menu à propos

        //Ajout de ce que doit faire le "?"
        aProposItem.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent arg0) {
       JOptionPane jop = new JOptionPane();
       ImageIcon img = new ImageIcon("images/cysboy.gif");

       String mess = "Merci ! \n J'espère que vous vous amusez bien !
\n";
       mess += "Je crois qu'il est temps d'ajouter des accélérateurs
et des mnémoniques dans tout ça...\n";
       mess += "\n Allez, GO les ZérOs !";

       jop.showMessageDialog(null, mess, "À propos",
JOptionPane.INFORMATION_MESSAGE, img);

      }
       });
        aPropos.add(aProposItem);

        //Ajout des menus dans la barre de menus
        animation.setMnemonic('A');
        menuBar.add(animation);

        forme.setMnemonic('F');
        menuBar.add(forme);

        aPropos.setMnemonic('P');
        menuBar.add(aPropos);

        //Ajouit de la barre de menus sur la fenêtre
        this.setJMenuBar(menuBar);
       }

    private void go(){
     x = pan.getPosX();
     y = pan.getPosY();
           while(this.animated){
            //System.out.println("OK");
            //Si le mode morphing est activé, on utilise la taille
actuelle de la forme
         if(pan.isMorph())
         {
          if(x < 1)backX = false;
                if(x > pan.getWidth() - pan.getDrawSize())backX =
true;
```

```
                if(y < 1)backY = false;
                if(y > pan.getHeight() - pan.getDrawSize())backY =
true;
        }
        //Sinon, comme d'habitude
        else
        {
            if(x < 1)backX = false;
                if(x > pan.getWidth()-50)backX = true;
                if(y < 1)backY = false;
                if(y > pan.getHeight()-50)backY = true;
        }

           if(!backX)pan.setPosX(++x);
              else pan.setPosX(--x);
              if(!backY) pan.setPosY(++y);
              else pan.setPosY(--y);
              pan.repaint();

              try {
                      Thread.sleep(3);
              } catch (InterruptedException e) {
                      // TODO Auto-generated catch block
                      e.printStackTrace();
              }
          }
  }

  /**
 * Écouteur du menu Lancer
 * @author CHerby
 */
  public class StartAnimationListener implements ActionListener{

    public void actionPerformed(ActionEvent arg0) {

     JOptionPane jop = new JOptionPane();
     int option = jop.showConfirmDialog(null, "Voulez-vous lancer
l'animation ?", "Lancement de l'animation",
JOptionPane.YES_NO_OPTION, JOptionPane.PLAIN_MESSAGE);

     if(option == JOptionPane.OK_OPTION)
     {
      lancer.setEnabled(false);
      arreter.setEnabled(true);

      //ON AJOUTE L'INSTRUCTION POUR LE MENU CONTEXTUEL
      //*********************************************
      launch.setEnabled(false);
      stop.setEnabled(true);

      animated = true;
      t = new Thread(new PlayAnimation());
      t.start();
     }
    }
  }

  /**
 * Écouteur du menu Quitter
 * @author CHerby
 */
  class StopAnimationListener  implements ActionListener{

    public void actionPerformed(ActionEvent e) {

     JOptionPane jop = new JOptionPane();
     int option = jop.showConfirmDialog(null, "Voulez-vous arrêter
l'animation ?", "Arrêt de l'animation",
JOptionPane.YES NO CANCEL OPTION, JOptionPane.QUESTION MESSAGE);
```

```
    if(option != JOptionPane.NO_OPTION && option !=
JOptionPane.CANCEL_OPTION && option != JOptionPane.CLOSED_OPTION)
    {
     animated = false;
     //On remplace nos boutons par nos JMenuItem
     lancer.setEnabled(true);
     arreter.setEnabled(false);

     //ON AJOUTE L'INSTRUCTION POUR LE MENU CONTEXTUEL
     //*********************************************
     launch.setEnabled(true);
     stop.setEnabled(false);
    }
   }
  }

  /**
 * Lance le thread.
 * @author CHerby
 */
  class PlayAnimation implements Runnable{
   public void run() {
    go();
   }
  }

  /**
 * Écoute les menus forme
 * @author CHerby
 */
  class FormeListener implements ActionListener{
   public void actionPerformed(ActionEvent e) {
    pan.setForme(((JRadioButtonMenuItem)e.getSource()).getText());
   }
  }

  /**
 * Écoute le menu Morphing
 * @author CHerby
 */
  class MorphListener implements ActionListener{
   public void actionPerformed(ActionEvent e) {
    //Si la case est cochée, activation du mode morphing
    if(morph.isSelected())pan.setMorph(true);
    //Sinon rien !
    else pan.setMorph(false);
   }
  }
 }
```

Voici ce que j'obtiens :



Victoire !
Il est beau, il est fonctionnel, il est zérotissime notre menu contextuel !

Je sens que vous êtes prêts pour les nouvelles fonctionnalités... Même si je me doute que certains d'entre vous ont déjà fait ce qu'il fallait.

Il n'est pas très difficile de faire ce genre de chose, surtout que vous êtes habitués, maintenant.
Dans notre classe **Panneau**, nous utilisons des couleurs définies à l'avance, il nous suffit donc de mettre ces couleurs dans des variables et de permettre l'édition de celles-ci.

Rien de difficile ici, voici donc les codes sources de nos deux classes :

Panneau.java

Code : Java

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Image;
import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;
import javax.swing.JPanel;

public class Panneau extends JPanel {

 private int posX = -50;
    private int posY = -50;
    private int drawSize = 50;
    //boolean pour le mode morphing et pour savoir si la taille
doit être réduite
    private boolean morph = false, reduce = false;;
    private String forme = "ROND";

    //*********************************
    //Voici nos deux couleurs
    //*********************************
    private Color couleurForme = Color.red;
    private Color couleurFond = Color.white;

    //Le compteur de rafraîchissement
    private int increment = 0;
```

```
        public void paintComponent(Graphics g){
            //affectation de la couleur de fond
         g.setColor(couleurFond);
                g.fillRect(0, 0, this.getWidth(), this.getHeight());

                //Affectation de la couleur de la forme
                g.setColor(couleurForme);
                //Si le mode morphing est activé, on peint le morphing
                if(this.morph)
                 drawMorph(g);
                //sinon, mode normal
                else
                 draw(g);
        }



        /**
    * Méthode qui redéfinit la couleur du fond
    * @param color
    */
        public void setCouleurFond(Color color){
         this.couleurFond = color;
        }

        /**
    * Méthode qui redéfinit la couleur de la forme
    * @param color
    */
        public void setCouleurForme(Color color){
         this.couleurForme = color;
        }


        private void draw(Graphics g){

         if(this.forme.equals("ROND")){
          g.fillOval(posX, posY, 50, 50);
         }
         if(this.forme.equals("CARRE") || this.forme.equals("CARRÉ")){
          g.fillRect(posX, posY, 50, 50);
         }
         if(this.forme.equals("TRIANGLE")){

          int s1X = posX + 50/2;
          int s1Y = posY;
          int s2X = posX + 50;
          int s2Y = posY + 50;
          int s3X = posX;
          int s3Y = posY + 50;

          int[] ptsX = {s1X, s2X, s3X};
          int[] ptsY = {s1Y, s2Y, s3Y};

          g.fillPolygon(ptsX, ptsY, 3);
         }
         if(this.forme.equals("ETOILE")){

          int s1X = posX + 50/2;
          int s1Y = posY;
          int s2X = posX + 50;
          int s2Y = posY + 50;
          g.drawLine(s1X, s1Y, s2X, s2Y);

          int s3X = posX;
          int s3Y = posY + 50/3;
          g.drawLine(s2X, s2Y, s3X, s3Y);

          int s4X = posX + 50;
          int s4Y = posY + 50/3;
```

```
            g.drawLine(s3X, s3Y, s4X, s4Y);

            int s5X = posX;
            int s5Y = posY + 50;
            g.drawLine(s4X, s4Y, s5X, s5Y);
            g.drawLine(s5X, s5Y, s1X, s1Y);
           }

        }

        /**
     * Méthode qui peint le morphing
     * @param g
     */
        private void drawMorph(Graphics g){
         //On incrémente le tour
         increment++;
         //On regarde si on doit réduire ou non
         if(drawSize >= 50)reduce = true;
         if(drawSize <= 10)reduce = false;

         if(reduce)
          drawSize = drawSize - getUsedSize();
         else
          drawSize = drawSize + getUsedSize();

         if(this.forme.equals("ROND")){
          g.fillOval(posX, posY, drawSize, drawSize);
         }
         if(this.forme.equals("CARRE") || this.forme.equals("CARRÉ")){
          g.fillRect(posX, posY, drawSize, drawSize);
         }
         if(this.forme.equals("TRIANGLE")){

          int s1X = posX + drawSize/2;
          int s1Y = posY;
          int s2X = posX + drawSize;
          int s2Y = posY + drawSize;
          int s3X = posX;
          int s3Y = posY + drawSize;

          int[] ptsX = {s1X, s2X, s3X};
          int[] ptsY = {s1Y, s2Y, s3Y};

          g.fillPolygon(ptsX, ptsY, 3);
         }
         if(this.forme.equals("ETOILE")){

          int s1X = posX + drawSize/2;
          int s1Y = posY;
          int s2X = posX + drawSize;
          int s2Y = posY + drawSize;
          g.drawLine(s1X, s1Y, s2X, s2Y);

          int s3X = posX;
          int s3Y = posY + drawSize/3;
          g.drawLine(s2X, s2Y, s3X, s3Y);

          int s4X = posX + drawSize;
          int s4Y = posY + drawSize/3;
          g.drawLine(s3X, s3Y, s4X, s4Y);

          int s5X = posX;
          int s5Y = posY + drawSize;
          g.drawLine(s4X, s4Y, s5X, s5Y);
          g.drawLine(s5X, s5Y, s1X, s1Y);
         }


        }
```

```
    /**
* Méthode qui retourne le nombre à retrancher ou ajouter pour le
morphing
* @return res
*/
    private int getUsedSize(){
     int res = 0;
     //Si le nombre de tours est 10
     //On réinitialise l'incrément et on retourne 1
     if(increment / 10 == 1){
      increment = 0;
      res = 1;
     }
     return res;
    }

    public int getDrawSize(){
     return drawSize;
    }

    public boolean isMorph(){
     return morph;
    }

    public void setMorph(boolean bool){
     this.morph = bool;
     //On réinitialise la taille
     drawSize = 50;
    }

    public void setForme(String form){
     this.forme = form.toUpperCase();
    }

    public int getPosX() {
            return posX;
    }

    public void setPosX(int posX) {
            this.posX = posX;
    }

    public int getPosY() {
            return posY;
    }

    public void setPosY(int posY) {
            this.posY = posY;
    }

}
```

Fenetre.java

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
```

```
import javax.swing.ButtonGroup;
import javax.swing.ImageIcon;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JPopupMenu;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.KeyStroke;

public class Fenetre extends JFrame{

 private Panneau pan = new Panneau();
    private JPanel container = new JPanel();
    private int compteur = 0;
    private boolean animated = true;
    private boolean backX, backY;
    private int x,y ;
    private Thread t;



    //*******************************************************
    // La déclaration pour le menu de la JMenuBar
    //*******************************************************
    private JMenuBar menuBar = new JMenuBar();

    private JMenu animation = new JMenu("Animation"),
      forme = new JMenu("Forme"),
      typeForme = new JMenu("Type de forme"),
      aPropos = new JMenu("À propos");

    private JMenuItem  lancer = new JMenuItem("Lancer l'animation"),
         arreter = new JMenuItem("Arrêter l'animation"),
         quitter = new JMenuItem("Quitter"),
         aProposItem = new JMenuItem("?");

    private JCheckBoxMenuItem morph = new
JCheckBoxMenuItem("Morphing");
    private JRadioButtonMenuItem  carre = new
JRadioButtonMenuItem("Carré"),
            rond = new JRadioButtonMenuItem("Rond"),
            triangle = new JRadioButtonMenuItem("Triangle"),
            etoile = new JRadioButtonMenuItem("Etoile");

    private ButtonGroup bg = new ButtonGroup();

    //*******************************************************
    // La déclaration pour le menu contextuel
    //*******************************************************
    private JPopupMenu jpm = new JPopupMenu();
    private JMenu background = new JMenu("Couleur de fond");
    private JMenu couleur = new JMenu("Couleur de la forme");

    private JMenuItem launch = new JMenuItem("Lancer l'animation");
    private JMenuItem stop = new JMenuItem("Arrêter l'animation");
    private JMenuItem  rouge = new JMenuItem("Rouge"),
         bleu = new JMenuItem("Bleu"),
         vert = new JMenuItem("Vert"),
         blanc = new JMenuItem("Blanc"),
         rougeBack = new JMenuItem("Rouge"),
         bleuBack = new JMenuItem("Bleu"),
         vertBack = new JMenuItem("Vert"),
         blancBack = new JMenuItem("Blanc");

    //*******************************************************
    // ON CRÉE DES LISTENER GLOBAUX
```

```
        //*******************************************************
        private StopAnimationListener stopAnimation = new
    StopAnimationListener();
        private StartAnimationListener startAnimation = new
    StartAnimationListener();
        private CouleurFondListener bgColor = new CouleurFondListener();
        private CouleurFormeListener frmColor = new
    CouleurFormeListener();
        //*******************************************************

        public Fenetre(){

                this.setTitle("Animation");
                this.setSize(300, 300);
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setLocationRelativeTo(null);

                container.setBackground(Color.white);
                container.setLayout(new BorderLayout());

                //On initialise le menu stop
                stop.setEnabled(false);
                //On affecte les écouteurs
                stop.addActionListener(stopAnimation);
          launch.addActionListener(startAnimation);

          //On affecte les écouteurs aux points de menu
          rouge.addActionListener(frmColor);
          bleu.addActionListener(frmColor);
          vert.addActionListener(frmColor);
          blanc.addActionListener(frmColor);

          rougeBack.addActionListener(bgColor);
          bleuBack.addActionListener(bgColor);
          vertBack.addActionListener(bgColor);
          blancBack.addActionListener(bgColor);
          //On crée et on passe l'écouteur pour afficher le menu
    contextuel
          //Création d'une implémentation de MouseAdapter
          //avec redéfinition de la méthode adéquate
                pan.addMouseListener(new MouseAdapter(){
                 public void mouseReleased(MouseEvent event){
                  //Seulement s'il s'agit d'un clic droit
                  if(event.isPopupTrigger())
                  {
                   background.add(blancBack);
                   background.add(rougeBack);
                   background.add(bleuBack);
                   background.add(vertBack);

                   couleur.add(blanc);
                   couleur.add(rouge);
                   couleur.add(bleu);
                   couleur.add(vert);

                   jpm.add(launch);
                   jpm.add(stop);
                   jpm.add(couleur);
                   jpm.add(background);

                   //La méthode qui va afficher le menu
                   jpm.show(pan, event.getX(), event.getY());
                  }
                 }
                });

                container.add(pan, BorderLayout.CENTER);

                this.setContentPane(container);
                this.initMenu();
```

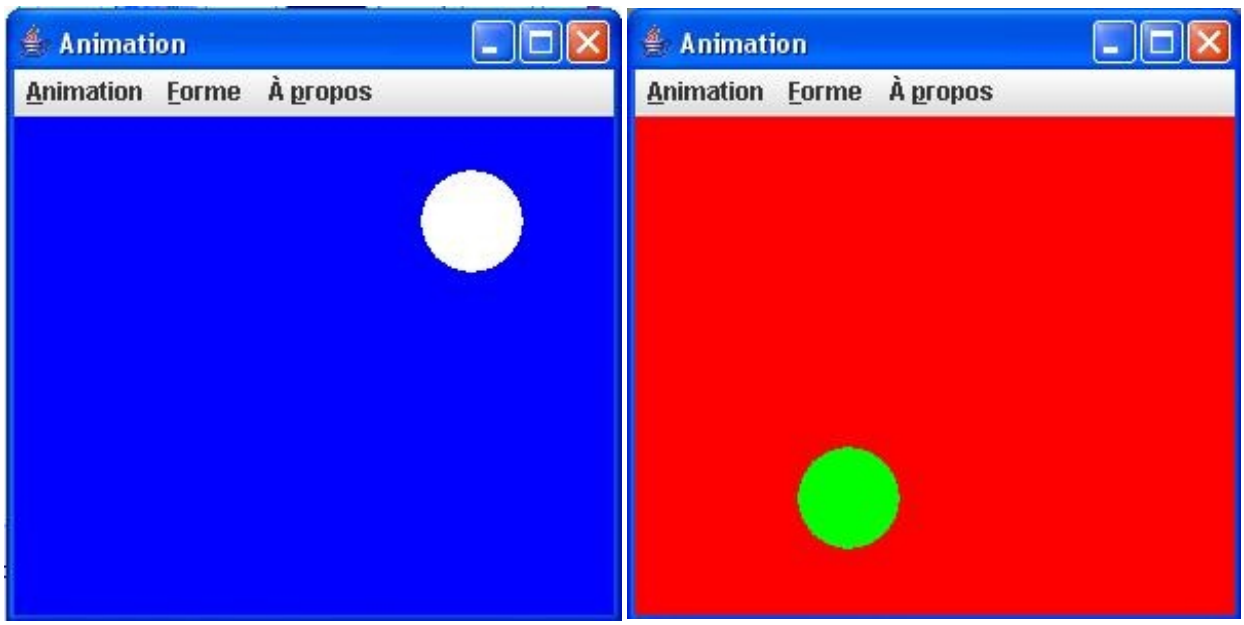
```
                this.setVisible(true);

        }



        private void initMenu(){
         //Menu animation
         //***************************

         //Ajout du listener pour lancer l'animation
         //ATTENTION LE LISTENER EST GLOBAL ! ! ! !
         //-----------------------------------------------
         lancer.addActionListener(startAnimation);
         //On attribut l'accélerateur c
         lancer.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_L,
                   KeyEvent.CTRL_MASK));
         animation.add(lancer);

         //Ajout du listener pour arrêter l'animation
         //LISTENER À CHANGER ICI AUSSI
         //-------------------------------------------
         arreter.addActionListener(stopAnimation);
         arreter.setEnabled(false);
         arreter.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_A,
                   KeyEvent.CTRL_DOWN_MASK +
    KeyEvent.SHIFT_DOWN_MASK));
         animation.add(arreter);

         animation.addSeparator();
         quitter.addActionListener(new ActionListener(){
          public void actionPerformed(ActionEvent event){
           System.exit(0);
          }
         });
         animation.add(quitter);

         //Menu forme

         bg.add(carre);
         bg.add(triangle);
         bg.add(rond);
         bg.add(etoile);

         //On crée un nouvel écouteur, inutile de créer 4 instances
    différentes
         FormeListener fl = new FormeListener();
         carre.addActionListener(fl);
         rond.addActionListener(fl);
         triangle.addActionListener(fl);
         etoile.addActionListener(fl);

         typeForme.add(rond);
         typeForme.add(carre);
         typeForme.add(triangle);
         typeForme.add(etoile);

         rond.setSelected(true);

         forme.add(typeForme);

         //Ajout du listener pour le morphing
         morph.addActionListener(new MorphListener());
         forme.add(morph);

         //menu à propos

         //Ajout de ce que doit faire le "?"
         aProposItem.addActionListener(new ActionListener(){
       public void actionPerformed(ActionEvent arg0) {
```

```
        JOptionPane jop = new JOptionPane();
        ImageIcon img = new ImageIcon("images/cysboy.gif");

        String mess = "Merci ! \n J'espère que vous vous amusez bien !
\n";
        mess += "Je crois qu'il est temps d'ajouter des accélérateurs et
des mnémoniques dans tout ça...\n";
        mess += "\n Allez, GO les ZérOs !";

        jop.showMessageDialog(null, mess, "À propos",
JOptionPane.INFORMATION_MESSAGE, img);

      }
        });
        aPropos.add(aProposItem);

        //Ajout des menus dans la barre de menus
        animation.setMnemonic('A');
        menuBar.add(animation);

        forme.setMnemonic('F');
        menuBar.add(forme);

        aPropos.setMnemonic('P');
        menuBar.add(aPropos);

        //Ajout de la barre de menus sur la fenêtre
        this.setJMenuBar(menuBar);
       }

    private void go(){
     x = pan.getPosX();
     y = pan.getPosY();
          while(this.animated){
           //System.out.println("OK");
           //Si le mode morphing est activé, on utilise la taille
actuelle de la forme
         if(pan.isMorph())
         {
          if(x < 1)backX = false;
                if(x > pan.getWidth() - pan.getDrawSize())backX = true;
                if(y < 1)backY = false;
                if(y > pan.getHeight() - pan.getDrawSize())backY =
true;
         }
         //Sinon, comme d'habitude
         else
         {
             if(x < 1)backX = false;
                if(x > pan.getWidth()-50)backX = true;
                if(y < 1)backY = false;
                if(y > pan.getHeight()-50)backY = true;
         }

            if(!backX)pan.setPosX(++x);
               else pan.setPosX(--x);
               if(!backY) pan.setPosY(++y);
               else pan.setPosY(--y);
               pan.repaint();

               try {
                       Thread.sleep(3);
               } catch (InterruptedException e) {
                       // TODO Auto-generated catch block
                       e.printStackTrace();
               }
          }
    }

    /**
```

```
        * Écouteur du menu Lancer
        * @author
        */
         public class StartAnimationListener implements ActionListener{

          public void actionPerformed(ActionEvent arg0) {

           JOptionPane jop = new JOptionPane();
           int option = jop.showConfirmDialog(null, "Voulez-vous lancer
        l'animation ?", "Lancement de l'animation",
        JOptionPane.YES_NO_OPTION, JOptionPane.PLAIN_MESSAGE);

           if(option == JOptionPane.OK_OPTION)
           {
            lancer.setEnabled(false);
            arreter.setEnabled(true);

            //ON AJOUTE L'INSTRUCTION POUR LE MENU CONTEXTUEL
            //***********************************************
            launch.setEnabled(false);
            stop.setEnabled(true);

            animated = true;
            t = new Thread(new PlayAnimation());
            t.start();
           }
          }
         }

         /**
        * Écouteur du menu Quitter
        * @author CHerby
        */
         class StopAnimationListener  implements ActionListener{

          public void actionPerformed(ActionEvent e) {

           JOptionPane jop = new JOptionPane();
           int option = jop.showConfirmDialog(null, "Voulez-vous arrêter
        l'animation ?", "Arrêt de l'animation",
        JOptionPane.YES_NO_CANCEL_OPTION, JOptionPane.QUESTION_MESSAGE);

           if(option != JOptionPane.NO_OPTION && option !=
        JOptionPane.CANCEL_OPTION && option != JOptionPane.CLOSED_OPTION)
           {
            animated = false;
            //On remplace nos boutons par nos JMenuItem
            lancer.setEnabled(true);
            arreter.setEnabled(false);

            //ON AJOUTE L'INSTRUCTION POUR LE MENU CONTEXTUEL
            //***********************************************
            launch.setEnabled(true);
            stop.setEnabled(false);
           }
          }
         }

         /**
        * Lance le thread.
        * @author CHerby
        */
         class PlayAnimation implements Runnable{
          public void run() {
           go();
          }
         }

         /**
        * Écoute les menus forme
```

```
 * @author CHerby
 */
 class FormeListener implements ActionListener{
  public void actionPerformed(ActionEvent e) {
   pan.setForme(((JRadioButtonMenuItem)e.getSource()).getText());
  }
 }

 /**
 * Écoute le menu Morphing
 * @author CHerby
 */
 class MorphListener implements ActionListener{
  public void actionPerformed(ActionEvent e) {
   //Si la case est cochée, activation du mode morphing
   if(morph.isSelected())pan.setMorph(true);
   //Sinon rien !
   else pan.setMorph(false);
  }
 }

 //************************************************************
 // CLASSE QUI VONT ÉCOUTER LE CHANGEMENT DE COULEURS
 //************************************************************

 /**
 * Écoute le changement de couleur du fond
 */
 class CouleurFondListener implements ActionListener{
  public void actionPerformed(ActionEvent e) {

   if(e.getSource() == vertBack)
    pan.setCouleurFond(Color.green);
   else if (e.getSource() == bleuBack)
    pan.setCouleurFond(Color.blue);
   else if(e.getSource() == rougeBack)
    pan.setCouleurFond(Color.red);
   else
    pan.setCouleurFond(Color.white);
  }
 }

 /**
 * Écoute le changement de couleur du fond
 */
 class CouleurFormeListener implements ActionListener{
  public void actionPerformed(ActionEvent e) {
   if(e.getSource() == vert)
    pan.setCouleurForme(Color.green);
   else if (e.getSource() == bleu)
    pan.setCouleurForme(Color.blue);
   else if(e.getSource() == rouge)
    pan.setCouleurForme(Color.red);
   else
    pan.setCouleurForme(Color.white);
  }
 }

}
```

Et voici quelques résultats obtenus :



Vous constatez que les menus et les menus contextuels peuvent être très utiles et très ergonomiques !
Ils sont relativement simples à utiliser et à implémenter.
Mais vous aurez sans doute remarqué qu'il y a toujours un clic superflu, qu'il s'agisse d'un menu ou d'un menu contextuel : il faut au moins un clic pour afficher le contenu d'un menu (sauf cas avec accélérateur).

Pour contrer ce genre de chose, il existe un outil très puissant : la barre d'outils !
Retour en haut

Les barres d'outils

Voici un exemple de barre d'outils :

Vous voyez en rose la barre de menus et en vert la barre d'outils.
En fait, pour faire simple, la barre d'outils sert à effectuer des actions présentes dans le menu mais sans avoir à chercher dans celui-ci ou à mémoriser le raccourci clavier (accélérateur). Elle permet donc de pouvoir faire des actions rapides.

Elle est généralement composée de multitudes de boutons sur lesquels est apposée une image symbolisant l'action que ce bouton peut effectuer.

Pour créer et utiliser une barre d'outils, nous allons utiliser l'objet **`JToolBar`**.
Je vous rassure tout de suite, cet objet fonctionne comme un menu classique, à la différence près que celui-ci prend des boutons (**`JButton`**).

Il n'y a pas d'endroit spécifique pour incorporer votre barre d'outils, il faudra l'expliciter lors de sa création !

Il nous faut, tout d'abord, des images à mettre sur nos boutons... J'en ai fait de toutes simples :

Vous devez avoir une idée de ce que nous allons mettre dans notre barre d'outils...

Concernant les actions à gérer, pour le lancement de l'animation et l'arrêt, il faudra penser à éditer le comportement des boutons de la barre d'outils, comme fait pour les deux actions du menu contextuel...

Concernant les boutons pour les formes, c'est un peu plus délicat.
Les autres composants pouvant éditer la forme de notre animation étaient des boutons radios. Or, ici, nous avons des boutons standards.
Hormis le fait qu'il va falloir une instance précise de la classe **`FormeListener`**, nous allons devoir modifier un peu son comportement...

Nous devrons savoir si l'action vient d'un bouton radio du menu ou d'un bouton de la barre d'outils...

Ça m'a l'air compliqué, tout ça !

En fait, non, et ceci grâce à l'objet **`ActionEvent`**. Nous allons pouvoir connaître le déclencheur de l'action ! Nous n'allons pas tester tous les boutons radio un par un... Le système utilisé jusque là, pour ces composants, était très bien.
Non, nous allons juste voir si celui qui a déclenché l'action est un **`JRadioButtonMenuItem`** ou non et, dans ce cas, nous testerons nos boutons...

Comment savoir ça ?

Rappelez-vous le chapitre sur la réflexivité !
La méthode `getSource()` nous retourne un objet, il est donc possible de connaître sa classe avec la méthode `getClass()` et donc, le nom de celle-ci avec la méthode `getName()` ...

Il va falloir qu'on pense à mettre à jour le bouton radio sélectionné dans le menu, alors ?

Tout à fait !
Je vois que vous commencez à penser événementiel !

Et là, pour votre plus grand plaisir, j'ai une astuce qui ne marche pas mal du tout : il suffit, lors du clic sur un bouton de la barre d'outils, de déclencher l'événement sur le bouton radio correspondant !

On serait curieux de savoir comment tu fais ça !

Dans la classe **AbstractButton**, dont héritent tous les boutons, il y a la méthode `doClick()` .
Cette méthode déclenche un événement identique à un vrai clic de souris sur le composant ! Donc, plutôt que de gérer la même façon de faire à deux endroits, nous allons rediriger l'action sur un autre composant...

Vous avez toutes les cartes pour faire votre barre d'outils.
N'oubliez pas que vous devez spécifier sa place sur le conteneur principal !

Bon, faites des tests, comparez, codez, effacez mais, au final, vous devriez avoir quelque chose comme ça :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

import javax.swing.BorderFactory;
import javax.swing.ButtonGroup;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JPopupMenu;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.JToolBar;
import javax.swing.KeyStroke;

public class Fenetre extends JFrame{

 private Panneau pan = new Panneau();
    private JPanel container = new JPanel();
    private int compteur = 0;
    private boolean animated = true;
    private boolean backX, backY;
    private int x,y ;
    private Thread t;



    //*******************************************************
    // La déclaration pour le menu de la JMenuBar
    //*******************************************************
    private JMenuBar menuBar = new JMenuBar();

    private JMenu animation = new JMenu("Animation"),
      forme = new JMenu("Forme"),
      typeForme = new JMenu("Type de forme"),
      aPropos = new JMenu("À propos");

    private JMenuItem  lancer = new JMenuItem("Lancer l'animation"),
         arreter = new JMenuItem("Arrêter l'animation"),
         quitter = new JMenuItem("Quitter"),
```

```
        aProposItem = new JMenuItem("?");

    private JCheckBoxMenuItem morph = new JCheckBoxMenuItem("Morphing");
    private JRadioButtonMenuItem  carre = new JRadioButtonMenuItem("Carré"),
            rond = new JRadioButtonMenuItem("Rond"),
            triangle = new JRadioButtonMenuItem("Triangle"),
            etoile = new JRadioButtonMenuItem("Etoile");

    private ButtonGroup bg = new ButtonGroup();

    //*********************************************************
    // La déclaration pour le menu contextuel
    //*********************************************************
    private JPopupMenu jpm = new JPopupMenu();
    private JMenu background = new JMenu("Couleur de fond");
    private JMenu couleur = new JMenu("Couleur de la forme");

    private JMenuItem launch = new JMenuItem("Lancer l'animation");
    private JMenuItem stop = new JMenuItem("Arrêter l'animation");
    private JMenuItem  rouge = new JMenuItem("Rouge"),
        bleu = new JMenuItem("Bleu"),
        vert = new JMenuItem("Vert"),
        blanc = new JMenuItem("Blanc"),
        rougeBack = new JMenuItem("Rouge"),
        bleuBack = new JMenuItem("Bleu"),
        vertBack = new JMenuItem("Vert"),
        blancBack = new JMenuItem("Blanc");

    //*********************************************************
    // ON CRÉE DES LISTENER GLOBAUX
    //*********************************************************
    private StopAnimationListener stopAnimation = new StopAnimationListener();
    private StartAnimationListener startAnimation = new StartAnimationListener
    private CouleurFondListener bgColor = new CouleurFondListener();
    private CouleurFormeListener frmColor = new CouleurFormeListener();
    //*********************************************************


    //Création de notre barre d'outils
    private JToolBar toolBar = new JToolBar();

    //Les boutons
    private JButton  play = new JButton(new ImageIcon("images/play.jpg")),
        cancel = new JButton(new ImageIcon("images/stop.jpg")),
        square = new JButton(new ImageIcon("images/carré.jpg")),
        tri = new JButton(new ImageIcon("images/triangle.jpg")),
        circle = new JButton(new ImageIcon("images/rond.jpg")),
        star = new JButton(new ImageIcon("images/étoile.jpg"));

    private Color fondBouton = Color.white;
    private FormeListener fListener = new FormeListener();

    public Fenetre(){

            this.setTitle("Animation");
            this.setSize(400, 300);
            this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            this.setLocationRelativeTo(null);

            container.setBackground(Color.white);
            container.setLayout(new BorderLayout());

            //On initialise le menu stop
            stop.setEnabled(false);
            //On affecte les écouteurs
            stop.addActionListener(stopAnimation);
      launch.addActionListener(startAnimation);

      //On affecte les écouteurs aux points de menus
      rouge.addActionListener(frmColor);
```

```
            bleu.addActionListener(frmColor);
            vert.addActionListener(frmColor);
            blanc.addActionListener(frmColor);

            rougeBack.addActionListener(bgColor);
            bleuBack.addActionListener(bgColor);
            vertBack.addActionListener(bgColor);
            blancBack.addActionListener(bgColor);
            //On crée et on passe l'écouteur pour afficher le menu contextuel
            //Création d'une implémentation de MouseAdapter
            //avec redéfinition de la méthode adéquate
                  pan.addMouseListener(new MouseAdapter(){
                   public void mouseReleased(MouseEvent event){
                    //Seulement s'il s'agit d'un clic droit
                    if(event.isPopupTrigger())
                    {
                     background.add(blancBack);
                     background.add(rougeBack);
                     background.add(bleuBack);
                     background.add(vertBack);

                     couleur.add(blanc);
                     couleur.add(rouge);
                     couleur.add(bleu);
                     couleur.add(vert);

                     jpm.add(launch);
                     jpm.add(stop);
                     jpm.add(couleur);
                     jpm.add(background);

                     //La méthode qui va afficher le menu
                     jpm.show(pan, event.getX(), event.getY());
                    }
                   }
                  });

                  container.add(pan, BorderLayout.CENTER);

                  this.setContentPane(container);
                  this.initMenu();
                  this.initToolBar();
                  this.setVisible(true);

        }


        private void initToolBar(){

         this.cancel.setEnabled(false);
         this.cancel.addActionListener(stopAnimation);
         this.cancel.setBackground(fondBouton);
         this.play.addActionListener(startAnimation);
         this.play.setBackground(fondBouton);

         this.toolBar.add(play);
         this.toolBar.add(cancel);
         this.toolBar.addSeparator();

         //Ajout des Listeners
         this.circle.addActionListener(fListener);
         this.circle.setBackground(fondBouton);
         this.toolBar.add(circle);

         this.square.addActionListener(fListener);
         this.square.setBackground(fondBouton);
         this.toolBar.add(square);

         this.tri.setBackground(fondBouton);
         this.tri.addActionListener(fListener);
```

```
        this.toolBar.add(tri);

        this.star.setBackground(fondBouton);
        this.star.addActionListener(fListener);
        this.toolBar.add(star);

        this.add(toolBar, BorderLayout.NORTH);

      }


      private void initMenu(){
        //Menu animation
        //***************************

        //Ajout du listener pour lancer l'animation
        //ATTENTION LE LISTENER EST GLOBAL ! ! ! !
        //-----------------------------------------------
        lancer.addActionListener(startAnimation);
        //On attribue l'accélerateur c
        lancer.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_L,
                  KeyEvent.CTRL_MASK));
        animation.add(lancer);

        //Ajout du listener pour arrêter l'animation
        //LISTENER À CHANGER ICI AUSSI
        //-------------------------------------------
        arreter.addActionListener(stopAnimation);
        arreter.setEnabled(false);
        arreter.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_A,
                  KeyEvent.CTRL_DOWN_MASK + KeyEvent.SHIFT_DOWN_MASK));
        animation.add(arreter);

        animation.addSeparator();
        quitter.addActionListener(new ActionListener(){
         public void actionPerformed(ActionEvent event){
          System.exit(0);
         }
        });
        animation.add(quitter);

        //Menu forme

        bg.add(carre);
        bg.add(triangle);
        bg.add(rond);
        bg.add(etoile);

        //On crée un nouvel écouteur, inutile de créer 4 instances différentes

        carre.addActionListener(fListener);
        rond.addActionListener(fListener);
        triangle.addActionListener(fListener);
        etoile.addActionListener(fListener);

        typeForme.add(rond);
        typeForme.add(carre);
        typeForme.add(triangle);
        typeForme.add(etoile);

        rond.setSelected(true);

        forme.add(typeForme);

        //Ajout du listener pour le morphing
        morph.addActionListener(new MorphListener());
        forme.add(morph);

        //menu à propos
```

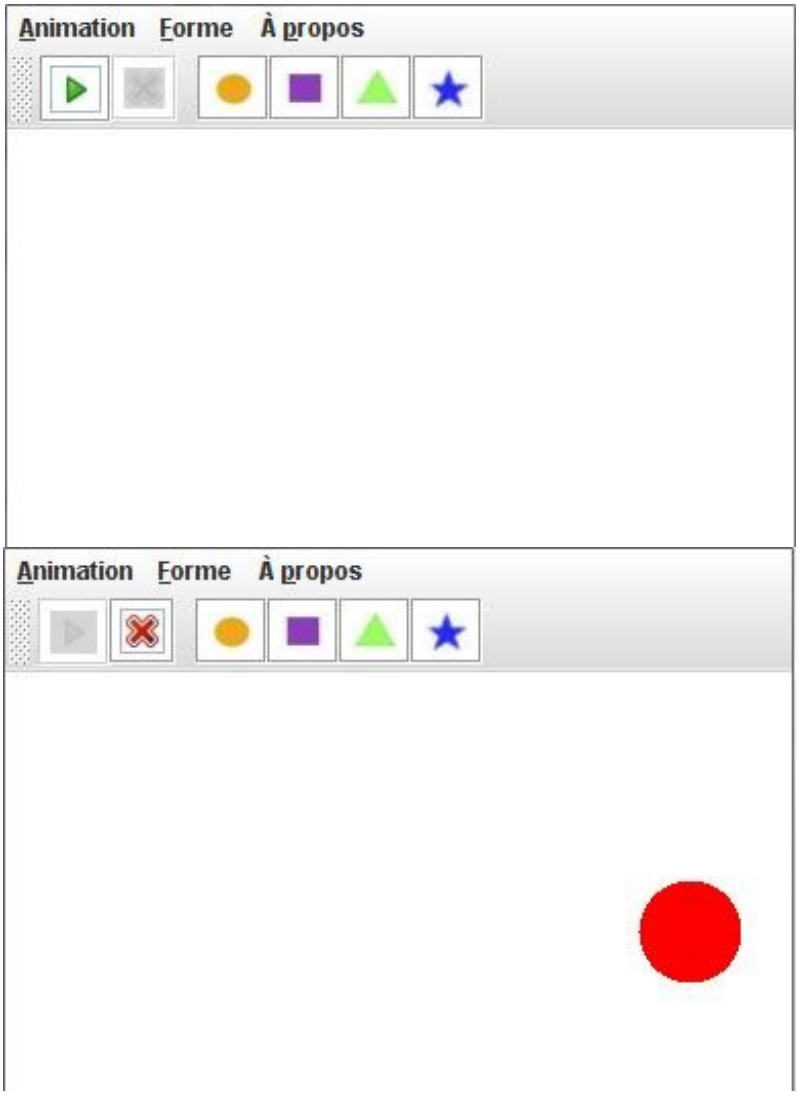
```
        //Ajout de ce que doit faire le "?"
        aProposItem.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent arg0) {
       JOptionPane jop = new JOptionPane();
       ImageIcon img = new ImageIcon("images/cysboy.gif");

       String mess = "Merci ! \n J'espère que vous vous amusez bien ! \n";
       mess += "Je crois qu'il est temps d'ajouter des accélérateurs et des
mnémoniques dans tout ça...\n";
       mess += "\n Allez, GO les ZérOs !";

       jop.showMessageDialog(null, mess, "À propos",
JOptionPane.INFORMATION_MESSAGE, img);

      }
        });
        aPropos.add(aProposItem);

        //Ajout des menus dans la barre de menus
        animation.setMnemonic('A');
        menuBar.add(animation);

        forme.setMnemonic('F');
        menuBar.add(forme);

        aPropos.setMnemonic('P');
        menuBar.add(aPropos);

        //Ajout de la barre de menus sur la fenêtre
        this.setJMenuBar(menuBar);
       }

    private void go(){
     x = pan.getPosX();
     y = pan.getPosY();
       while(this.animated){
        //System.out.println("OK");
        //Si le mode morphing est activé, on utilise la taille actuelle de la
forme
      if(pan.isMorph())
      {
       if(x < 1)backX = false;
            if(x > pan.getWidth() - pan.getDrawSize())backX = true;
            if(y < 1)backY = false;
            if(y > pan.getHeight() - pan.getDrawSize())backY = true;
      }
      //Sinon, comme d'habitude
      else
      {
          if(x < 1)backX = false;
            if(x > pan.getWidth()-50)backX = true;
            if(y < 1)backY = false;
            if(y > pan.getHeight()-50)backY = true;
      }

         if(!backX)pan.setPosX(++x);
           else pan.setPosX(--x);
           if(!backY) pan.setPosY(++y);
           else pan.setPosY(--y);
           pan.repaint();

           try {
                   Thread.sleep(3);
           } catch (InterruptedException e) {
                   // TODO Auto-generated catch block
                   e.printStackTrace();
           }
       }
    }
```

```
 /**
* Écouteur du menu Lancer
* @author CHerby
*/
 public class StartAnimationListener implements ActionListener{

  public void actionPerformed(ActionEvent arg0) {

   JOptionPane jop = new JOptionPane();
   int option = jop.showConfirmDialog(null, "Voulez-vous lancer l'animation ?",
"Lancement de l'animation", JOptionPane.YES_NO_OPTION,
JOptionPane.PLAIN_MESSAGE);

   if(option == JOptionPane.OK_OPTION)
   {
    lancer.setEnabled(false);
    arreter.setEnabled(true);

    //ON AJOUTE L'INSTRUCTION POUR LE MENU CONTEXTUEL
    //************************************************
    launch.setEnabled(false);
    stop.setEnabled(true);

    play.setEnabled(false);
    cancel.setEnabled(true);

    animated = true;
    t = new Thread(new PlayAnimation());
    t.start();
   }
  }
 }

 /**
* Écouteur du menu Quitter
* @author CHerby
*/
 class StopAnimationListener  implements ActionListener{

  public void actionPerformed(ActionEvent e) {


   JOptionPane jop = new JOptionPane();
   int option = jop.showConfirmDialog(null, "Voulez-vous arrêter l'animation ?"
"Arrêt de l'animation", JOptionPane.YES_NO_CANCEL_OPTION,
JOptionPane.QUESTION_MESSAGE);

   if(option != JOptionPane.NO_OPTION && option != JOptionPane.CANCEL_OPTION &&
option != JOptionPane.CLOSED_OPTION)
   {
    animated = false;
    //On remplace nos bouton par nous MenuItem
    lancer.setEnabled(true);
    arreter.setEnabled(false);

    //ON AJOUTE L'INSTRUCTION POUR LE MENU CONTEXTUEL
    //************************************************
    launch.setEnabled(true);
    stop.setEnabled(false);

    play.setEnabled(true);
    cancel.setEnabled(false);

   }
  }
 }

 /**
* Lance le thread.
* @author CHerby
```

```
*/
 class PlayAnimation implements Runnable{
  public void run() {
   go();
  }
 }

 /**
* Écoute les menus forme
* @author CHerby
*/
 class FormeListener implements ActionListener{
  public void actionPerformed(ActionEvent e) {

    //Si l'action vient d'un bouton radio du menu

if(e.getSource().getClass().getName().equals("javax.swing.JRadioButtonMenuItem'
     pan.setForme(((JRadioButtonMenuItem)e.getSource()).getText());
    else{
     if(e.getSource() == square){
      carre.doClick();
     }
     else if(e.getSource() == tri){
      triangle.doClick();
     }
     else if(e.getSource() == star){
      etoile.doClick();
     }
     else{
      rond.doClick();
     }
    }
   }
 }

 /**
* Écoute le menu Morphing
* @author CHerby
*/
 class MorphListener implements ActionListener{
  public void actionPerformed(ActionEvent e) {
   //Si la case est cochée, activation du mode morphing
   if(morph.isSelected())pan.setMorph(true);
   //Sinon rien !
   else pan.setMorph(false);
  }
 }

 //*************************************************************
 // CLASSES QUI VONT ECOUTER LE CHANGEMENT DE COULEURS
 //*************************************************************

 /**
* Écoute le changement de couleur du fond
*/
 class CouleurFondListener implements ActionListener{
  public void actionPerformed(ActionEvent e) {

   if(e.getSource() == vertBack)
    pan.setCouleurFond(Color.green);
   else if (e.getSource() == bleuBack)
    pan.setCouleurFond(Color.blue);
   else if(e.getSource() == rougeBack)
    pan.setCouleurFond(Color.red);
   else
    pan.setCouleurFond(Color.white);
  }
 }

 /**
```

```
        * Écoute le changement de couleur du fond
        */
         class CouleurFormeListener implements ActionListener{
          public void actionPerformed(ActionEvent e) {
           if(e.getSource() == vert)
            pan.setCouleurForme(Color.green);
           else if (e.getSource() == bleu)
            pan.setCouleurForme(Color.blue);
           else if(e.getSource() == rouge)
            pan.setCouleurForme(Color.red);
           else
            pan.setCouleurForme(Color.white);
          }
         }

        }
```

Vous devez obtenir une IHM ressemblant à ceci :



Elle n'est pas zolie, votre IHM, maintenant ?
Vous avez bien travaillé, surtout que je vous explique les grandes lignes mais vous avez une part de réflexion, à présent !
Eh oui, vous avez appris à penser en orienté objet, vous connaissez les grandes lignes de la programmation événementielle.
Maintenant, ce ne sont juste que des points techniques spécifiques à acquérir comme l'utilisation de certains objets !

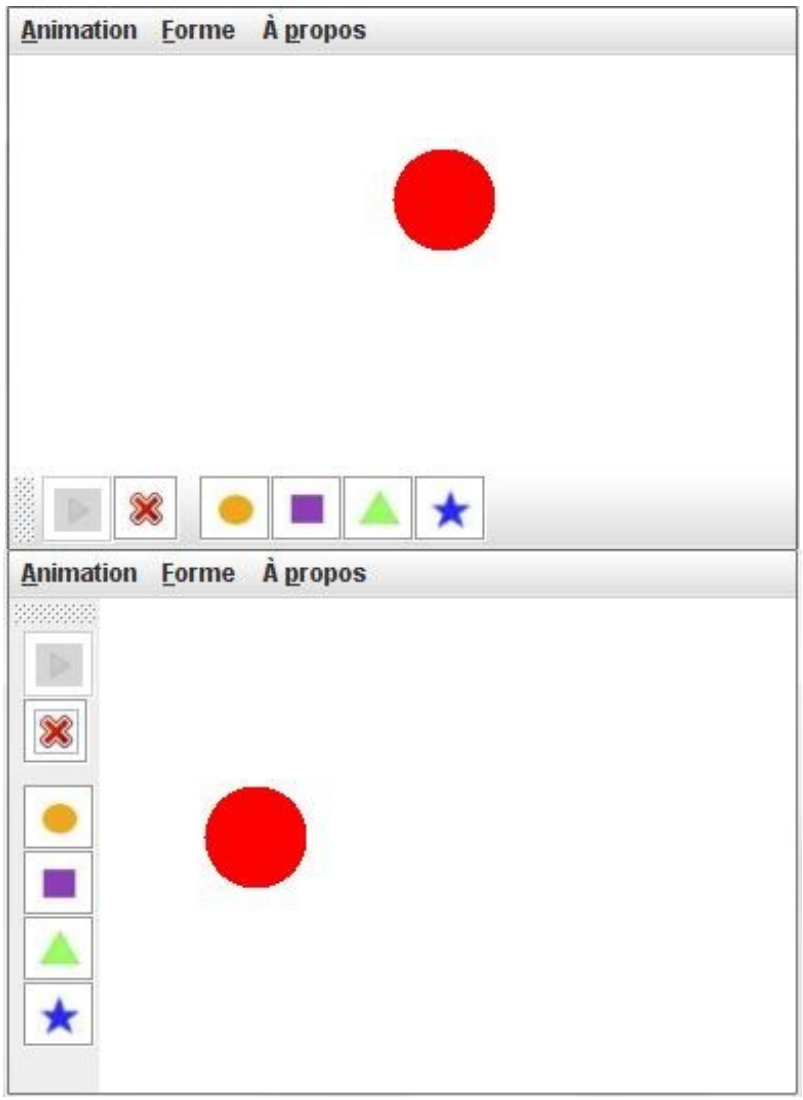
D'ailleurs, vous devez savoir une dernière chose : dans les barres d'outils comme celle-ci, vous pouvez mettre d'autres

composants que des boutons (combo...).

Pour ceux qui l'auraient remarqué, la barre d'outils est déplaçable ! Si vous cliquez sur cette zone :

si vous laissez le clic appuyé et faites glisser votre souris vers la droite, la gauche ou encore le bas, vous pourrez voir un carré se déplacer :

Et lorsque vous relâchez les clics, votre barre à changé d'emplacement :



Elles sont fortes ces barres d'outils, tout de même !
Par contre, vous devez savoir que vous pouvez utiliser autre chose qu'un composant sur une barre d'outils...

Retour en haut

Utiliser les actions abstraites

Nous avons vu, plus haut, comment centraliser les actions sur différents composants.
Vous devez savoir qu'il existe une classe abstraite qui permet de gérer ce genre de chose car elle peut s'adapter à beaucoup de composants (en général à ceux qui ne font qu'une action comme un bouton, une case à cocher et non une liste...).

Le but de cette classe est d'attribuer automatiquement une action à un ou plusieurs composants. Le principal avantage de cette façon de faire réside dans le fait que plusieurs composants travaillent avec une implémentation de la classe **AbstractAction**, mais le gros inconvénient réside dans le fait que vous devrez programmer une implémentation par action :

- une action pour la couleur de la forme en rouge ;
- une action pour la couleur de la forme en bleu ;
- une action pour la couleur de la forme en vert ;
- une action pour la couleur du fond en rouge ;
- une action pour la couleur du fond en bleu ;
- ...

Ceci peut être très lourd à faire, mais après, je vous laisse juges d'utiliser telle ou telle façon selon vos besoins !

Voici comment s'implémente cette classe :

Code : Java

```
public class Fenetre extends JFrame{

   //Nous pouvons utiliser les actions abstraites directement dans
un JButton
   private JButton bouton1 = new JButton(new
RougeAction("ActionRouge", new ImageIcon("images/rouge.jpg"));

    //Ou créer un instance concrète
    private RougeAction rAct = new RougeAction("ActionRouge", new
ImageIcon("images/rouge.jpg"));
    private JToolBar toolBar = new JToolBar();

    //...

    //Et le plus fort : UTILISER UNE INSTANCE D'ACTION DIRECTEMENT
DANS UNE BARRE D'OUTILS
    private void initToolBar(){
        toolBar.add(rAct);
    }



   //...

   class RougeAction extends AbstractAction{

       //Constructeur avec le nom uniquement
       public RougeAction(String name){super(name);}

       //Le constructeur prend le nom et une icône en paramètre
       public RougeAction(String name, ImageIcon){super(name, img);}

       public void actionPerformed(ActionEvent){
           //Vous connaissez la marche à suivre
       }
   }

}
```

Vous pouvez voir que cela peut être très pratique. De plus, si vous ajoutez une action sur une barre d'outils, celle-ci crée

automatiquement un bouton correspondant !

Ensuite, le choix d'utiliser les actions abstraites ou des implémentations de telle ou telle interface vous revient.

Nous pouvons d'ailleurs très bien appliquer ce principe au code de notre animation.

Voici ce que peut donner le code vu précédemment avec des implémentations de la classe **`AbstractAction`** :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

import javax.swing.AbstractAction;
import javax.swing.ButtonGroup;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JPopupMenu;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.JToolBar;
import javax.swing.KeyStroke;

public class ZFenetre extends JFrame{

 private Panneau pan = new Panneau();
    private JPanel container = new JPanel();
    private int compteur = 0;
    private boolean animated = true;
    private boolean backX, backY;
    private int x,y ;
    private Thread t;

   //Création des instances d'actions de forme
   private CarreAction cAction = new CarreAction("Carré", new
ImageIcon("images/carré.jpg"));
   private RondAction rAction = new RondAction("Rond",  new
ImageIcon("images/rond.jpg"));
   private TriangleAction tAction = new TriangleAction("Triangle",
new ImageIcon("images/triangle.jpg"));
   private EtoileAction eAction = new EtoileAction("Etoile",  new
ImageIcon("images/étoile.jpg"));

   //Création des instances d'actions de couleurs de forme
   private FormeRougeAction rfAction = new
FormeRougeAction("Rouge");
   private FormeBleuAction bfAction = new FormeBleuAction("Bleu");
   private FormeVertAction vfAction = new FormeVertAction("Vert");
   private FormeBlancAction wfAction = new
FormeBlancAction("Blanc");

   //Création des instances de couleurs de fond
   private FondRougeAction rFondAction = new
FondRougeAction("Rouge");
   private FondBleuAction bFondAction = new FondBleuAction("Bleu");
   private FondVertAction vFondAction = new FondVertAction("Vert");
```

```
    private FondVertAction vFondAction = new FondVertAction("Vert");
    private FondBlancAction wFondAction = new
FondBlancAction("Blanc");

    //Création des instances d'actions pour le lancement et l'arrêt
    private LancerAction lAction = new LancerAction("Lancer
l'animation",   new ImageIcon("images/play.jpg"));
    private ArretAction sAction = new ArretAction("Arrêter
l'animation",   new ImageIcon("images/stop.jpg"));

    //*********************************************************
     // La déclaration pour le menu de la JMenuBar
     //*********************************************************
     private JMenuBar menuBar = new JMenuBar();

     private JMenu animation = new JMenu("Animation"),
       forme = new JMenu("Forme"),
       typeForme = new JMenu("Type de forme"),
       aPropos = new JMenu("À propos");

     private JMenuItem  lancer = new JMenuItem(lAction),
          arreter = new JMenuItem(sAction),
          quitter = new JMenuItem("Quitter"),
          aProposItem = new JMenuItem("?");

     private JCheckBoxMenuItem morph = new
JCheckBoxMenuItem("Morphing");
     private JRadioButtonMenuItem  carre = new
JRadioButtonMenuItem(cAction),
             rond = new JRadioButtonMenuItem(rAction),
             triangle = new JRadioButtonMenuItem(tAction),
             etoile = new JRadioButtonMenuItem(eAction);

     private ButtonGroup bg = new ButtonGroup();

     //*********************************************************
     // La déclaration pour le menu contextuel
     //*********************************************************
     private JPopupMenu jpm = new JPopupMenu();
     private JMenu background = new JMenu("Couleur de fond");
     private JMenu couleur = new JMenu("Couleur de la forme");

     private JMenuItem launch = new JMenuItem(lAction);
     private JMenuItem stop = new JMenuItem(sAction);
     private JMenuItem  rouge = new JMenuItem(rfAction),
          bleu = new JMenuItem(bfAction),
          vert = new JMenuItem(vfAction),
          blanc = new JMenuItem(wfAction),
          rougeBack = new JMenuItem(rFondAction),
          bleuBack = new JMenuItem(bFondAction),
          vertBack = new JMenuItem(vFondAction),
          blancBack = new JMenuItem(wFondAction);



     //Création de notre barre d'outils
     private JToolBar toolBar = new JToolBar();

     //Les boutons
     private JButton  play = new JButton(lAction),
          cancel = new JButton(sAction),
          square = new JButton(cAction),
          tri = new JButton(tAction),
          circle = new JButton(rAction),
          star = new JButton(eAction);

     private Color fondBouton = Color.white;


     public ZFenetre(){
```

```
                this.setTitle("Animation");
                this.setSize(800, 600);
                this.setResizable(false);
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setLocationRelativeTo(null);

                container.setBackground(Color.white);
                container.setLayout(new BorderLayout());

                //On initialise le menu stop
                stop.setEnabled(false);

        //On crée et on passe l'écouteur pour afficher le menu
    contextuel
        //Création d'une implémentation de MouseAdapter
        //avec redéfinition de la méthode adéquate
                pan.addMouseListener(new MouseAdapter(){
                 public void mouseReleased(MouseEvent event){
                  //Seulement s'il s'agit d'un clic droit
                  if(event.isPopupTrigger())
                  {
                   background.add(blancBack);
                   background.add(rougeBack);
                   background.add(bleuBack);
                   background.add(vertBack);

                   couleur.add(blanc);
                   couleur.add(rouge);
                   couleur.add(bleu);
                   couleur.add(vert);

                   jpm.add(launch);
                   jpm.add(stop);
                   jpm.add(couleur);
                   jpm.add(background);

                   //La méthode qui va afficher le menu
                   jpm.show(pan, event.getX(), event.getY());
                  }
                 }
                });

                container.add(pan, BorderLayout.CENTER);

                this.setContentPane(container);
                this.initMenu();
                this.initToolBar();
                this.setVisible(true);

        }


        private void initToolBar(){

         this.cancel.setEnabled(false);
         this.cancel.setBackground(fondBouton);
         this.play.setBackground(fondBouton);
         this.toolBar.setFont(new Font("Courier", Font.PLAIN, 0));

         this.toolBar.add(play);
         this.toolBar.add(cancel);
         this.toolBar.addSeparator();

         this.toolBar.add(rAction);
         this.toolBar.add(cAction);
         this.toolBar.add(tAction);
         this.toolBar.add(eAction);
```

```
        this.add(toolBar, BorderLayout.NORTH);

    }


    private void initMenu(){
     //Menu animation
     //***************************

     //Ajout du listener pour lancer l'animation
     //ATTENTION LE LISTENER EST GLOBAL ! ! ! !
     //-----------------------------------------------
     //On attribue l'accélerateur c
     lancer.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_L,
                KeyEvent.CTRL_MASK));
     animation.add(lancer);

     //Ajout du listener pour arrêter l'animation
     //LISTENER À CHANGER ICI AUSSI
     //-----------------------------------------
     arreter.setEnabled(false);
     arreter.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_A,
                KeyEvent.CTRL_DOWN_MASK +
KeyEvent.SHIFT_DOWN_MASK));
     animation.add(arreter);

     animation.addSeparator();
     quitter.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent event){
       System.exit(0);
      }
     });
     animation.add(quitter);

     //Menu forme

     bg.add(carre);
     bg.add(triangle);
     bg.add(rond);
     bg.add(etoile);

     //On crée un nouvel écouteur, inutile de créer 4 instances
différentes

     typeForme.add(rond);
     typeForme.add(carre);
     typeForme.add(triangle);
     typeForme.add(etoile);

     rond.setSelected(true);

     forme.add(typeForme);

     //Ajout du listener pour le morphing
     morph.addActionListener(new MorphListener());
     forme.add(morph);

     //menu à propos

     //Ajout de ce que doit faire le "?"
     aProposItem.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent arg0) {
    JOptionPane jop = new JOptionPane();
    ImageIcon img = new ImageIcon("images/cysboy.gif");

    String mess = "Merci ! \n J'espère que vous vous amusez bien !
\n";
    mess += "Je crois qu'il est temps d'ajouter des accélérateurs et
des mnémoniques dans tout ça...\n";
    mess += "\n Allez, GO les ZérOs !";
```

```
        jop.showMessageDialog(null, mess, "À propos",
    JOptionPane.INFORMATION_MESSAGE, img);

       }
         });
         aPropos.add(aProposItem);

         //Ajout des menus dans la barre de menus
         animation.setMnemonic('A');
         menuBar.add(animation);

         forme.setMnemonic('F');
         menuBar.add(forme);

         aPropos.setMnemonic('P');
         menuBar.add(aPropos);

         //Ajout de la barre de menus sur la fenêtre
         this.setJMenuBar(menuBar);
        }

     private void go(){
      x = pan.getPosX();
      y = pan.getPosY();
            while(this.animated){
             //System.out.println("OK");
             //Si le mode morphing est activé, on utilise la taille
    actuelle de la forme
          if(pan.isMorph())
          {
           if(x < 1)backX = false;
                 if(x > pan.getWidth() - pan.getDrawSize())backX = true;
                 if(y < 1)backY = false;
                 if(y > pan.getHeight() - pan.getDrawSize())backY =
    true;
          }
          //Sinon, comme d'habitude
          else
          {
              if(x < 1)backX = false;
                 if(x > pan.getWidth()-50)backX = true;
                 if(y < 1)backY = false;
                 if(y > pan.getHeight()-50)backY = true;
          }

             if(!backX)pan.setPosX(++x);
                else pan.setPosX(--x);
                if(!backY) pan.setPosY(++y);
                else pan.setPosY(--y);
                pan.repaint();

                try {
                        Thread.sleep(3);
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
            }
     }

     /**
    * Lance le thread.
    * @author CHerby
    */
     class PlayAnimation implements Runnable{
      public void run() {
       go();
      }
     }
```

```
 /**
* Écoute le menu Morphing
* @author CHerby
*/
 class MorphListener implements ActionListener{
  public void actionPerformed(ActionEvent e) {
   //Si la case est cochée, activation du mode morphing
   if(morph.isSelected())pan.setMorph(true);
   //Sinon rien !
   else pan.setMorph(false);
  }
 }


 //************************************************************
 // LES CLASSES D'ACTIONS POUR LA FORME
 //************************************************************

 /**
* Classe gérant le changement de la forme en carré
* @author CHerby
*/
 class CarreAction extends AbstractAction{
  /**
* Le constructeur prend le nom de l'icône en paramètre
* @param name
* @param img
*/
  public CarreAction(String name, ImageIcon img){
   super(name, img);
  }
  /**
* Celui-ci ne prend que le nom
* @param name
*/
  public CarreAction(String name){
   super(name);
  }

  /**
* L'action effectuée
*/
  public void actionPerformed(ActionEvent e) {
   pan.setForme("CARRE");
   carre.setSelected(true);
  }
 }

 //************************************************************

 class RondAction extends AbstractAction{
  public RondAction(String name, ImageIcon img){
   super(name, img);
  }
  public void actionPerformed(ActionEvent arg0) {
   pan.setForme("ROND");
   rond.setSelected(true);
  }
 }

 //************************************************************

 class TriangleAction extends AbstractAction{
  public TriangleAction(String name, ImageIcon img){
   super(name, img);
  }
  public void actionPerformed(ActionEvent arg0) {
   pan.setForme("TRIANGLE");
   triangle.setSelected(true);
```

```
  }
}

//*************************************************************

class EtoileAction extends AbstractAction{
 public EtoileAction(String name, ImageIcon img){
  super(name, img);
 }
 public void actionPerformed(ActionEvent arg0) {
  pan.setForme("ETOILE");
  etoile.setSelected(true);
 }
}

//*************************************************************
// ACTIONS POUR LES COULEURS DE FOND
//*************************************************************


class FondRougeAction extends AbstractAction{
 public FondRougeAction(String name){
  super(name);
 }
 public void actionPerformed(ActionEvent arg0) {
  pan.setCouleurFond(Color.red);
 }
}

//*************************************************************

class FondBleuAction extends AbstractAction{
 public FondBleuAction(String name){
  super(name);
 }
 public void actionPerformed(ActionEvent arg0) {
  pan.setCouleurFond(Color.blue);
 }
}

//*************************************************************

class FondVertAction extends AbstractAction{
 public FondVertAction(String name){
  super(name);
 }
 public void actionPerformed(ActionEvent arg0) {
  pan.setCouleurFond(Color.green);
 }
}

//*************************************************************

class FondBlancAction extends AbstractAction{
 public FondBlancAction(String name){
  super(name);
 }
 public void actionPerformed(ActionEvent arg0) {
  pan.setCouleurFond(Color.white);
 }
}




//*************************************************************
// ACTIONS POUR LES COULEURS DE LA FORME
//*************************************************************
```

```
class FormeRougeAction extends AbstractAction{
 public FormeRougeAction(String name){
  super(name);
 }
 public void actionPerformed(ActionEvent arg0) {
  pan.setCouleurForme(Color.red);
 }
}

//*************************************************************

class FormeBleuAction extends AbstractAction{
 public FormeBleuAction(String name){
  super(name);
 }
 public void actionPerformed(ActionEvent arg0) {
  pan.setCouleurForme(Color.blue);
 }
}

//*************************************************************

class FormeVertAction extends AbstractAction{
 public FormeVertAction(String name){
  super(name);
 }
 public void actionPerformed(ActionEvent arg0) {
  pan.setCouleurForme(Color.green);
 }
}

//*************************************************************

class FormeBlancAction extends AbstractAction{
 public FormeBlancAction(String name){
  super(name);
 }
 public void actionPerformed(ActionEvent arg0) {
  pan.setCouleurForme(Color.white);
 }
}




//*************************************************************
// ACTIONS POUR LE LANCEMENT/ARRÊT
//*************************************************************


class ArretAction extends AbstractAction{
 public ArretAction(String name, ImageIcon img){
  super(name, img);
 }
 public void actionPerformed(ActionEvent arg0) {
  JOptionPane jop = new JOptionPane();
  int option = jop.showConfirmDialog(null, "Voulez-vous arrêter
l'animation ?", "Arrêt de l'animation",
JOptionPane.YES_NO_CANCEL_OPTION, JOptionPane.QUESTION_MESSAGE);

  if(option != JOptionPane.NO_OPTION && option !=
JOptionPane.CANCEL_OPTION && option != JOptionPane.CLOSED_OPTION)
  {
   animated = false;
   //On remplace nos boutons par nos MenuItem
   lancer.setEnabled(true);
   arreter.setEnabled(false);

   //ON AJOUTE L'INSTRUCTION POUR LE MENU CONTEXTUEL
   //*********************************************
```

```
            launch.setEnabled(true);
            stop.setEnabled(false);

            play.setEnabled(true);
            cancel.setEnabled(false);

          }
        }
      }

      //*************************************************************

      class LancerAction extends AbstractAction{
       public LancerAction(String name, ImageIcon img){
        super(name, img);
       }
       public void actionPerformed(ActionEvent arg0) {

        JOptionPane jop = new JOptionPane();
        int option = jop.showConfirmDialog(null, "Voulez-vous lancer
l'animation ?", "Lancement de l'animation",
JOptionPane.YES_NO_OPTION, JOptionPane.PLAIN_MESSAGE);

        if(option == JOptionPane.OK_OPTION)
        {
         lancer.setEnabled(false);
         arreter.setEnabled(true);

         //ON AJOUTE L'INSTRUCTION POUR LE MENU CONTEXTUEL
         //*********************************************
         launch.setEnabled(false);
         stop.setEnabled(true);

         play.setEnabled(false);
         cancel.setEnabled(true);

         animated = true;
         t = new Thread(new PlayAnimation());
         t.start();
        }
       }
      }

     }
```

Vous pouvez voir que le code s'est un peu alourdi...
Mais, comme je vous le disais, c'est une question de choix et de conception.

Bon, je crois qu'un topo et un QCM vous attendent...

Retour en haut

Ce qu'il faut retenir

- L'objet pour mettre une barre de menu sur vos IHM swing est un **`JMenuBar`**.
- Dans cet objet, vous pouvez mettre des objets **`JMenu`** afin de créer un menu déroulant.
- L'objet cité ci-dessus accepte des objets **`JMenu`**, **`JMenuItem`**, **`JCheckBoxMenuItem`** et **`JRadioButtonMenuItem`**.
- Afin d'interagir avec vos points de menu, vous pouvez utiliser une implémentation de l'interface **`ActionListener`**.
- Pour faciliter l'accès aux menus de la barre de menus, vous pouvez ajouter des **mnémoniques** à ceux-ci.
- L'ajout d'**accélérateurs** permet de déclencher des actions, le plus souvent, par des combinaisons de touches.
- Afin de pouvoir récupérer les codes des touches du clavier, vous devrez utiliser un objet **`KeyStroke`** ainsi qu'un objet **`KeyEvent`**.
- Les menus contextuels fonctionnent tout comme un menu normal, à la différence qu'il s'agit d'un objet **`JPopupMenu`**.
- Vous devez toutefois spécifier sur quel composant doit s'afficher le menu contextuel.
- La détection du clic droit se fait grâce à la méthode **`isPopupTrigger()`** de l'objet **`MouseEvent`**.
- L'ajout d'une barre d'outils nécessite l'utilisation de l'objet **`JToolBar`**.

Retour en haut

Vous avez encore appris beaucoup de choses dans ce chapitre...
Je vous rassure, vous n'êtes pas au bout de vos peines !

Il nous reste encore tant de choses à voir. D'ailleurs, vous vous demandez peut-être comment faire pour créer un menu "Enregistrer" ou encore "Enregistrer sous" ?

Avant de voir comment faire ceci, nous allons passer par un petit TP : **L'ardoise mazique !**

Retour en haut

TP : l'ardoise maZique

Nous voilà partis pour un nouveau TP.

Dans celui-ci, ce qui prime c'est surtout :

- d'utiliser les menus, les accélérateurs et mnémoniques ;
- d'ajouter une barre d'outils ;
- de créer des implémentations et savoir les utiliser sur plusieurs composants ;
- d'utiliser des classes anonymes ;
- ...

Je ne vous demande pas de faire un logiciel hyper-pointu !
Vous constaterez d'ailleurs qu'il peut y avoir des bugs d'affichage... Je ne vous dis pas lesquels, on ne sait jamais, si vous ne les trouvez pas...

Ce n'est pas tout ça mais... si on commençait ?

Cahier des charges

Voilà les recommandations.

Vous devez faire une sorte d'ardoise magique. En gros, celle-ci devra être composée d'un `JPanel` amélioré (ça sent l'héritage...) sur lequel vous pourrez tracer des choses en cliquant et en déplaçant la souris.

Vos tracés devront être effectués point par point, ceux-ci d'une taille que je vous laisse apprécier. Par contre, vous devrez pouvoir utiliser deux sortes de "*pinceaux*" :

- un carré ;
- un rond.

Vous aurez aussi la possibilité de changer la couleur de vos traits, les couleurs que j'ai choisies sont :

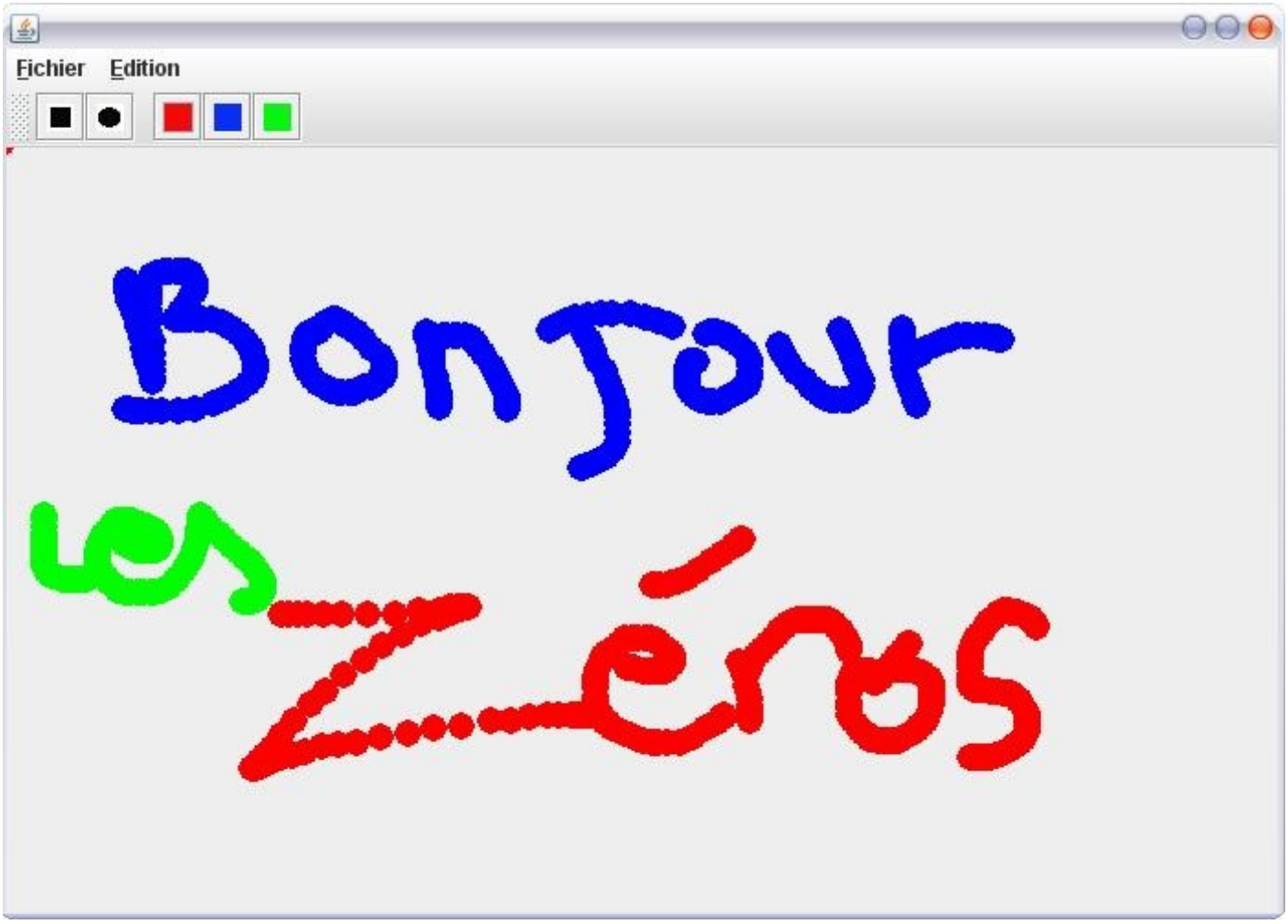
- le bleu ;
- le rouge ;
- le vert.

Il faut **OBLIGATOIREMENT** :

- un menu avec accélérateurs et mnémoniques ;
- une barre d'outils avec les formes et les couleurs ;
- un menu "**Quitter**" et un menu "**Effacer**" ;
- que les formes et les couleurs soient accessibles via le menu !

Voici ce que j'ai obtenu :

Et voilà ce que j'ai fait rien que pour vous :



Attention : vous allez utiliser la méthode `repaint()` de votre composant ; cependant, souvenez-vous que celle-ci est appelée automatiquement lors du redimensionnement de votre fenêtre, de la réduction, de l'agrandissement... Vous allez devoir gérer ce cas de figure ! Sinon, votre zone de dessin s'effacera à chaque redimensionnement !

Je vous conseille de créer une classe `Point` qui va contenir les informations relatives à un point tracé (couleur, taille, position...).
Il va falloir que vous gériez une collection de points (générique, la collection) dans votre classe dérivée de `JPanel` !

J'en ai presque trop dit...

Bon, je suis un ange aujourd'hui.
Voici les images que j'ai utilisées :

Je sais... Elles ne sont pas terribles.
Mais le but n'est pas là !

Avant de vous lancer dans votre code, vous devez savoir quelques choses...
Retour en haut

Prérequis

Afin que vous puissiez faire des tracés, vous allez devoir détecter le mouvement de votre souris.
Je ne vous ai pas encore parlé de ça auparavant, mais vous avez l'habitude d'utiliser des interfaces de gestion d'événements, maintenant...

Afin de détecter les mouvements de votre souris, vous allez devoir utiliser l'interface `MouseMotionListener` ; celle-ci contient deux méthodes :

- `mouseMoved(MouseEvent e)` : détecte le mouvement de la souris sur le composant ;
- `mouseDragged(MouseEvent e)` : idem que ci-dessus mais vous devrez avoir cliqué sur le composant et maintenir ce clic enfoncé pendant le mouvement (exactement ce qu'il vous faut).

Voilà : vous allez devoir créer une implémentation de cette interface pour réussir à dessiner sur votre conteneur !

Bon ! Ne vous hâtez pas trop, réfléchissez bien à ce dont vous avez besoin. Comment utiliser vos implémentations... Bref, surtout ne pas vous précipiter !
Un code bien réfléchi est un code rapidement opérationnel !

C'est à vous, maintenant...
À vos claviers.
Retour en haut

Correction

Roulement de tambour... C'est le moment de vérité !

J'ai mis des images sur mes boutons de barre d'outils...

Secret (cliquez pour afficher)

Point.java

Code : Java

```
import java.awt.Color;


public class Point {

 //Couleur du point
 private Color color = Color.red;
 //Taille
 private int size = 10;
 //position sur l'axe X : initialisé au dehors du conteneur
 private int x = -10;
 //Position sur l'axe Y : initialisé au dehors du conteneur
 private int y = -10;
 //Type de point
 private String type = "ROND";

 /**
* Constructeur par défaut
*/
 public Point(){}
 /**
* Constructeur avec paramètres
* @param x
* @param y
* @param size
* @param color
* @param type
*/
 public Point(int x, int y, int size, Color color, String type){
  this.size = size;
  this.x = x;
  this.y = y;
  this.color = color;
  this.type = type;
 }

 //*************************************
 // ACCESSEURS
 //*************************************
 public Color getColor() {
  return color;
 }
 public void setColor(Color color) {
  this.color = color;
 }
 public int getSize() {
  return size;
 }
 public void setSize(int size) {
  this.size = size;
 }
```

```
 public int getX() {
  return x;
 }
 public void setX(int x) {
  this.x = x;
 }
 public int getY() {
  return y;
 }
 public void setY(int y) {
  this.y = y;
 }
 public String getType() {
  return type;
 }
 public void setType(String type) {
  this.type = type;
 }
}
```

DrawPanel.java

Code : Java

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionListener;
import java.util.ArrayList;

import javax.swing.JPanel;


public class DrawPanel extends JPanel{

 //Couleur du pointeur
 private Color pointerColor = Color.red;
 //Forme du pointeur
 private String pointerType = "CIRCLE";
 //Position X du pointeur
 private int posX = -10, oldX = -10;
 //Position Y du pointeur
 private int posY = -10, oldY = -10;
 //pour savoir si on doit dessiner ou non
 private boolean erasing = true;
 //Taille du pointeur
 private int pointerSize = 15;
 //Collection de points !
 private ArrayList<Point> points = new ArrayList<Point>();

 /**
* Constructeur
*/
 public DrawPanel(){

  this.addMouseListener(new MouseAdapter(){
   public void mousePressed(MouseEvent e){
    points.add(new Point(e.getX() - (pointerSize / 2), e.getY() -
(pointerSize / 2), pointerSize, pointerColor, pointerType));
    repaint();
   }
  });

  this.addMouseMotionListener(new MouseMotionListener(){
```

```
        public void mouseDragged(MouseEvent e) {
         //On récupère les coordonnées de la souris
         //et on enlève la moitié de la taille du pointeur
         //pour centrer le tracé
         points.add(new Point(e.getX() - (pointerSize / 2), e.getY() -
(pointerSize / 2), pointerSize, pointerColor, pointerType));
         repaint();
        }

        public void mouseMoved(MouseEvent e) {}
       });

      }

      /**
     * Vous la connaissez maintenant, celle-là ;)
     */
      public void paintComponent(Graphics g) {

       g.setColor(Color.white);
       g.fillRect(0, 0, this.getWidth(), this.getHeight());

       //Si on doit effacer, on ne passe pas dans le else => pas de
dessin
       if(this.erasing){
        this.erasing = false;
       }
       else{
        //On parcourt notre collection de points
        for(Point p : this.points)
        {
         //On récupère la couleur
         g.setColor(p.getColor());

         //Selon le type de point
         if(p.getType().equals("SQUARE")){
          g.fillRect(p.getX(), p.getY(), p.getSize(), p.getSize());
         }
         else{
          g.fillOval(p.getX(), p.getY(), p.getSize(), p.getSize());
         }
        }
       }
      }

      /**
     * Efface le contenu
     */
      public void erase(){
       this.erasing = true;
       this.points = new ArrayList<Point>();
       repaint();
      }

      /**
     * Définit la couleur du pointeur
     * @param c
     */
      public void setPointerColor(Color c){
       this.pointerColor = c;
      }

      /**
     * Définit la forme du pointeur
     * @param str
     */
      public void setPointerType(String str){
       this.pointerType = str;
      }
```

```
}
```

Fenetre.java

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.awt.event.KeyEvent;

import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;
import javax.swing.JToolBar;
import javax.swing.KeyStroke;


public class Fenetre extends JFrame {

  //*************************************
  // LE MENU
  //*************************************
  private JMenuBar menuBar = new JMenuBar();
  JMenu  fichier = new JMenu("Fichier"),
    edition = new JMenu("Edition"),
    forme = new JMenu("Forme du pointeur"),
    couleur = new JMenu("Couleur du pointeur");

  JMenuItem  nouveau = new JMenuItem("Effacer"),
     quitter = new JMenuItem("Quitter"),
     rond = new JMenuItem("Rond"),
     carre = new JMenuItem("Carré"),
     bleu = new JMenuItem("Bleu"),
     rouge = new JMenuItem("Rouge"),
     vert = new JMenuItem("Vert");

  //*************************************
  // LA BARRE D'OUTILS
  //*************************************

  JToolBar toolBar = new JToolBar();

  JButton square = new JButton(new ImageIcon("images/carré.jpg")),
    circle = new JButton(new ImageIcon("images/rond.jpg")),
    red = new JButton(new ImageIcon("images/rouge.jpg")),
    green = new JButton(new ImageIcon("images/vert.jpg")),
    blue = new JButton(new ImageIcon("images/bleu.jpg"));

  //*************************************
  // LES ÉCOUTEURS
  //*************************************
  private FormeListener fListener = new FormeListener();
  private CouleurListener cListener = new CouleurListener();
```

```
    //Notre zone de dessin
    private DrawPanel drawPanel = new DrawPanel();

    public Fenetre(){
     this.setSize(700, 500);
     this.setLocationRelativeTo(null);
     this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

     //On initialise le menu
     this.initMenu();
     //Idem pour la barre d'outils
     this.initToolBar();
     //On positionne notre zone de dessin
     this.getContentPane().add(drawPanel, BorderLayout.CENTER);
     this.setVisible(true);
    }

    /**
   * Initialise le menu
   */
    private void initMenu(){
     nouveau.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent arg0) {
       drawPanel.erase();
      }
     });

     nouveau.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_N,
   KeyEvent.CTRL_DOWN_MASK));

     quitter.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent arg0) {
       System.exit(0);
      }
     });
     quitter.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_W,
   KeyEvent.CTRL_DOWN_MASK));

     fichier.add(nouveau);
     fichier.addSeparator();
     fichier.add(quitter);
     fichier.setMnemonic('F');

     carre.addActionListener(fListener);
     rond.addActionListener(fListener);
     forme.add(rond);
     forme.add(carre);

     rouge.addActionListener(cListener);
     vert.addActionListener(cListener);
     bleu.addActionListener(cListener);
     couleur.add(rouge);
     couleur.add(vert);
     couleur.add(bleu);

     edition.setMnemonic('E');
     edition.add(forme);
     edition.addSeparator();
     edition.add(couleur);

     menuBar.add(fichier);
     menuBar.add(edition);

     this.setJMenuBar(menuBar);
    }

    /**
   * Initialise l barre d'outils
   */
```

```
    private void initToolBar(){

     JPanel panneau = new JPanel();
     square.addActionListener(fListener);
     circle.addActionListener(fListener);
     red.addActionListener(cListener);
     green.addActionListener(cListener);
     blue.addActionListener(cListener);

     toolBar.add(square);
     toolBar.add(circle);

     toolBar.addSeparator();
     toolBar.add(red);
     toolBar.add(blue);
     toolBar.add(green);

     this.getContentPane().add(toolBar, BorderLayout.NORTH);
    }


    //ÉCOUTEUR POUR LE CHANGEMENT DE FORME
    //*************************************
    class FormeListener implements ActionListener{
     public void actionPerformed(ActionEvent e) {

   if(e.getSource().getClass().getName().equals("javax.swing.JMenuItem")){
       if(e.getSource()==carre)drawPanel.setPointerType("SQUARE");
       else drawPanel.setPointerType("CIRCLE");
      }
      else{
       if(e.getSource()==square)drawPanel.setPointerType("SQUARE");
       else drawPanel.setPointerType("CIRCLE");
      }
     }
    }


    //ÉCOUTEUR POUR LE CHANGEMENT DE COULEUR
    //***************************************
    class CouleurListener implements ActionListener{
     public void actionPerformed(ActionEvent e) {
      System.out.println(e.getSource().getClass().getName());

   if(e.getSource().getClass().getName().equals("javax.swing.JMenuItem")){
       System.out.println("OK !");
       if(e.getSource()==vert)drawPanel.setPointerColor(Color.green);
       else if(e.getSource()==bleu)drawPanel.setPointerColor(Color.blue);
       else drawPanel.setPointerColor(Color.red);
      }
      else{
       if(e.getSource()==green)drawPanel.setPointerColor(Color.green);
       else if(e.getSource()==blue)drawPanel.setPointerColor(Color.blue);
       else drawPanel.setPointerColor(Color.red);
      }
     }
    }


    public static void main(String[] args){
     Fenetre fen = new Fenetre();
    }

   }
```

Vous avez pu voir que c'est un code assez simple. Il n'y a rien de difficile et surtout, il a le mérite de vous faire travailler un peu tout ce que vous avez vu jusqu'ici...

Comme je vous l'ai dit dans l'introduction, ce code n'est pas parfait et il n'avait pas vocation à l'être... Je sais que vous êtes assez calés en Java pour avoir ajouté plus de fonctionnalités que moi sur ce TP. Mon code est une solution, ET NON PAS LA SOLUTION.

Retour en haut

Améliorations possibles

Voici ce que vous pouvez faire afin de rendre cette application plus attractive :

- permettre de changer la taille du pinceau ;
- avoir une plus grande palette de couleurs ;
- avoir des pinceaux supplémentaires ;
- créer une gomme ;
- ...

Vous voyez que ce ne sont pas les idées qui manquent...
Vous pouvez essayer de faire ce TP avec ces suppléments. Prenez ça comme une version plus difficile du TP, et encore...

Bon. Nous sommes à la fin de ce TP. Retour au tuto.
Retour en haut
Même s'il n'est pas parfait, ce TP était tout de même marrant à faire...

Bon, la récréation est finie.
Je sais que certains Zér0s ont une envie folle de créer des menus comme "**Enregistrer**" ou "**Enregistrer sous...**".

Mais, avant d'en arriver là, nous allons devoir aborder **Les flux d'entrée / de sortie** !
Retour en haut

Les flux d'entrées / sorties (1/2)

Je vous préviens tout de suite, ce chapitre sera certainement le plus difficile de tous. Déjà, parce que vous allez voir beaucoup de nouvelles choses, mais surtout parce que la notion de fichier et de son traitement n'est pas évidente au premier abord.

Ne vous inquiétez pas trop tout de même : je vais tenter de vous expliquer tout cela dans le détail, mais là, je ne vous cache pas qu'il faudra de la pratique pour tout assimiler...
J'espère que le fait de découper cette partie en deux vous facilitera le travail.

Alors, pourquoi utiliser les entrées / sorties ?
Tout simplement parce qu'il peut être utile de sauvegarder des données, traitées par votre programme, afin de les réutiliser !

Je tiens à signaler que je ne vais pas trop approfondir le sujet. Sinon il faudrait tout une partie rien que les différents type de fichier (.zip, .properties...). Et je tiens à dire aux puristes que je ne ferai que mettre en garde contre les exceptions que la lecture ou l'écriture dans des fichiers peut engendrer !

Les flux : qu'est-ce donc ?

Avant de nous lancer dans la programmation avec des fichiers, nous devons voir ce que sont les entrées / sorties, ou communément ce qui est appelé les flux d'entrée / sortie.

Une entrée / sortie en Java consiste en un échange de données entre le programme et une autre source, qui peut être :

- la mémoire ;
- un fichier ;
- le programme lui-même ;
- ...

Pour réaliser cela, Java emploie ce qu'on appelle un **stream** (qui signifie flux).
Celui-ci joue le rôle de médiateur entre la source des données et sa destination.
Nous allons voir que Java met à notre disposition toute une panoplie d'objets afin de pouvoir communiquer de la sorte. Tout comme l'objet `Scanner` que je ne vous présente plus maintenant, ces objets sont rangés dans un **package** : `java.io` (*io* signifie ici *in/out*, pour *entrée / sortie*). Il n'y a pas loin d'une cinquantaine de classes dans ce package... Heureusement, nous n'allons pas toutes les aborder.

Toute opération sur les entrées / sorties, en Java, doit suivre le schéma suivant :

- ouverture du flux ;
- lecture du flux ;
- fermeture du flux.

Je ne vous cache pas qu'il existe plusieurs objets différents qui ont tous des spécificités de travail avec les flux. Dans un souci de simplicité, nous n'aborderons que ceux qui traitent avec des fichiers (une liste des types d'objets est disponible en annexe).

Sachez aussi que Java a décomposé les objets traitant des flux en deux catégories :

- les objets travaillant avec des flux d'entrée (in), **lecture de flux** ;
- les objets travaillant avec des flux de sortie (out), **écriture de flux**.

Et comme je vous l'ai dit plus haut, il existe un objet Java pour chaque cas.
Par exemple, il existe :

- l'objet `FileInputStream` pour ouvrir un flux vers un fichier en lecture ;
- l'objet `FileOutputStream` pour ouvrir un flux vers un fichier en écriture ;
- ...

Ces objets peuvent prendre une chaîne de caractères, précisant le chemin et le nom du fichier à utiliser, en paramètres de leurs constructeurs ! Cependant, il peut être de bon ton d'utiliser un objet `File`.
Celui-ci permet de faire des tests sur le fichier ou de récupérer des informations le concernant.

C'est par là que nous commencerons.
Retour en haut

L'objet File

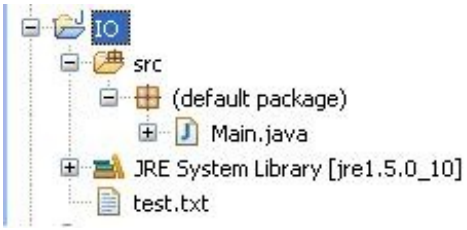
Avant de commencer, créez-vous un fichier, avec l'extension que vous voulez pour le moment, et enregistrez-le à la racine de votre projet Eclipse ! Personnellement, je me suis fait un fichier **test.txt** dont voici le contenu :

Code : Autre

```
Voici une ligne de test.
Voici une autre ligne de test.
Et comme je suis motivé, en voici une troisième !
```

Dans votre projet Eclipse, vous pouvez faire un **clic droit (sur le dossier de votre projet) / New / File**. Vous pouvez saisir le nom de votre fichier et ensuite y écrire !

Le nom du dossier contenant mon projet s'appelle "***IO***", mon fichier .txt est à cette adresse "***D:\Mes documents\Codage\SDZ\Java-SDZ\IO\test.txt***".
Voyez :



Maintenant, nous allons voir ce que sait faire l'objet `File`.
Vous allez voir que cet objet est très simple à utiliser et que ses méthodes sont très explicites.

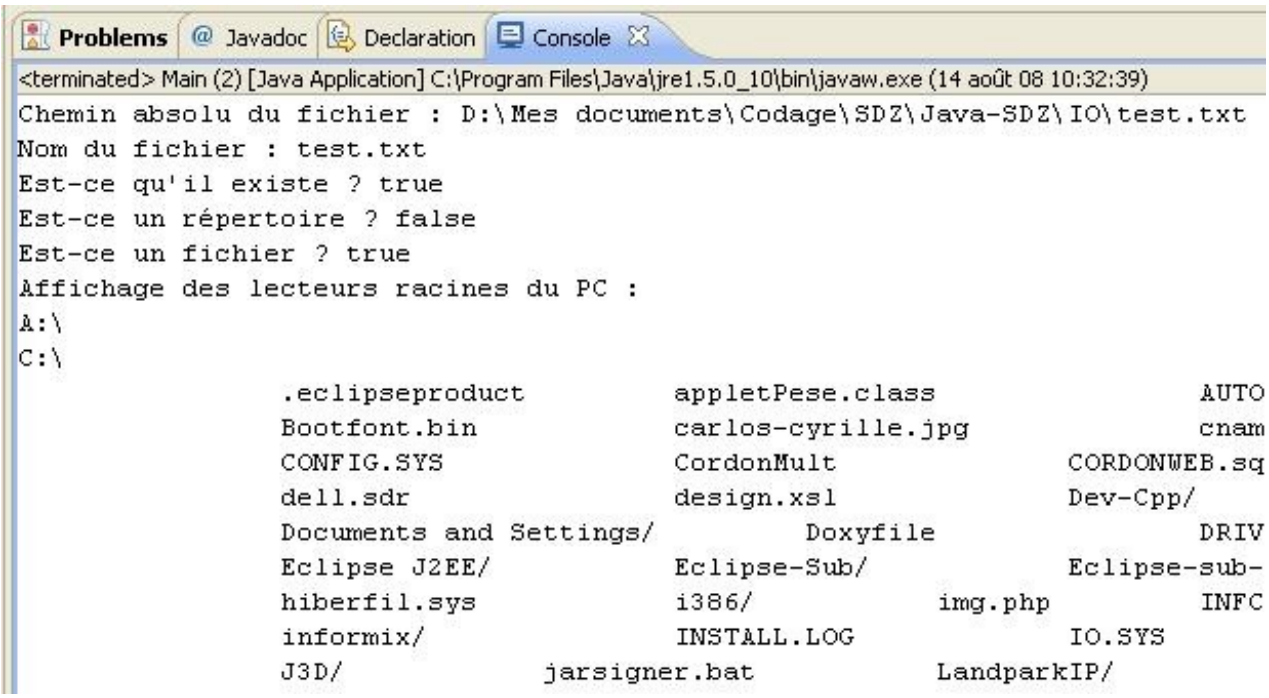
Pour commencer, nous allons retourner en mode console !

Code : Java

```
//Package à importer afin d'utiliser l'objet File
import java.io.File;

public class Main {
 public static void main(String[] args) {
  //Création de l'objet File
  File f = new File("test.txt");
  System.out.println("Chemin absolu du fichier : " +
f.getAbsolutePath());
  System.out.println("Nom du fichier : " + f.getName());
  System.out.println("Est-ce qu'il existe ? " + f.exists());
  System.out.println("Est-ce un répertoire ? " + f.isDirectory());
  System.out.println("Est-ce un fichier ? " + f.isFile());

  System.out.println("Affichage des lecteurs racines du PC : ");
  for(File file : f.listRoots())
  {
   System.out.println(file.getAbsolutePath());
   try {
    int i = 1;
    //On parcourt la liste des fichiers et répertoires
    for(File nom : file.listFiles()){
      //S'il s'agit d'un dossier, on ajoute un "/"
      System.out.print("\t\t" + ((nom.isDirectory()) ?
nom.getName()+"/" : nom.getName()));
      if((i%4) == 0){
       System.out.print("\n");
      }
```

```
                i++;
              }
              System.out.println("\n");
            } catch (NullPointerException e) {
             //L'instruction peut générer un NullPointerException s'il n'y a
             //pas de sous-fichier ! !
            }
          }
        }
      }
```

Le résultat est bluffant :



Vous conviendrez que les méthodes de cet objet peuvent s'avérer très utiles !
Nous venons d'en essayer quelques-unes et nous avons même listé les sous-fichiers et sous-dossiers de nos lecteurs racines.

Vous pouvez aussi effacer le fichier grâce la méthode `delete()` , créer des répertoires avec la méthode `mkdir()` (le nom passé à l'objet ne devra pas avoir de ".")...

Maintenant que vous en savez un peu plus sur cet objet, nous allons pouvoir commencer à travailler avec notre fichier créé précédemment !
Retour en haut

Les objets FileInputStream et FileOutputStream

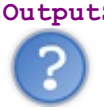
C'est par le biais de ces objets que nous allons pouvoir :

- lire dans un fichier ;
- écrire dans un fichier.

Ces classes héritent mutuellement des classes abstraites **InputStream** et **OutputStream**, présentes dans le package `java.io` .

Ce sont deux des super-classes présentes dans ce package et une grande partie des objets traitant des flux d'entrée / sortie héritent de ces objets.

Comme vous l'avez sans doute remarqué, il y a une hiérarchie de classe pour les traitements **in** et une autre pour les traitements **out**.
Ne vous y trompez pas, les classes héritant de **InputStream** sont destinées à la lecture et les classes héritant de **OutputStream** se chargent de l'écriture !



C'est bizarre, j'aurais dit le contraire...

Oui, comme beaucoup de gens au début. Mais c'est uniquement parce que vous situez les flux par rapport à vous, et non à votre programme !

Lorsque ce dernier va lire des informations dans un fichier, ce sont des informations qu'il reçoit, et par conséquent, elles s'apparentent par conséquent à une entrée : **in**.

Sachez tout de même que, lorsque vous tapez au clavier, cette action est considérée comme un flux d'entrée !

Au contraire, lorsque celui-ci va écrire dans un fichier (ou à l'écran, souvenez-vous de `System.out.println`), par exemple, il va faire sortir des informations : donc, pour lui, ce mouvement de données correspond à une sortie : **out**.

Nous allons enfin commencer à travailler avec notre fichier...
Le but est d'aller en lire le contenu et de le copier dans un autre, dont nous spécifierons le nom dans notre programme, par le biais d'un programme Java !

Ce code est assez compliqué... Donc accrochez-vous à vos claviers !

Code : Java

```
//Package à importer afin d'utiliser les objets
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class Main {
 public static void main(String[] args) {
  //Nous déclarons nos objets en dehors du bloc try/catch
        FileInputStream fis;
        FileOutputStream fos;

        try {
                //On instancie nos objets.
         //fis va lire le fichier et
         //fos va écrire dans le nouveau !
                fis = new FileInputStream(new File("test.txt"));
                fos = new FileOutputStream(new File("test2.txt"));

                //On créer un tableau de byte
                //pour dire de combien en combien on va lire les
données
                byte[] buf = new byte[8];
```

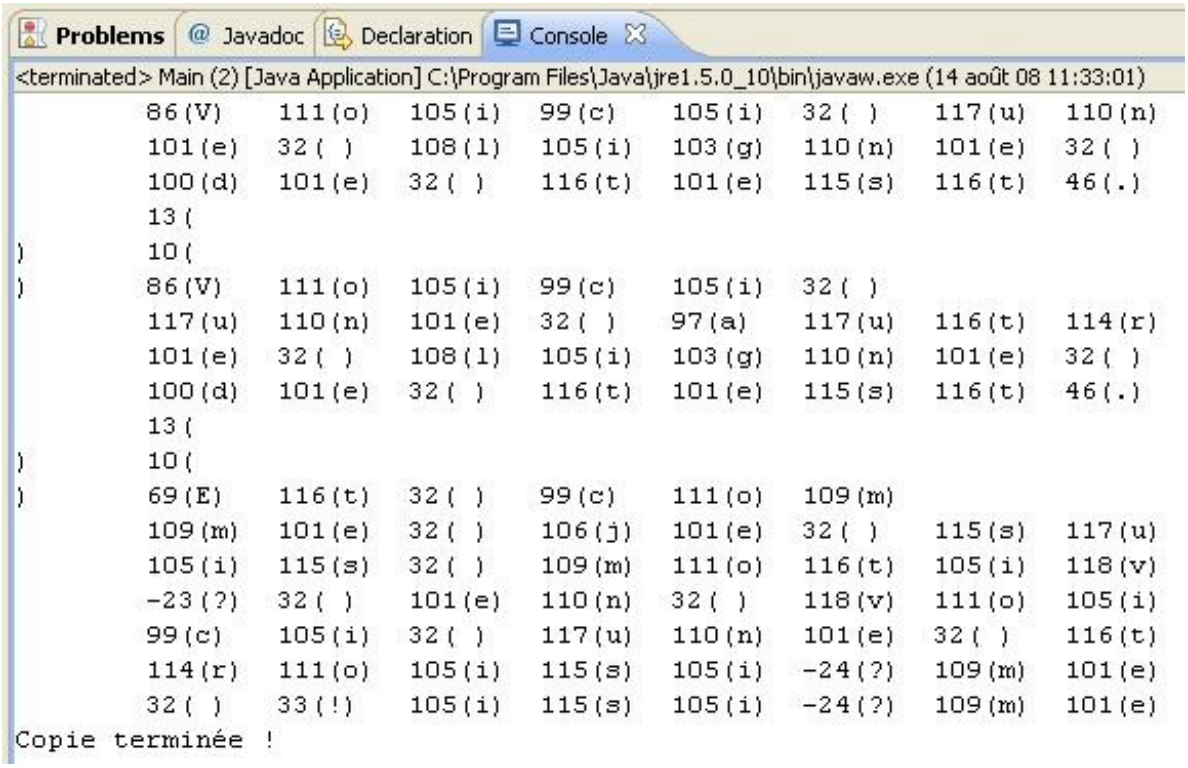
```
                //On crée une variable de type int
                //pour affecter le résultat de la lecture
                //retourne -1 quand c'est fini
                int n = 0;

                //Tant que l'affectation dans la variable est
possible, on boucle.
                //Lorsque le fichier est terminé, l'affectation
n'est plus possible !
                //Donc on sort de la boucle.
                while((n = fis.read(buf)) >= 0)
                {
                        //On écrit dans notre deuxième fichier
                  //avec l'objet adéquat
                        fos.write(buf);
                        //On affiche ce qu'a lu notre boucle
                        //au format byte et au format char
                        for(byte bit : buf)
                         System.out.print("\t" + bit + "(" +
(char)bit + ")");

                        System.out.println("");
                }

                //On ferme nos flux de données
                fis.close();
                fos.close();
                System.out.println("Copie terminée !");

          } catch (FileNotFoundException e) {
        //Cette exception est levée
        //si l'objet FileInputStream ne trouve aucun fichier
              e.printStackTrace();
          } catch (IOException e) {
           //Celle-ci se produit lors d'une erreur
           //d'écriture ou de lecture
              e.printStackTrace();
          }
    }
}
```

Pour que l'objet **`FileInputStream`** fonctionne, le fichier DOIT exister ! Sinon l'exception **`FileNotFoundException`** est levée.
Par contre, si vous ouvrez un flux en écriture (`FileOutputStream`) vers un fichier inexistant, celui-ci sera créé AUTOMATIQUEMENT !

Notez bien les imports pour pouvoir utiliser ces objets ! Mais comme vous le savez déjà, vous pouvez taper votre code et ensuite faire "***control + shift + O***" pour faire les imports automatiques.

À l'exécution de ce code, vous pouvez voir que le fichier "**test2.txt**" a bien été créé et qu'il contient exactement la même chose que "**test.txt**" !

De plus, j'ai ajouté dans la console les données que votre programme va utiliser (lecture et écriture).
Voici le résultat de ce code :



Les objets **`FileInputStream`** et **`FileOutputStream`** sont assez rudimentaires car ils travaillent avec un certain nombre d'octets à lire. Ceci explique pourquoi ma condition de boucle était si tordue...

Justement, tu ne pourrais pas nous expliquer un peu plus...

Mais bien sûr, je n'allais pas vous laisser comme ça...

Voici un rappel important.
Lorsque vous voyez des caractères dans un fichier ou sur votre écran, ceux-ci ne veulent pas dire grand-chose pour votre PC, car lui, il ne comprend que le binaire ! Vous savez, les suites de 0 et de 1...
Donc, afin de pouvoir afficher et travailler avec des caractères, un système d'encodage a été mis au point (qui a fort évolué d'ailleurs) !

Sachez qu'à chaque caractère que vous saisissez ou que vous lisez dans un fichier, correspond à un code binaire et ce code binaire correspond à un code décimal : voici la table de jeu de caractères.

Cependant, au début, seuls les caractères de a-z, de A-Z et les chiffres de 0-9 (127 premiers caractères de la table du lien ci-dessus) étaient codés (codage UNICODE 1) correspondant aux caractères se trouvant dans la langue anglaise mais, rapidement, ce codage s'est avéré trop limité pour des langues ayant des caractères accentués (français, espagnol...). Un jeu de codage étendu a été mis en place afin de pallier ce problème !

Chaque code binaire UNICODE 1 est codé sur 8 bits, soit 1 octet. Une variable de type **`byte`**, en Java, correspond en fait à 1 octet et non 1 bit !

Les objets que nous venons d'utiliser emploient la première version d'UNICODE 1 qui ne comprend pas les caractères accentués, c'est pour cela que les caractères de ce type, dans notre fichier, ont un code décimal négatif !
Lorsque nous définissons un tableau de **`byte`** à 8 entrées, cela signifie que nous allons lire 8 octets à la fois.

Vous pouvez voir qu'à chaque tour de boucle, notre tableau de **`byte`** contient huit valeurs correspondant chacune à un code décimal qui, lui, correspond à un caractère (valeur entre parenthèses à côté du code décimal).

Vous pouvez voir que les codes décimaux négatifs sont inconnus car ils sont représentés par "**?**" ; de plus, il y a des caractères invisibles dans notre fichier :

- les espaces : SP pour SPace, code décimal 32 ;
- les sauts de lignes : LF pour Line Feed, code décimal 13 ;
- les retours chariot : CR pour Cariage Return, code décimal 10.

Il en existe d'autres ; en fait, les 32 premiers caractères du tableau des caractères sont invisibles !

On comprend mieux...

Vous voyez que le traitement des flux suivent une logique et une syntaxe précises !
Lorsque nous avons copié notre fichier, nous avons récupéré un certain nombre d'octets dans un flux entrant que nous avons passé à un flux sortant.
À chaque tour de boucle, le flux entrant est lu en suivant tandis que le flux sortant, lui, écrit dans un fichier en suivant.

Cependant, il existe à présent des objets beaucoup plus faciles à utiliser, mais ceux-ci travaillent tout de même avec les deux objets que nous venons de voir !:D
Ces objets font aussi partie de la hiérarchie citée précédemment... Seulement, il existe une super-classe définissant ceux-ci.

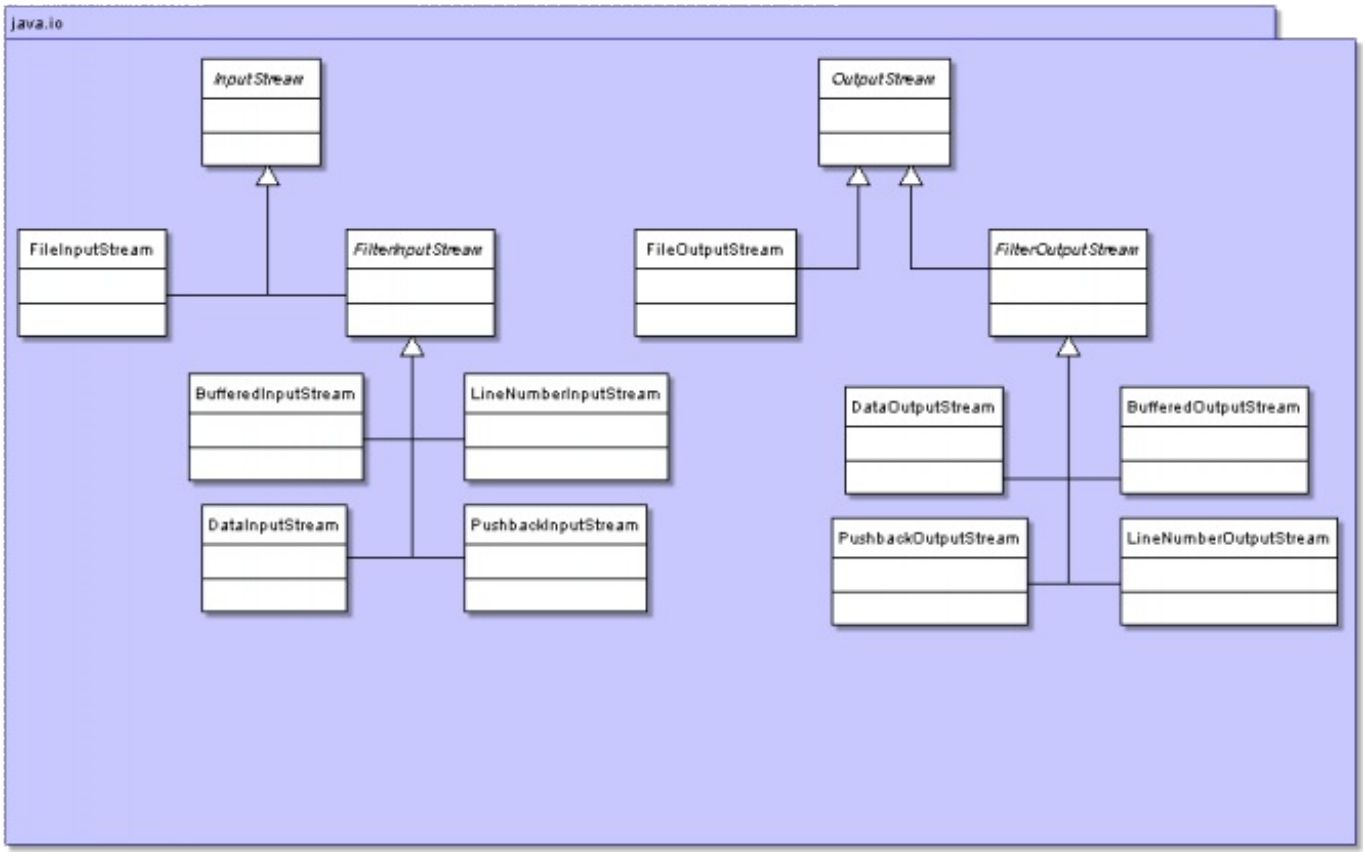
Continuons avec les flux filtrés.

Retour en haut

Les flux filtrés : les classes FilterInputStream et FilterOutputStream

Ces deux classes sont en fait des classes abstraites. Elles définissent un comportement global pour ces classes filles qui, elles, permettent d'ajouter des fonctionnalités aux flux d'entrée / sortie !

Voici un diagramme de classe représentant la hiérarchie de classes :



Vous pouvez voir qu'il y a quatre classes filles héritant de **FilterInputStream** (idem pour **FilterOutputStream**) :

- **DataInputStream** : offre la possibilité de lire directement des types primitifs (**double, char, int**), ceci grâce à des méthodes comme `readDouble()`, `readInt()` ...
- **BufferedInputStream** : cette classe permet d'avoir un tampon à disposition dans la lecture du flux. En gros, les données vont tout d'abord remplir le tampon et, dès que celui-ci est rempli, le programme a accès aux données ;
- **PushbackInputStream** : permet de remettre un octet déjà lu dans le flux entrant ;
- **LineNumberInputStream** : cette classe offre la possibilité de récupérer le numéro de ligne lue à un instant T.

Les classes dérivant de **FilterOutputStream** ont les mêmes fonctionnalités mais pour l'écriture !

Ces classes prennent une instance dérivant des classes **InputStream** (pour les classes héritant de **FilterInputStream**) ou de **OutputStream** (pour les classes héritant de **FilterOutputStream**)...

Vous pouvez cumuler les filtres, c'est-à-dire que, vu que ces classes acceptent une instance de leur super-classe, ils acceptent une instance de leur cousin !

Donc, vous pouvez avoir des choses du genre :

Code : Java

```
FileInputStream fis = new FileInputStream(new File("toto.txt"));
DataInputStream dis = new DataInputStream(fis);
BufferedInputStream bis = new BufferedInputStream(dis);
```

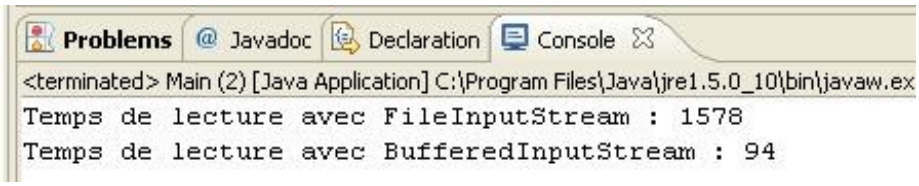
```
//Ou en condensé :
//BufferedInputStream bis = new BufferredInputStream(new
DataInputStream(FileInputStream(new File("toto.txt"))));
```

Afin de vous rendre compte des améliorations apportées par ces classes, nous allons lire un ÉNORME fichier texte (3.6Mo) de façon conventionnelle avec l'objet vu précédemment et avec un buffer !
Pour obtenir cet énorme fichier, rendez-vous à cette adresse. Faites un **clic droit / Enregistrer sous...** et remplacez le contenu de votre fichier test.txt par le contenu de ce fichier.
Maintenant, voici un code qui permet de tester le temps d'exécution de la lecture :

Code : Java

```
//Package à importer afin d'utiliser l'objet File
import java.io.BufferedInputStream;
import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class Main {
 public static void main(String[] args) {
  //Nous déclarons nos objets en dehors du bloc try/catch
        FileInputStream fis;
        BufferedInputStream bis;

        try {
         fis = new FileInputStream(new File("test.txt"));
                bis = new BufferedInputStream(new
FileInputStream(new File("test.txt")));
                byte[] buf = new byte[8];

                //On récupère le temps du système
                long startTime = System.currentTimeMillis();
                //Inutile de faire des traitements dans notre
boucle
                while(fis.read(buf) != -1);
                //On affiche le temps d'exécution
                System.out.println("Temps de lecture avec
FileInputStream : " + (System.currentTimeMillis() - startTime));

                //On réinitialise
                startTime = System.currentTimeMillis();
                //Inutile de faire des traitements dans notre
boucle
                while(bis.read(buf) != -1);
                //On réaffiche
                System.out.println("Temps de lecture avec
BufferedInputStream : " + (System.currentTimeMillis() - startTime));

                //On ferme nos flux de données
                fis.close();
                bis.close();

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Et le résultat est bluffant :



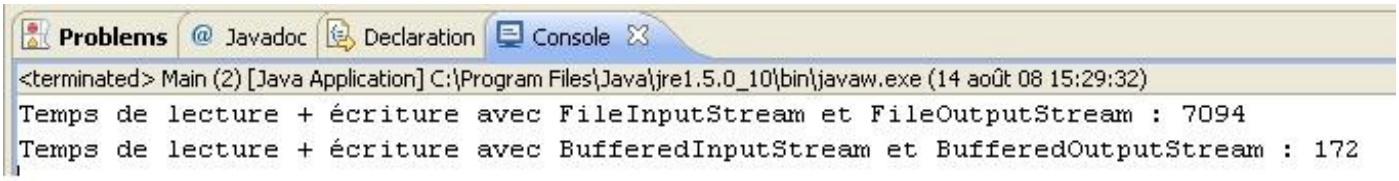
La différence de temps est ÉNORME ! 1.578 secondes pour la première méthode et 0.094 seconde pour la deuxième !
Vous conviendrez que l'utilisation d'un buffer permet une nette amélioration des performances de votre code !
Je vous conseille de faire le test pour l'écriture. D'ailleurs, nous allons le faire de ce pas :

Code : Java

```
//Package à importer afin d'utiliser l'objet File
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class Main {
 public static void main(String[] args) {
  //Nous déclarons nos objets en dehors du bloc try/catch
        FileInputStream fis;
        FileOutputStream fos;
        BufferedInputStream bis;
        BufferedOutputStream bos;

        try {
         fis = new FileInputStream(new File("test.txt"));
         fos = new FileOutputStream(new File("test2.txt"));
         bis = new BufferedInputStream(new FileInputStream(new
File("test.txt")));
                bos = new BufferedOutputStream(new
FileOutputStream(new File("test3.txt")));
                byte[] buf = new byte[8];

                //On récupère le temps du system
                long startTime = System.currentTimeMillis();

                while(fis.read(buf) != -1){
                 fos.write(buf);
                }
                //On affiche le temps d'exécution
                System.out.println("Temps de lecture + écriture avec
FileInputStream et FileOutputStream : " +
(System.currentTimeMillis() - startTime));

                //On réinitialise
                startTime = System.currentTimeMillis();

                while(bis.read(buf) != -1){
                 bos.write(buf);
                }
                //On réaffiche
                System.out.println("Temps de lecture + écriture avec
BufferedInputStream et BufferedOutputStream : " +
(System.currentTimeMillis() - startTime));

                //On ferme nos flux de données
                fis.close();
                bis.close();
```

```
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

Là, la différence de performance devient démentielle :



7 secondes en temps normal et 0.1 seconde avec un buffer !
Si avec ça vous n'êtes pas convaincus de l'utilité des buffers !

Je ne vais pas passer en revue tous les objets cités un peu plus haut, mais vu que vous risquez d'utiliser les objets **Data(Input/Output)Stream**, nous allons les aborder rapidement, puisqu'ils s'utilisent comme les objets **BufferedInputStream**. Je vous ai dit plus haut que ceux-ci ont des méthodes de lecture pour chaque type primitif : cependant, il faut que le fichier soit généré par le biais d'un **DataOutputStream** pour que les méthodes fonctionnent correctement.

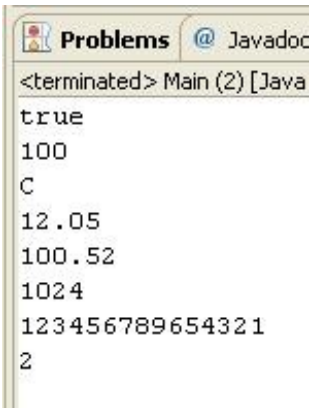
Nous allons donc créer un fichier de toute pièce pour le lire par la suite.

Code : Java

```
//Package à importer afin d'utiliser l'objet File
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class Main {
 public static void main(String[] args) {
  //Nous déclarons nos objets en dehors du bloc try/catch
        DataInputStream dis;
        DataOutputStream dos;
        try {
         dos = new DataOutputStream(
            new BufferedOutputStream(
              new FileOutputStream(
                new File("sdz.txt"))));

         //Nous allons écrire chaque primitif
         dos.writeBoolean(true);
         dos.writeByte(100);
         dos.writeChar('C');
         dos.writeDouble(12.05);
         dos.writeFloat(100.52f);
         dos.writeInt(1024);
         dos.writeLong(123456789654321L);
         dos.writeShort(2);
         dos.close();

         //On récupère maintenant les données !
         dis = new DataInputStream(
```

```
                new BufferedInputStream(
                  new FileInputStream(
                    new File("sdz.txt"))));

                 System.out.println(dis.readBoolean());
                 System.out.println(dis.readByte());
                 System.out.println(dis.readChar());
                 System.out.println(dis.readDouble());
                 System.out.println(dis.readFloat());
                 System.out.println(dis.readInt());
                 System.out.println(dis.readLong());
                 System.out.println(dis.readShort());

                } catch (FileNotFoundException e) {
                    e.printStackTrace();
                } catch (IOException e) {
                    e.printStackTrace();
                }
        }
       }
```

Et le résultat



Le code est simple et clair et concis...
Vous avez pu constater que ce type d'objet ne manque pas de fonctionnalités ! Mais vous savez, jusqu'ici, nous ne travaillions qu'avec des types primitifs, mais vous pouvez aussi travailler avec des objets !

Retour en haut

La sérialisation : les objets ObjectInputStream et ObjectOutputStream

La quoi ?

La sérialisation. C'est le nom que porte l'action de sauvegarder des objets !
Cela fait quelques temps déjà que vous utilisez des objets et, j'en suis sûr, vous auriez aimé que certains d'entre eux aient pu être réutilisés ensuite...

Le moment est venu de sauver vos objets d'une mort certaine !
Pour commencer, nous allons voir comment sérialiser un objet de notre composition.
Voici la classe avec laquelle nous allons travailler :

Code : Java

```
//package à importer
import java.io.Serializable;

public class Game implements Serializable{
 private String nom, style;
 private double prix;

 public Game(String nom, String style, double prix) {
  this.nom = nom;
  this.style = style;
  this.prix = prix;
 }

 public String toString(){
  return  "Nom du jeu : " + this.nom +
    "\nStyle de jeu : " + this.style +
    "\nPrix du jeu : " + this.prix +
    "\n";
 }
}
```

Qu'est-ce que c'est que cette interface ? Tu n'as même pas implémenté de méthode !

En fait, cette interface n'a pas de méthode à redéfinir !
L'interface **`Serializable`** est ce qu'on appelle une **interface marqueur !**
Juste en implémentant cette interface dans un objet, Java sait que cet objet peut être sérialisé et j'irais même plus loin : **si vous n'implémentez pas cette interface dans vos objets, ceux-ci ne pourront pas être sérialisés !**

Par contre, si une super-classe implémente l'interface **`Serializable`**, ses enfants seront considérés comme sérialisables !

Vous savez quasiment tout...
Maintenant, voilà comment vont se passer les choses :

- nous allons créer deux ou trois objets **`Game`** ;
- nous allons les sérialiser dans un fichier de notre choix ;
- nous allons ensuite les dé-sérialiser afin de pouvoir les réutiliser.

Nous allons donc pouvoir faire ceci grâce aux objets **`ObjectInputStream`** et **`ObjectOutputStream`** ?

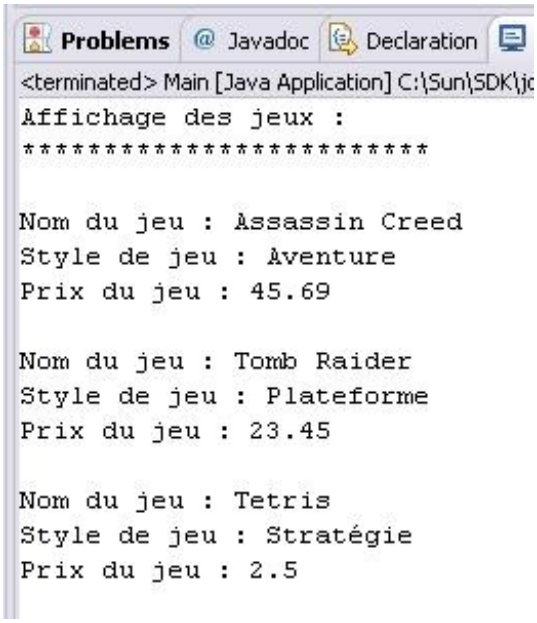
Tout à fait !
Vous avez sûrement déjà deviné comment se servir de ces objets, mais nous allons tout de même travailler sur un exemple.
D'ailleurs, le voici :

Code : Java

```
//Package à importer afin d'utiliser l'objet File
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class Main {
 public static void main(String[] args) {
  //Nous déclarons nos objets en dehors du bloc try/catch
        ObjectInputStream ois;
        ObjectOutputStream oos;
        try {
         oos = new ObjectOutputStream(
            new BufferedOutputStream(
              new FileOutputStream(
                new File("game.txt"))));

         //Nous allons écrire chaque objet Game dans le fichier
         oos.writeObject(new Game("Assassin Creed", "Aventure",
45.69));
         oos.writeObject(new Game("Tomb Raider", "Plateforme",
23.45));
         oos.writeObject(new Game("Tetris", "Stratégie", 2.50));
         //NE PAS OUBLIER DE FERMER LE FLUX ! ! !
         oos.close();

         //On récupère maintenant les données !
         ois = new ObjectInputStream(
        new BufferedInputStream(
          new FileInputStream(
            new File("game.txt"))));

         try {
     System.out.println("Affichage des jeux :");
     System.out.println("**************************\n");
           System.out.println(((Game)ois.readObject()).toString());
     System.out.println(((Game)ois.readObject()).toString());
     System.out.println(((Game)ois.readObject()).toString());
    } catch (ClassNotFoundException e) {
     e.printStackTrace();
    }
    ois.close();

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

La dé-sérialisation d'objet peut engendrer une **ClassNotFoundException** ! Pensez donc à capturer cette exception !

Et le résultat :



VICTOIRE !
Nos objets sont enregistrés et nous avons réussi à ré-utiliser ceux-ci après enregistrement !

Ce qu'il se passe est simple : les données de vos objets sont enregistrés dans le fichier ; mais que se passerait-il si notre objet **Game** avait un autre objet de votre composition en son sein ?

Voyons ça tout de suite. Créez la classe **Notice** comme suit :

Code : Java

```
public class Notice {
 private String langue ;
 public Notice(){
  this.langue = "Français";
 }
 public Notice(String lang){
  this.langue = lang;
 }
 public String toString() {
  return "\t Langue de la notice : " + this.langue + "\n";
 }
}
```

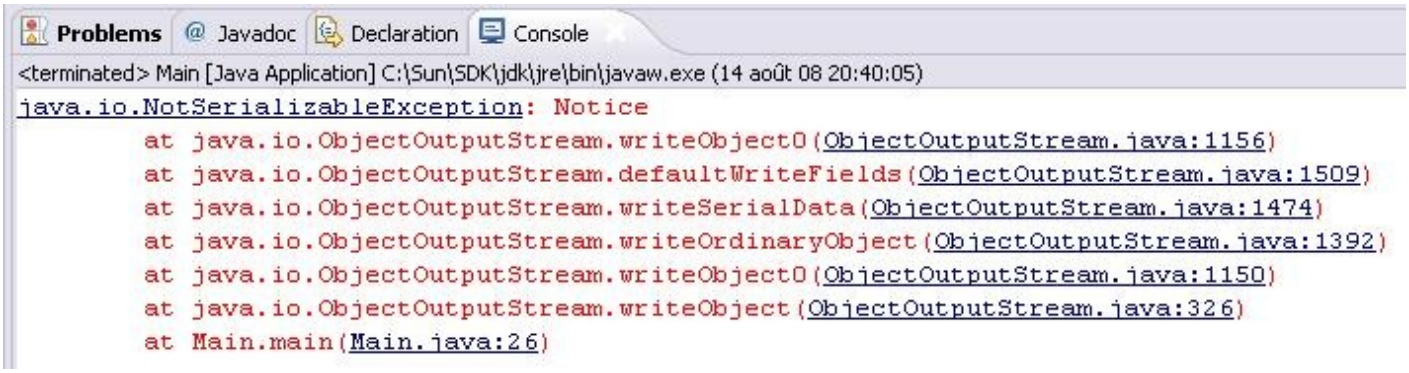
Nous allons maintenant implémenter une notice par défaut dans notre objet **Game**. Voici notre classe modifiée :

Code : Java

```
import java.io.Serializable;

public class Game implements Serializable{
 private String nom, style;
 private double prix;
 private Notice notice;

 public Game(String nom, String style, double prix) {
  this.nom = nom;
  this.style = style;
  this.prix = prix;
  this.notice = new Notice();
 }

 public String toString(){
```

```
        return  "Nom du jeu : " + this.nom +
          "\nStyle de jeu : " + this.style +
          "\nPrix du jeu : " + this.prix +
          "\n";
      }
    }
```

Réessayez votre code sauvegardant vos objets **Game**. Et voici le résultat :



Eh oui, votre code ne compile plus ! Ceci pour une bonne raison : votre objet **Notice** n'est pas sérialisable !
Une erreur de compilation est levée !
Maintenant, deux choix s'offrent à vous :

- soit vous faites en sorte de rendre votre objet sérialisable ;
- soit vous spécifiez dans votre classe **Game** que la variable "**notice**" n'a pas à être sérialisée !

Pour la première option, c'est simple, il suffit d'implémenter l'interface sérialisable dans notre classe **Notice**.
Par contre, on ne voit pas comment dire qu'un attribut n'a pas à être sérialisé...

Vous allez voir que c'est très simple, il suffit de déclarer votre variable : **transient**.

Comme ceci :

Code : Java

```
import java.io.Serializable;

public class Game implements Serializable{
 private String nom, style;
 private double prix;
 //Maintenant, cette variable ne sera pas sérialisée
 //Elle sera tout bonnement ignorée ! !
 private transient Notice notice;

 public Game(String nom, String style, double prix) {
  this.nom = nom;
  this.style = style;
  this.prix = prix;
  this.notice = new Notice();
 }

 public String toString(){
  return  "Nom du jeu : " + this.nom +
    "\nStyle de jeu : " + this.style +
    "\nPrix du jeu : " + this.prix +
    "\n";
 }
}
```

Vous aurez sans doute remarqué que nous n'utilisons pas la variable **notice** dans la méthode `toString()` de notre objet **Game**.
Si vous faites ceci, que vous sérialisez puis dé-sérialisez vos objets, vous aurez une **NullPointerException** à l'invocation de la dite méthode.
Eh oui ! L'objet **Notice** est ignoré : il n'existe donc pas !

C'est simple, n'est-ce pas ?
Pour ceux que la sérialisation XML intéresse, je vous propose d'aller faire un tour sur ce cours. Il est très bien fait et vous apprendrez beaucoup de choses.

Ce que je vous propose maintenant, c'est d'appliquer cela avec un cas concret : nous allons sauvegarder les zolis dessins que nous pouvons faire sur notre ardoise mazique...

Pour ceux qui n'auraient pas gardé le projet, je propose donc d'aller refaire un tour sur le TP en question.

Retour en haut

Cas pratique

Vu que, dans ce projet, les dessins se font grâce à l'objet **Point**, c'est celui-ci que nous allons sérialiser !
Je ne vais pas seulement vous proposer d'en sérialiser un seul, mais plusieurs et dans différents fichiers afin que vous puissiez réutiliser différents objets.

Nous allons avoir besoin d'un objet très pratique pour réussir ceci : un **JFileChooser** !
Cet objet affiche une mini-fenêtre demandant à l'utilisateur de choisir l'endroit où ouvrir ou sauvegarder des données ! Il possède aussi tout une batterie de méthodes très intéressante :

- `showOpenDialog()` : affiche une fenêtre d'ouverture de fichier ;
- `showSaveDialog()` : ouvre une fenêtre de sauvegarde ;
- `getSelectedFile()` : retourne le fichier sélectionné par l'utilisateur ;
- `getSelectedFiles()` : retourne les fichiers sélectionnés par l'utilisateur. S'il y a plusieurs sélections...
- ...

Grâce à tout ceci, vous pourrez enregistrer vos objets dans des fichiers.

Vous devez savoir que l'objet **JFileChooser** est très laxiste en matière d'extension de fichier.
Il ne connaît rien du contenu des fichiers qu'il traite... Donc, rien ne vous empêchera de sauvegarder vos fichiers avec l'extension **.exe** ou encore **.java** et de les relire !

Afin d'avoir une cohérence dans nos fichiers de sauvegarde, nous allons spécifier à notre **JFileChooser** qu'il ne devra sauvegarder et lire que des fichiers ayant une certaine extension !
Pour faire ceci, nous devons créer une classe à part entière héritant de la classe **FileFilter** car celle-ci est une classe abstraite.

Cette classe permet de redéfinir deux méthodes :

- `accept(File file)` : retourne vrai si le fichier est accepté ;
- `getDescription()` : retourne la description de l'extension de fichier.

Voici donc la classe ZFileFilter que nous allons utiliser afin de créer des filtres pour nos fichiers :

Code : Java

```
import java.io.File;
import javax.swing.filechooser.FileFilter;

public class ZFileFilter extends FileFilter {

 private String extension = ".sdz", description = "Fichier Ardoise
Mazique";

 public ZFileFilter(String ext, String descrip){
  this.extension = ext;
  this.description = descrip;
 }

 public boolean accept(File file){
  return (file.isDirectory() ||
file.getName().endsWith(this.extension));
 }

 public String getDescription(){
  return this.extension + " - " + this.description;
 }
}
```

Maintenant nous pouvons créer des filtres !
Voici les codes source que vous devriez parfaitement comprendre vu qu'il n'y a pas beaucoup de nouveautés !

Secret (cliquez pour afficher)

Point.java

Code : Java

```
import java.awt.Color;
import java.io.Serializable;


public class Point implements Serializable{

 //Couleur du point
 private Color color = Color.red;
 //Taille
 private int size = 10;
 //position sur l'axe X : initialisé au dehors du conteneur
 private int x = -10;
 //Position sur l'axe Y : initialisé au dehors du conteneur
 private int y = -10;
 //Type de point
 private String type = "ROND";

 /**
* Constructeur par défaut
*/
 public Point(){}
 /**
* Constructeur avec paramètre
* @param x
* @param y
* @param size
* @param color
* @param type
*/
 public Point(int x, int y, int size, Color color, String type){
  this.size = size;
  this.x = x;
  this.y = y;
  this.color = color;
  this.type = type;
 }

 //****************************************
 // ACCESSEURS
 //****************************************
 public Color getColor() {
  return color;
 }
 public void setColor(Color color) {
  this.color = color;
 }
 public int getSize() {
  return size;
 }
 public void setSize(int size) {
  this.size = size;
 }
 public int getX() {
  return x;
 }
 public void setX(int x) {
  this.x = x;
 }
 public int getY() {
  return y;
 }
```

```
    public void setY(int y) {
     this.y = y;
    }
    public String getType() {
     return type;
    }
    public void setType(String type) {
     this.type = type;
    }
}
```

DrawPanel.java

J'ai rajouté des accesseurs pour la collection de points à sauvegarder !

Code : Java

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionListener;
import java.util.ArrayList;

import javax.swing.JPanel;


public class DrawPanel extends JPanel{

 //Couleur du pointeur
 private Color pointerColor = Color.red;
 //Forme du pointeur
 private String pointerType = "CIRCLE";
 //Position X du pointeur
 private int posX = -10, oldX = -10;
 //Position Y du pointeur
 private int posY = -10, oldY = -10;
 //pour savoir si on doit dessiner ou non
 private boolean erasing = true;
 //Taille du pointeur
 private int pointerSize = 15;
 //Collection de points !
 private ArrayList<Point> points = new ArrayList<Point>();

 /**
* Constructeur
*/
 public DrawPanel(){

  this.addMouseListener(new MouseAdapter(){
   public void mousePressed(MouseEvent e){
    points.add(new Point(e.getX() - (pointerSize / 2), e.getY() -
(pointerSize / 2), pointerSize, pointerColor, pointerType));
    repaint();
   }
  });

  this.addMouseMotionListener(new MouseMotionListener(){
   public void mouseDragged(MouseEvent e) {
    //On récupère les coordonnées de la souris
    //et on enlève la moitié de la taille du pointeur
    //pour centrer le tracé
    points.add(new Point(e.getX() - (pointerSize / 2), e.getY() -
(pointerSize / 2), pointerSize, pointerColor, pointerType));
    repaint();
   }
```

```
        public void mouseMoved(MouseEvent e) {}
      });

    }

    /**
   * Vous la connaissez maintenant, celle-là ;)
   */
    public void paintComponent(Graphics g) {

      g.setColor(Color.white);
      g.fillRect(0, 0, this.getWidth(), this.getHeight());

      //Si on doit effacer, on ne passe pas dans le else => pas de
   dessin
      if(this.erasing){
       this.erasing = false;
      }
      else{
       //On parcourt notre collection de points
       for(Point p : this.points)
       {
        //On récupère la couleur
        g.setColor(p.getColor());

        //Selon le type de point
        if(p.getType().equals("SQUARE")){
         g.fillRect(p.getX(), p.getY(), p.getSize(), p.getSize());
        }
        else{
         g.fillOval(p.getX(), p.getY(), p.getSize(), p.getSize());
        }
       }
      }
    }

    /**
   * Efface le contenu
   */
    public void erase(){
      this.erasing = true;
      this.points = new ArrayList<Point>();
      repaint();
    }

    /**
   * Définit la couleur du pointeur
   * @param c
   */
    public void setPointerColor(Color c){
      this.pointerColor = c;
    }

    /**
   * Définit la forme du pointeur
   * @param str
   */
    public void setPointerType(String str){
      this.pointerType = str;
    }

    public ArrayList<Point> getPoints() {
      return points;
    }

    public void setPoints(ArrayList<Point> points) {
      this.points = points;
      repaint();
    }
```

```
}
```

Fenetre.java

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;

import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JToolBar;
import javax.swing.KeyStroke;


public class Fenetre extends JFrame {

 //*************************************
 // LE MENU
 //*************************************
 private JMenuBar menuBar = new JMenuBar();
 JMenu  fichier = new JMenu("Fichier"),
   edition = new JMenu("Edition"),
   forme = new JMenu("Forme du pointeur"),
   couleur = new JMenu("Couleur du pointeur");

 JMenuItem  nouveau = new JMenuItem("Effacer"),

    //Nos nouveaux points de menus
    enregistrer = new JMenuItem("Enregistrer"),
    enregistrerSous = new JMenuItem("Enregistrer Sous"),
    ouvrir = new JMenuItem("Ouvrir"),

    quitter = new JMenuItem("Quitter"),
    rond = new JMenuItem("Rond"),
    carre = new JMenuItem("Carré"),
    bleu = new JMenuItem("Bleu"),
    rouge = new JMenuItem("Rouge"),
    vert = new JMenuItem("Vert");

 //*************************************
 // LA BARRE D OUTILS
 //*************************************

 JToolBar toolBar = new JToolBar();
```

```
        JButton square = new JButton(new ImageIcon("images/carré.jpg")),
          circle = new JButton(new ImageIcon("images/rond.jpg")),
          red = new JButton(new ImageIcon("images/rouge.jpg")),
          green = new JButton(new ImageIcon("images/vert.jpg")),
          blue = new JButton(new ImageIcon("images/bleu.jpg"));

        //**************************************
        // LES ÉCOUTEURS
        //**************************************
        private FormeListener fListener = new FormeListener();
        private CouleurListener cListener = new CouleurListener();

        //Le JFileChooser
        //ici j'ai créé un répertoire "backup" à la racine de mon projet
        //En le spécifiant dans le constructeur, mon chooser s'ouvrira dans
      ce répertoire !
        JFileChooser fileChooser = new JFileChooser("backup/");
        //Nos filtres
        ZFileFilter zFiltre = new ZFileFilter();
        ZFileFilter filtre = new ZFileFilter(".amz", "Fichier Ardoise
      Mazique");
        File file;

        //Notre zone de dessin
        private DrawPanel drawPanel = new DrawPanel();

        public Fenetre(){
          this.setSize(700, 500);
          this.setLocationRelativeTo(null);
          this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

          //On ajoute nos filtres sur l'objet !
          this.fileChooser.addChoosableFileFilter(zFiltre);
          this.fileChooser.addChoosableFileFilter(filtre);


          //On initialise le menu
          this.initMenu();
          //Idem pour la barre d'outils
          this.initToolBar();
          //On positionne notre zone de dessin
          this.getContentPane().add(drawPanel, BorderLayout.CENTER);
          this.setVisible(true);
        }

        /**
      * Initialise le menu
      */
        private void initMenu(){
          nouveau.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent arg0) {
              drawPanel.erase();
            }
          });

          nouveau.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_N,
      KeyEvent.CTRL_DOWN_MASK));

          quitter.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent arg0) {
              System.exit(0);
            }
          });
          quitter.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_W,
      KeyEvent.CTRL_DOWN_MASK));

          fichier.add(nouveau);
          fichier.addSeparator();

          //On ajoute les nouveau menus !
```

```
        enregistrer.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_S,
KeyEvent.CTRL_DOWN_MASK));
        //On détermine l'action à faire !
        enregistrer.addActionListener(new ActionListener(){

         public void actionPerformed(ActionEvent arg0) {
          ObjectOutputStream oos ;
          //S'il ne s'agit pas du premier enregistrement !
          if(file != null){
           try {

            oos = new ObjectOutputStream(new FileOutputStream(file));
            oos.writeObject(drawPanel.getPoints());
            oos.close();

           } catch (FileNotFoundException e) {
            e.printStackTrace();
           } catch (IOException e) {
            e.printStackTrace();
           }
          }
          //Sinon on demande le nom du fichier
          else{
           if(fileChooser.showSaveDialog(null) ==
JFileChooser.APPROVE_OPTION){
            file = fileChooser.getSelectedFile();
            //Si l'extension est valide
            if(fileChooser.getFileFilter().accept(file))
            {
             try {

              oos = new ObjectOutputStream(new FileOutputStream(file));
              oos.writeObject(drawPanel.getPoints());
              oos.close();

             } catch (FileNotFoundException e) {
              e.printStackTrace();
             } catch (IOException e) {
              e.printStackTrace();
             }
            }
            else{
             //Si vous n'avez pas spécifié une extension valide !
             JOptionPane alert = new JOptionPane();
             alert.showMessageDialog(null, "Erreur d'extension de fichier !
\nVotre sauvegarde a échoué !", "Erreur", JOptionPane.ERROR_MESSAGE);
            }
           }
          }
         }
        });
        fichier.add(enregistrer);

        enregistrerSous.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_S,
KeyEvent.CTRL_DOWN_MASK + KeyEvent.SHIFT_DOWN_MASK));
        //On détermine l'action à faire !
        enregistrerSous.addActionListener(new ActionListener(){

         public void actionPerformed(ActionEvent arg0) {
          if(fileChooser.showSaveDialog(null) ==
JFileChooser.APPROVE_OPTION){
           file = fileChooser.getSelectedFile();
           //Si l'extension est valide
           if(fileChooser.getFileFilter().accept(file))
           {
            try {

             ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(file));
             oos.writeObject(drawPanel.getPoints());
```

```
            oos.close();

          } catch (FileNotFoundException e) {
           e.printStackTrace();
          } catch (IOException e) {
           e.printStackTrace();
          }
         }
         else{
          //Si vous n'avez pas spécifié une extension valide !
          JOptionPane alert = new JOptionPane();
          alert.showMessageDialog(null, "Erreur d'extension de fichier !
\nVotre sauvegarde a échoué !", "Erreur", JOptionPane.ERROR_MESSAGE);
         }
        }
       }
      });
      fichier.add(enregistrerSous);

      ouvrir.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_O,
KeyEvent.CTRL_DOWN_MASK));
      ouvrir.addActionListener(new ActionListener(){
       public void actionPerformed(ActionEvent e){
        if(fileChooser.showOpenDialog(null) ==JFileChooser.APPROVE_OPTION){
         file = fileChooser.getSelectedFile();
         if(fileChooser.getFileFilter().accept(file))
         {
          try {

           ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(file));
           drawPanel.setPoints((ArrayList<Point>)ois.readObject());
           ois.close();

          } catch (FileNotFoundException e1) {
           e1.printStackTrace();
          } catch (IOException e1) {
           e1.printStackTrace();
          } catch (ClassNotFoundException e2) {
           e2.printStackTrace();
          }
         }
         else{
          JOptionPane alert = new JOptionPane();
          alert.showMessageDialog(null, "Erreur d'extension de fichier !
\nVotre chargement a échoué !", "Erreur", JOptionPane.ERROR_MESSAGE);
         }
        }
       }
      });
      fichier.add(ouvrir);
      fichier.addSeparator();

      fichier.add(quitter);
      fichier.setMnemonic('F');

      carre.addActionListener(fListener);
      rond.addActionListener(fListener);
      forme.add(rond);
      forme.add(carre);

      rouge.addActionListener(cListener);
      vert.addActionListener(cListener);
      bleu.addActionListener(cListener);
      couleur.add(rouge);
      couleur.add(vert);
      couleur.add(bleu);

      edition.setMnemonic('E');
      edition.add(forme);
```

```
        edition.addSeparator();
        edition.add(couleur);

        menuBar.add(fichier);
        menuBar.add(edition);

        this.setJMenuBar(menuBar);
      }

      /**
     * Initialise la barre d'outils
     */
      private void initToolBar(){

        JPanel panneau = new JPanel();
        square.addActionListener(fListener);
        circle.addActionListener(fListener);
        red.addActionListener(cListener);
        green.addActionListener(cListener);
        blue.addActionListener(cListener);

        toolBar.add(square);
        toolBar.add(circle);

        toolBar.addSeparator();
        toolBar.add(red);
        toolBar.add(blue);
        toolBar.add(green);

        this.getContentPane().add(toolBar, BorderLayout.NORTH);
      }


      //ÉCOUTEUR POUR LE CHANGEMENT DE FORME
      //************************************
      class FormeListener implements ActionListener{
        public void actionPerformed(ActionEvent e) {

    if(e.getSource().getClass().getName().equals("javax.swing.JMenuItem")){
         if(e.getSource()==carre)drawPanel.setPointerType("SQUARE");
         else drawPanel.setPointerType("CIRCLE");
        }
        else{
         if(e.getSource()==square)drawPanel.setPointerType("SQUARE");
         else drawPanel.setPointerType("CIRCLE");
        }
       }
      }


      //ÉCOUTEUR POUR LE CHANGEMENT DE COULEUR
      //**************************************
      class CouleurListener implements ActionListener{
       public void actionPerformed(ActionEvent e) {
        System.out.println(e.getSource().getClass().getName());

    if(e.getSource().getClass().getName().equals("javax.swing.JMenuItem")){
         System.out.println("OK !");
         if(e.getSource()==vert)drawPanel.setPointerColor(Color.green);
         else if(e.getSource()==bleu)drawPanel.setPointerColor(Color.blue);
         else drawPanel.setPointerColor(Color.red);
        }
        else{
         if(e.getSource()==green)drawPanel.setPointerColor(Color.green);
         else if(e.getSource()==blue)drawPanel.setPointerColor(Color.blue);
         else drawPanel.setPointerColor(Color.red);
        }
       }
      }
```

```
    public static void main(String[] args){
     Fenetre fen = new Fenetre();
    }

   }
```

Vous pouvez tester ce code, il fonctionne très bien !

Vous êtes obligés d'écrire le nom de fichier avec extension pour la sauvegarde et l'ouverture, sinon le traitement sera refusé !

Si vous voulez que l'extension soit ajoutée automatiquement à votre nom de fichier, vous devez créer votre propre **`JFileChooser`** et redéfinir la méthode `approveSelection()` .

Bon : je crois que le moment est venu d'aller faire un tour sur le topo !

Retour en haut

Ce qu'il faut retenir

- Les classes traitant des entrées / sorties se trouvent dans le package `java.io` .
- Les classes que nous avons vues dans ce chapitre sont héritées des classes :
  - **`InputStream`**, pour les classes gérant les flux en **entrées** ;
  - **`OutputStream`**, pour les classes gérant les flux en **sorties**.
- La façon dont on travaille avec des flux doit respecter la logique suivante :
  - ouverture de flux ;
  - lecture / écriture de flux ;
  - fermeture de flux.
- La gestion des flux peut engendrer la levée d'exception comme : **`FileNotFoundException`**, **`IOException`**...
- L'action de sauvegarder les objets s'appelle **la sérialisation** !
- Pour qu'un objet soit sérialisable, celui-ci doit implémenter l'interface **`Serializable`**.
- Si un objet sérialisable a un objet d'instance non sérialisable, une exception sera levée lorsque vous voudrez sauvegarder votre objet.
- Une solution consiste à rendre l'objet d'instance sérialisable ou alors à le déclarer **`transient`** afin que celui-ci soit ignoré à la sérialisation !
- L'utilisation de buffer permet une nette amélioration des performances, en lecture et en écriture, avec les fichiers.

Retour en haut

Vous avez pu voir que ce chapitre était très riche en informations et en nouveautés.
Quand je vous disais que le rythme allait s'accélérer...

Je vous rappelle tout de même que les classes que nous venons de voir dans ce chapitre héritent des classes **`InputStream`** et **`OutputStream`**.
Il existe encore des classes dans le package `java.io` qui sont très utiles aussi, mais qui n'héritent pas des mêmes classes...

Vous êtes encore motivés ? Alors, allons-y et terminons-en avec ces maudits flux.

Retour en haut

Les flux d'entrées / sorties (2/2)

Nous voilà dans la dernière ligne droite concernant le flux d'entrées / sorties.

Dans ce chapitre, nous allons aborder une autre hiérarchie de classes présente dans le package `java.io` .
Les super-classes de la deuxième hiérarchie sont :

- la classe **`Reader`** ;
- la classe **`Writer`**.

Vous verrez que l'utilisation des classes de cette hiérarchie est très semblable à ce que nous avons vu lors du chapitre précédent, à une différence près : ces classes ne vont pas lire / écrire des données binaires, mais des caractères UNICODE !

Je ne vais pas vous gâcher la surprise... Alors, on y va ?

Les objets CharArray(Writer/Reader) et String(Writer/Reader)

Nous allons utiliser des objets :

- **CharArray(Writer/Reader)** ;
- **String(Writer/Reader)**.

Ces deux types jouent quasiment le même rôle et ont les mêmes méthodes : celles de leur classe mère !
Ces deux objets n'ajoutent donc aucune nouvelle fonctionnalité à leur objet mère.

Leur principale fonction est de permettre d'écrire un flux de caractères dans un tampon de mémoire adaptatif : un emplacement en mémoire qui peut changer de taille selon les besoins.

Nous n'en avons pas parlé dans le chapitre précédent afin de ne pas l'alourdir, mais il existe des classes remplissant le même rôle que ces classes-ci : **ByteArray(Input/Output)Stream**.

Voyons comment utiliser ces deux objets.

Nous allons commencer par un exemple commenté des objets **CharArray(Writer/Reader)**, retour en mode console :

Code : Java

```
//Package à importer afin d'utiliser l'objet File
import java.io.CharArrayReader;
import java.io.CharArrayWriter;
import java.io.IOException;

public class Main {
 public static void main(String[] args) {
  CharArrayWriter caw = new CharArrayWriter();
  CharArrayReader car;

  try {
   caw.write("Coucouc les zéros");
   //Appel à la méthode toString
   //de manière tacite, de notre objet
   System.out.println(caw);

   //caw.close() n'a aucun effet sur le flux
   //Seul caw.reset() peut tout effacer
   caw.close();

   //on passe un tableau de caractères à l'objet
   //qui va lire le tampon
   car = new CharArrayReader(caw.toCharArray());

   int i ;
   //On remet tous les caractères lus dans un String
   String str = "";
   while(( i = car.read()) != -1)
    str += (char) i;

   System.out.println(str);

  } catch (IOException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
  }
 }
}
```

Je vous laisse le soin d'examiner ce code ainsi que ses effets. Celui-ci est assez commenté, il me semble, pour que vous en

compreniez toutes les subtilités.

L'objet **`String(Writer/Reader)`** fonctionne de la même façon :

Code : Java

```
//Package à importer afin d'utiliser l'objet File
import java.io.IOException;
import java.io.StringReader;
import java.io.StringWriter;

public class Main {
 public static void main(String[] args) {
  StringWriter sw = new StringWriter();
  StringReader sr;

  try {
   sw.write("Coucouc les zéros");
   //Appel à la méthode toString
   //de manière tacite, de notre objet
   System.out.println(sw);

   //caw.close() n'a aucun effet sur le flux
   //Seul caw.reset() peut tout effacer
   sw.close();

   //on passe un tableau de caractères à l'objet
   //qui va lire le tampon
   sr = new StringReader(sw.toString());

   int i ;
   //On remet tous les caractères lus dans un String
   String str = "";
   while(( i = sr.read()) != -1)
    str += (char) i;

   System.out.println(str);

  } catch (IOException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
  }
 }
}
```

En fait, il s'agit du même code avec des objets différents !
Vous savez maintenant comment écrire un flux texte dans un tampon de mémoire...
Peut-être en aurez-vous besoin un jour, qui sait ?

Je vous propose maintenant de voir comment traiter les fichiers texte avec des flux de caractères.
Retour en haut

les classes File(Writer/Reader) et Print(Writer/Reader)

Comme nous l'avons vu dans le chapitre précédent, les objets travaillant avec des flux utilisent des flux binaires.
La conséquence est que, même si vous ne mettez que des caractères dans un fichier, que vous le sauvegardez, les objets vus précédemment traiteront votre fichier comme un fichier contenant des données binaires !

Nous allons voir que, dans le package `java.io` , les objets citées dans le titre de cette sous-section servent à lire / écrire des données d'un fichier texte.

Ce que nous allons faire, c'est tout simplement créer un nouveau fichier et le lire, et le tout en Java !

Code : Java

```
//Package à importer afin d'utiliser l'objet File
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class Main {
 public static void main(String[] args) {

  File file = new File("testFileWriter.txt");

  FileWriter fw;
  FileReader fr;


  try {
   //Création de l'objet
   fw = new FileWriter(file);
   String str = "Bonjour à tous amis ZérOs !\n";
   str += "\tComment allez-vous ? \n";
   //On écrit la chaîne
   fw.write(str);
   //On ferme le flux
   fw.close();

   //création de l'objet de lecture
   fr = new FileReader(file);
   str = "";
   int i = 0;
   //Lecture des données
   while((i = fr.read()) != -1)
    str += (char)i;

   //affichage
   System.out.println(str);

  } catch (FileNotFoundException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
  } catch (IOException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
  }

 }
}
```

Vous pouvez voir que l'affichage est bon et qu'un nouveau fichier vient de faire son apparition dans le dossier contenant votre projet Eclipse !

Tout comme dans le chapitre précédent, la lecture d'un fichier inexistant entraîne une `FileNotFoundException` , et l'écriture peut entraîner un `IOException` !

Classes très simples à utiliser maintenant que vous savez utiliser les flux binaires...
En fait ce chapitre est un peu un clone du précédent, mais les objets ne travaillent pas avec le même type de données.

Pour voir la liste des classes présentes dans cette hiérarchie, allez faire un tour dans l'annexe prévue à cet effet.

Cependant, depuis le JDK 1.4, un nouveau package a vu le jour visant à améliorer les performances des flux, buffers... Traités par `java.io` . Car, ce que vous ignorez encore, c'est que le package que nous explorons depuis le chapitre précédent existe depuis la version 1.1 du JDK.

Il était temps d'avoir une remise à niveau afin d'améliorer les résultats obtenus avec les objets traitant les flux. C'est là que le package `java.nio` a vu le jour !
Retour en haut

Du renouveau chez les flux : le package java.nio

Vous l'avez sûrement deviné, mais "**nio**" signifie : **New I/O**.
Comme je vous l'ai dit précédemment, ce package a été créé afin d'améliorer les performances sur le traitement des fichiers, du réseau et des buffers.

Nous parlerons de la programmation réseau dans un chapitre dédié à ce type de programmation.

Ce package offre une nouvelle façon de lire les données. Nous nous intéresserons uniquement à l'aspect fichier, pour le moment.
Vous avez pu constater que les objets du package `java.io` traitaient les données par octets. Les objets du package `java.nio` , eux, les traitent par blocs de données : ce qui signifie que la lecture en est accélérée !

Tout repose sur deux objets dans ce nouveau package : les **channels** et les **buffers** .
Les channels sont en fait des flux, tout comme dans l'ancien package, mais ceux-ci sont amenés à travailler avec un buffer dont vous définissez la taille !

Pour simplifier au maximum

Lorsque vous ouvrez un flux vers un fichier avec un objet **FileInputStream**, vous pouvez récupérer un canal vers ce fichier. Celui-ci, combiné avec un buffer, vous permettra de lire votre fichier encore plus vite qu'avec un **BufferedInputStream** !

Reprenez le gros fichier que je vous ai fait faire dans le chapitre précédent. Voici l'adresse à laquelle le retrouver pour ceux qui auraient déjà effacé le dit fichier.

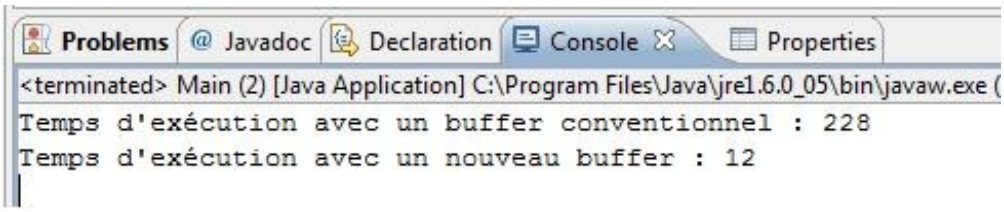
Nous allons maintenant le relire avec ce nouveau package en comparant le buffer conventionnel et la nouvelle façon de faire :

Code : Java

```
//Package à importer afin d'utiliser l'objet File
import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.channels.FileChannel;

public class Main {
 public static void main(String[] args) {

  FileInputStream fis;
  BufferedInputStream bis;
  FileChannel fc;

  try {

   //Création des objets
   fis = new FileInputStream(new File("test.txt"));
   bis = new BufferedInputStream(fis);
   //Démarrage du chrono
   long time = System.currentTimeMillis();
   //Lecture
   while(bis.read() != -1);
   //Temps d'exécution
   System.out.println("Temps d'exécution avec un buffer
conventionnel : " + (System.currentTimeMillis() - time));

   //Re-création d'un flux de fichier
   fis = new FileInputStream(new File("test.txt"));
   //On récupère le canal
   fc = fis.getChannel();
   //On en déduit la taille
   int size = (int)fc.size();
```

```
            //On crée un buffer
            //correspondant à la taille du fichier
            ByteBuffer bBuff = ByteBuffer.allocate(size);

            //Démarrage du chrono
            time = System.currentTimeMillis();
            //Démarrage de la lecture
            fc.read(bBuff);
            //On prépare à la lecture avec l'appel à flip
            bBuff.flip();
            //Affichage du temps d'exécution
            System.out.println("Temps d'exécution avec un nouveau buffer : "
        + (System.currentTimeMillis() - time));

            //Vu que nous avons pris un buffer de byte
            //Afin de récupérer les données, nous pouvons utiliser
            //un tableau de byte
            //La méthode array retourne un tableau de byte
            byte[] tabByte = bBuff.array();

           } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
           } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
           }

         }
        }
```

Et le résultat :



Vous constatez que les gains de performance ne sont pas négligeables...
Il est vrai aussi que ce nouveau package est le plus souvent utilisé pour les flux circulant sur les réseaux..
Je ne m'attarderai donc pas sur le sujet, mais une petite présentation était de mise.

Vous devez savoir tout de même que ce package offre un buffer par type primitif pour la lecture sur le channel, vous trouverez donc les classes :

- **IntBuffer** ;
- **CharBuffer** ;
- **ShortBuffer** ;
- **ByteBuffer** ;
- **DoubleBuffer** ;
- **FloatBuffer** ;
- **LongBuffer**.

Ce chapitre avait pour vocation de vous présenter le reste des classes disponibles dans la hiérarchie du package `java.io` et de vous présenter le package `java.nio` .

Vu qu'il n'y a rien de compliqué et de franchement nouveau (à part **nio**), je vous fais grâce du topo et du QCM... Mais ce sera

l'une des rares fois !

Retour en haut

Je vous conseille de prendre le temps de bien digérer tout ça, de faire des tests, de fumer une clope, de boire un café...
Bref, ne vous jetez pas tout de suite sur le prochain chapitre.

La gestion des flux n'est pas quelque chose d'évident.

Après la pause syndicale, je vous propose un TP des familles, histoire de mettre en pratique tout ce vous avez vu... Les flux y compris.

En avant pour : **le jeu du pendu !**

Retour en haut

TP : Le penduZ

Ce TP va sûrement être le plus difficile que vous ayez fait jusqu'à présent !
Il fait appel à énormément de choses, et surtout à de la logique.

Par contre, la solution que je vous apporte utilise des design patterns : je vous conseille donc de lire les quatre premiers chapitres de cette partie avant de commencer.

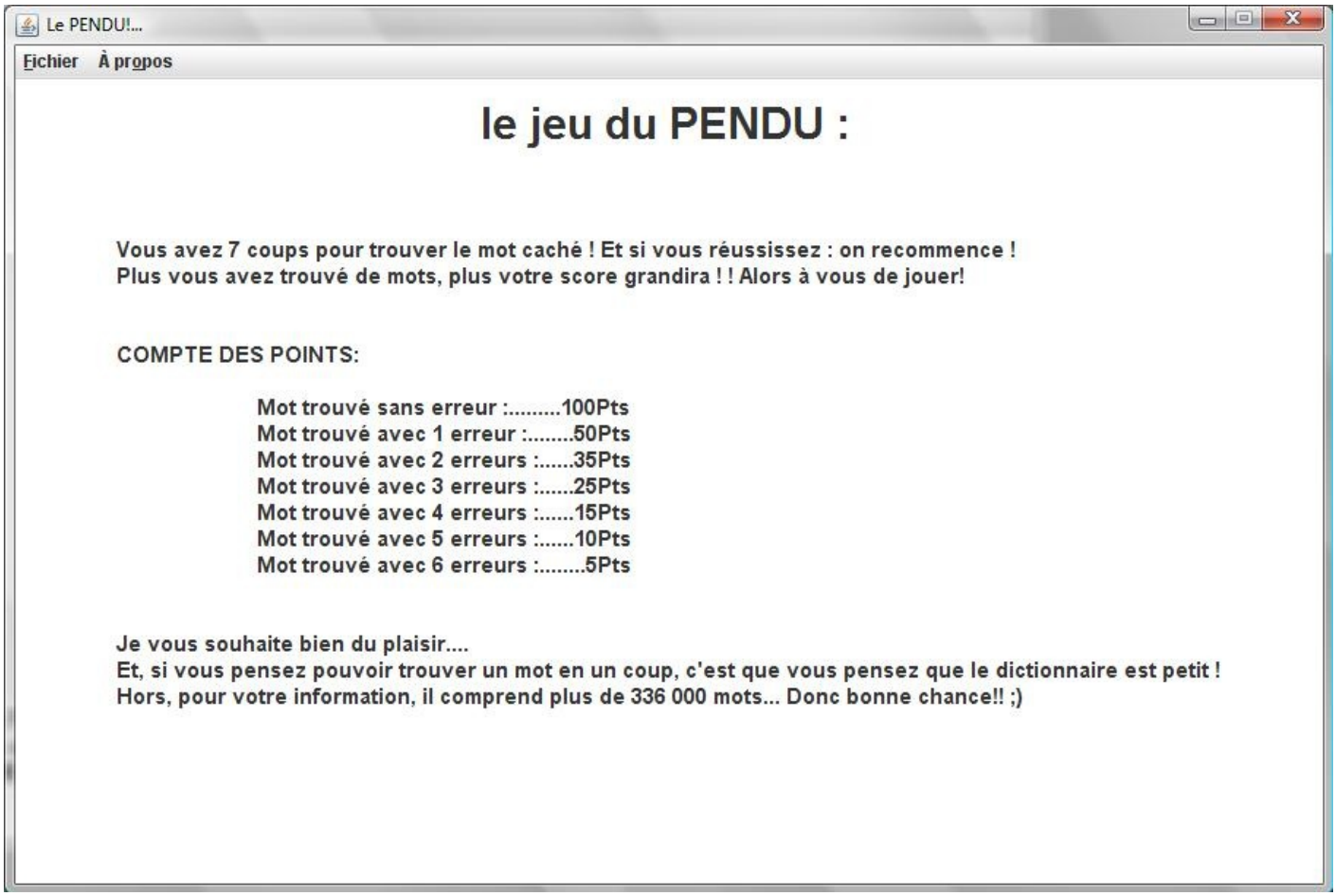
Bon, voyons voir à quelle sauce je vais vous manger !

Cahier des charges

Vous devez faire un jeu du pendu en Java évènementiel avec sauvegarde des 10 meilleurs scores !
Toutefois, j'ai des exigences :

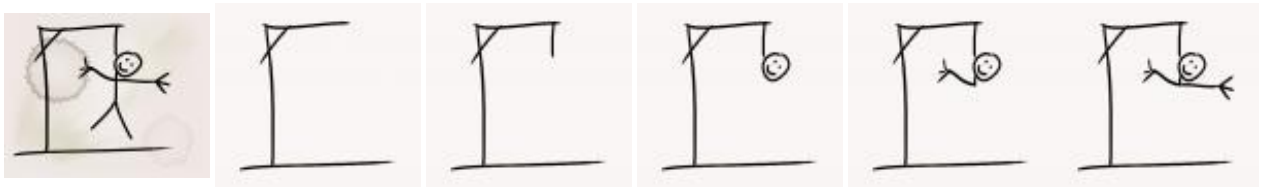
- l'application devra avoir les menus : Nouveau, Scores, Règles, À propos ;
- une page d'accueil devra être mise en place ;
- les points devront être cumulés sur le total des mots trouvés et selon les erreurs commises ;
- il faudra aller vérifier si le joueur est dans le top 10, le cas échéant, on lui demande son pseudo, on enregistre et on redirige vers la page des scores ;
- si le joueur n'a pas assez de points, on redirige vers la page d'accueil ;
- il faudra essayer d'utiliser au moins le pattern observer !

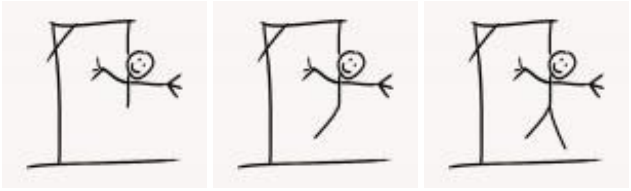
Voici les règles du jeu :



Et voici les écrans que j'ai obtenus :

Je vous donne aussi les images que j'ai utilisées pour faire le pendu :





Vous aurez aussi besoin du fichier que nous avons utilisé dans les chapitres précédents : vous savez, le fichier contenant plein de mots en français. Pour ceux qui l'auraient déjà effacé : c'est par là !

Il me reste encore quelques recommandations...
Retour en haut

Prérequis

Vous allez devoir utiliser les flux afin de parcourir le fichier texte...
Il y a plus de 336000 lignes dedans, vous pouvez donc choisir un chiffre aléatoire entre 0 et 336000 et aller récupérer le mot choisi !
Pour avoir un chiffre aléatoire entre 0 et 336529, j'ai fait ceci :

Code : Java

```
int i = (int)(Math.random() * 100000);
while(i > 336529){
  i /= 2;
}
```

Mais vous pouvez aussi faire comme ceci :

Code : Java

```
int nbre = (int)(Math.random()*336529);
```

Pour récupérer les mots par ligne, j'ai utilisé un **`LineNumberReader`** : vu que cet objet nous retourne le numéro de la ligne en invoquant la méthode `getLineNumber()` , cet objet était tout indiqué !

Il y a aussi un point qui devrait vous poser problème : la mise à jour de **`JPanel`**.
J'ai fait comme ceci : j'ai tout retiré de mon conteneur avec la méthode `removeAll()` , j'y ai remis des composants et invoqué la méthode `revalidate()` afin de modifier l'affichage !

Il va également falloir que vous pensiez à gérer les caractères accentués lorsque vous cliquerez sur le bouton 'E' par exemple, vous devrez aussi afficher les lettres 'E' accentuées...

Je ne vais pas tout vous dire... ce serait dommage...

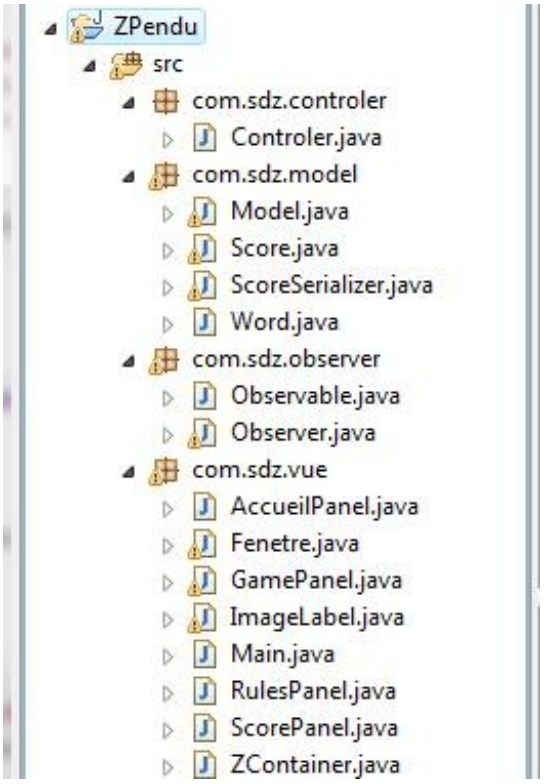
Par contre, j'insiste sur le fait que c'est un TP difficile, et il vous faudra sûrement plusieurs heures avant d'en venir à bout !
Prenez donc le temps de poser les problèmes, réfléchissez bien, et codez bien !

Juste pour être bien sûr que vous ayez compris : je vous conseille vivement d'aller lire les chapitres sur les design patterns : j'en ai utilisé, et en plus, j'ai rangé mes classes en packages...
Ne vous inquiétez pas, tout est expliqué dans un des chapitres sur les DP...

Allez, en avant les zéros !
Retour en haut

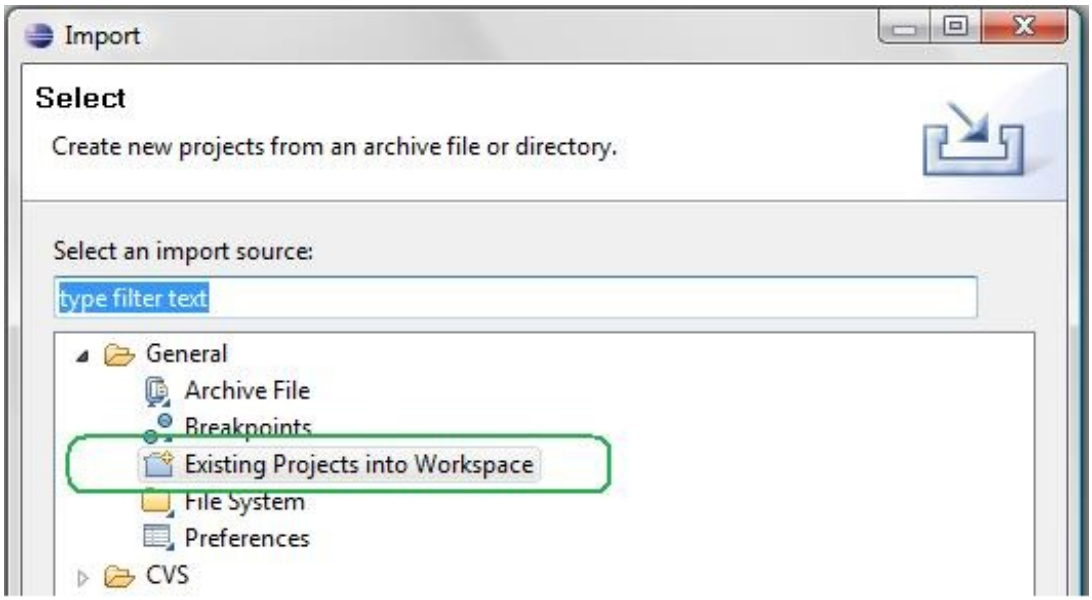
Correction

Une fois n'est pas coutume, je ne vais pas mettre tous les codes source ici, je vais vous founir tout mon projet Eclipse avec un .jar exécutable !
Et pour cause, j'ai beaucoup de classes :



Voici donc une astuce d'Eclipse afin de rapatrier un projet.

Une fois Eclipse ouvert, faite un clic droit dans la zone où se trouvent vos projets et faites "**import**" ; choisissez "**Existing project**" dans "**General**" :



Il ne vous reste plus qu'à spécifier l'endroit où vous avez décompressé cette archive.

Une fois décompressé, vous devriez pouvoir lancer le fichier .jar en double cliquant dessus !
Si rien ne se passe, mettez à jour vos variables d'environnement (cf première partie du tuto).

Vous devriez être capables de comprendre ce code sans souci !
Avec les progrès en Java que vous venez de faire, c'est du gâteau...

Je reste à disposition par MP pour toute précision !
Retour en haut
Je vous avais prévenus que celui-ci serait difficile !
Mais admettez que vous devez être contents de vous...

Bon, je vous conseille de faire un 'tite pause.
N'allez pas faire surchauffer vos méninges...

Dès que vous êtes prêts, on part pour : **Gérez vos conteneurs**.
Retour en haut

Gérez vos conteneurs

Dans le dernier TP, vous avez dû remarquer que vous devez spécifier une taille spécifique de conteneur. Ceci afin que vos `JPanel` puissent coexister dans une même fenêtre !
Dans ce chapitre, nous allons mettre fin à ce calvaire...

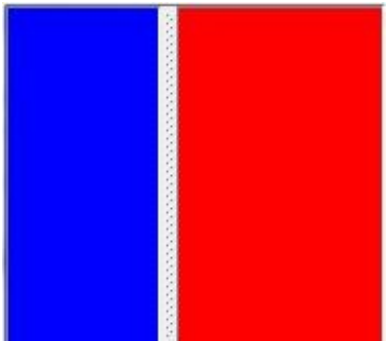
Il y a plusieurs objets qui peuvent vous aider à mieux gérer le contenu de vos IHM ; les deux objets abordés ici vont, je pense, vous rendre un sacré service...

Rendre vos conteneurs fractionnables

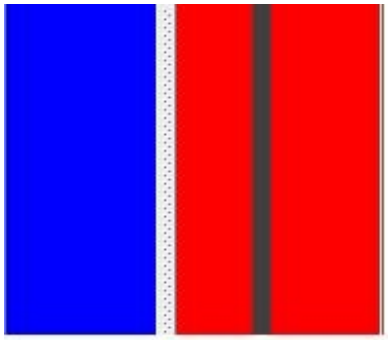
Qu'est-ce que tu entends par *fractionnables* ?

Avant de vous faire un laïus (un petit, je vous rassure), voici à quoi ressemblent des fenêtres à contenus fractionnables :

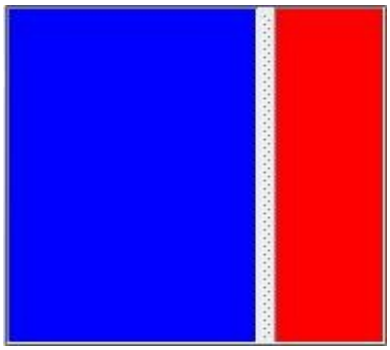
votre contenu avec une séparation



le même contenu pendant le déplacement de la séparation



et le résultat après déplacement



Ceci correspond à l'intérieur d'un objet `JFrame`.
La barre au milieu est un objet déplaçable qui permet d'agrandir une zone tout en rétrécissant celle d'à côté...
Ici, dans la première image, le barre est vers la gauche. La deuxième image est prise pendant que je déplace la barre centrale, et enfin la troisième correspond au résultat lorsque j'ai relâché le bouton de ma souris !
Vous pouvez constater que le conteneur de gauche est devenu plus grand au détriment de celui de droite...

C'est comme une espèce de séparateur qui fonctionne à la façon des vases communiquants...

Vous avez tout compris !
Je vous rassure tout de suite, ce composant est très simple d'utilisation...
En fait, les composants abordés dans ce chapitre s'utilisent facilement.

Je ne vais pas vous faire mariner plus longtemps : l'objet utilisé ici est un `JSplitPane`.

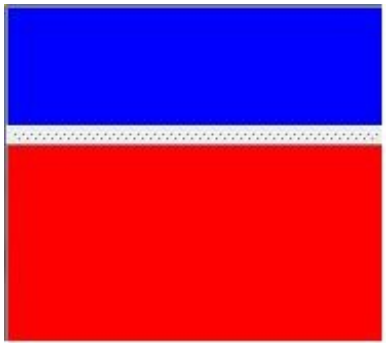
Voici le code source que j'ai utilisé pour avoir le résultat ci-dessus :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JSplitPane;


public class Fenetre extends JFrame {

 //On déclare notre objet JSplitPane
 private JSplitPane split;

 //Vous êtes habitués à cette classe, maintenant... ;)
 public Fenetre(){
  this.setLocationRelativeTo(null);
  this.setTitle("Gérer vos conteneur");
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setSize(200, 200);

  //On crée deux conteneurs de couleurs différentes
  JPanel pan = new JPanel();
  pan.setBackground(Color.blue);

  JPanel pan2 = new JPanel();
  pan2.setBackground(Color.red);

  //On construit enfin notre séparateur
  split = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, pan, pan2);

  //On le passe ensuite au contentPane de notre objet Fenetre
  //placé au centre pour qu'il utilise tout l'espace disponible
  this.getContentPane().add(split, BorderLayout.CENTER);
                this.setVisible(true);
 }

 public static void main(String[] args){
  Fenetre fen = new Fenetre();
 }

}
```

Nous voyons l'attribut `JSplitPane.HORIZONTAL_SPLIT` dans le constructeur de l'objet : cela veut-il dire que nous pouvons avoir d'autres types de séparations ?

Je vois que vous comprenez très vite !
Vous pouvez avoir une séparation verticale en utilisant l'attribut `JSplitPane.VERTICAL_SPLIT` :



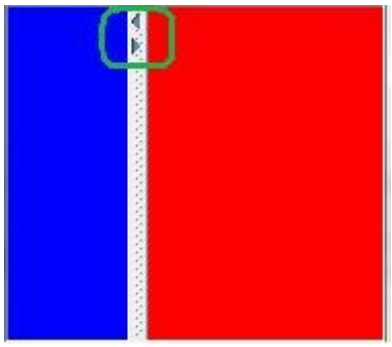
On le savait !
Mais, dis-nous, les deux autres paramètres sont nécessairement des **JPanel** ?

Ici, j'ai utilisé des **JPanel**, mais en fait, vous pouvez utiliser n'importe quelle classe dérivant de **JComponent** (conteneur, bouton, case à cocher...) : elle n'est pas belle, la vie ?

Je ne vous avais donc pas menti : cet objet est vraiment très simple d'utilisation, mais je ne vais pas vous laisser tout de suite...
Vous ne l'avez peut-être pas remarqué mais ces objets ne peuvent pas faire disparaître entièrement les côtés.
Dans notre cas, la fenêtre est petite, mais vous aurez peut-être l'occasion d'avoir une grande IHM et souvent d'agrandir / de rétrécir vos contenus.
L'objet **JSplitPane** a une méthode qui permet de rendre la barre de séparation intelligente, enfin presque...
La dite méthode ajoute deux petits boutons sur votre barre et, lorsque vous cliquerez dessus, rétrécira le côté vers lequel pointe la flèche dans le bouton.

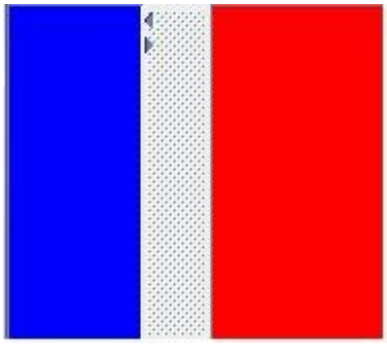
Hein ?!?

Voici l'illustration de mes propos :



Pour avoir ces deux boutons en plus sur votre barre, il vous suffit d'invoquer la méthode `split.setOneTouchExpandable(true);` (mon objet s'appelle toujours `split`) et le tour est joué !
Amusez-vous à cliquer sur ces boutons et vous verrez à quoi ils servent.

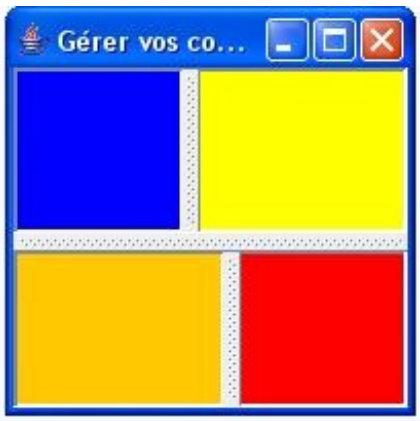
Avant de vous laisser fouiner un peu sur cet objet, vous devez savoir que vous pouvez définir une taille de séparateur grâce à la méthode `split.setDividerSize(int size)` ; voici ce que j'ai obtenu avec une taille de 35 :



Vous pouvez également définir où doit s'afficher la barre de séparation. Ceci se fait grâce à la méthode `setDividerLocation(int location);` ou `setDividerLocation(double location);` .
Avant de vous montrer un exemple de code utilisant cette méthode, vous avez dû comprendre que, vu que cet objet peut prendre des sous-classes de **JComponent**, il pouvait donc aussi prendre des **JSplitPane** !

Voici ce que j'ai pu obtenir :

Secret (cliquez pour afficher)



Ceci, avec ce code :

Secret (cliquez pour afficher)

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSplitPane;
import javax.swing.JTextArea;

public class Fenetre extends JFrame {

 //On déclare notre objet JTextPane
 private JTextArea textPane = new JTextArea();
 //L'objet qui va gérer le scroll
 //En lui passant un objet JComponent dans le constructeur
 private JScrollPane scroll = new JScrollPane(textPane);

 //Vous êtes habitués à cette classe, maintenant... ;)
 //On déclare notre objet JSplitPane
 private JSplitPane split, split2, split3;

 //Familiers avec celle-là également... ;)
 public Fenetre(){
  this.setLocationRelativeTo(null);
  this.setTitle("Gérer vos conteneur");
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setSize(200, 200);


  //On crée deux conteneurs de couleurs différentes
  JPanel pan = new JPanel();
  pan.setBackground(Color.blue);
  JPanel pan2 = new JPanel();
  pan2.setBackground(Color.red);
  JPanel pan3 = new JPanel();
  pan3.setBackground(Color.orange);
  JPanel pan4 = new JPanel();
  pan4.setBackground(Color.YELLOW);
  //On construit enfin notre séparateur
  split = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, pan, pan4);
  //On place le séparateur
  split.setDividerLocation(80);
  split2 = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, pan3,
pan2);
  //On place le séparateur
  split2.setDividerLocation(100);
```

```
        //On passe les deux précédents JSplitPane à celui-ci
        split3 = new JSplitPane(JSplitPane.VERTICAL_SPLIT, split,
    split2);
        //On place le séparateur
        split3.setDividerLocation(80);

        //On le passe ensuite au contentPane de notre objet Fenetre
        //placé au centre pour qu'il utilise tout l'espace disponible
        this.getContentPane().add(split3, BorderLayout.CENTER);
        this.setVisible(true);
       }

       public static void main(String[] args){
        Fenetre fen = new Fenetre();
       }

    }
```

Ce que je peux vous conseiller, c'est d'essayer d'adapter cet objet au dernier TP.
Je pense que vous en savez assez pour utiliser cet objet comme il convient. Nous allons à présent voir un autre objet bien pratique, lui aussi. Il permet d'avoir un scroll à côté de vos conteneurs afin de pouvoir dépasser les limites de ceux-ci !

Retour en haut

Ajouter des scrolls

Afin que vous puissiez mieux juger de l'utilité de l'objet que nous allons utiliser ici, nous allons voir un nouvel objet de texte : un **JTextArea**.
Cet objet est très simple au demeurant, c'est une forme de **JTextField**, mais plus grand ! Voyez plutôt :

Voici le code source utilisé pour avoir ce résultat (sans le texte, hein...) :

Code : Java

```
import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.JTextArea;

public class Fenetre extends JFrame {

 //On déclare notre objet JTextArea
 private JTextArea textPane = new JTextArea();

 //Vous êtes habitués à cette classe, maintenant... ;)
 public Fenetre(){
  this.setLocationRelativeTo(null);
  this.setTitle("Gérer vos conteneur");
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setSize(200, 200);

  //On ajoute l'objet au contentPane de notre fenêtre
  this.getContentPane().add(textPane, BorderLayout.CENTER);
  this.setVisible(true);
 }

 public static void main(String[] args){
  Fenetre fen = new Fenetre();
 }

}
```

Vous pourrez voir que nous pouvons directement écrire dans ce composant, et que celui-ci ne retourne pas directement à la ligne si vous atteignez le bord droit de la fenêtre.

Afin de voir si les lettres tapées au clavier sont bien dans notre objet, vous pouvez récupérer le texte saisi grâce à la méthode `getText()` .
Voici un code d'exemple ainsi que le résultat obtenu :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
```

```
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextArea;

public class Fenetre extends JFrame {

 //On déclare notre objet JTextArea
 private JTextArea textPane = new JTextArea();

 //Vous êtes habitués à cette classe, maintenant... ;)
 public Fenetre(){
  this.setLocationRelativeTo(null);
  this.setTitle("Gérer vos conteneur");
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setSize(200, 200);

  JButton bouton =  new JButton("Bouton");
  bouton.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent e){
    System.out.println("Texte écrit dans le JTextArea : ");
    System.out.println("--------------------------------");
    System.out.println(textPane.getText());
   }
  });
  //On ajoute l'objet au contentPane de notre fenêtre
  this.getContentPane().add(textPane, BorderLayout.CENTER);
  this.getContentPane().add(bouton, BorderLayout.SOUTH);
  this.setVisible(true);
 }

 public static void main(String[] args){
  Fenetre fen = new Fenetre();
 }

}
```

Et le résultat :

```
Texte écrit dans le JTextArea :
--------------------------------
Coucou les ZérOs !

C'est encore moi...

Hi ! Hi !Hi !
```

Le code est simple et clair !
Cependant, les plus curieux d'entre vous l'auront remarqué : si vous écrivez trop de lignes, vous dépassez la limite imposée par le bas de votre fenêtre... Le texte est bien écrit mais vous ne le voyez pas...
Exactement comme pour le bord droit...

Pour ce genre de problème, il existe ce qu'on appelle **des scrolls**.
Ce sont de petit ascenseurs positionnés sur le côté et / ou sur le bas de votre fenêtre et qui vous permettent de dépasser les limites imposées par la dite fenêtre !

Voyez plutôt :

Vous voyez le petit ascenseur à droite et en bas de la fenêtre...
Avec ceux-ci, fini les problèmes de taille de vos conteneurs !

Voici le code que j'ai utilisé afin d'avoir ce résultat :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

public class Fenetre extends JFrame {

 //On déclare notre objet JTextArea
 private JTextArea textPane = new JTextArea();
 //L'objet qui va gérer le scroll
 //En lui passant un objet JComponent dans le constructeur
 private JScrollPane scroll = new JScrollPane(textPane);

 //Vous êtes habitués à cette classe, maintenant... ;)
 public Fenetre(){
  this.setLocationRelativeTo(null);
  this.setTitle("Gérer vos conteneur");
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setSize(200, 200);

  JButton bouton =  new JButton("Bouton");
  bouton.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent e){
    System.out.println("Texte écrit dans le JTextArea : ");
    System.out.println("-------------------------------");
    System.out.println(textPane.getText());
   }
```

```
        });

        //On ajoute l'objet au contentPane de notre fenêtre
        this.getContentPane().add(scroll, BorderLayout.CENTER);
        this.getContentPane().add(bouton, BorderLayout.SOUTH);
        this.setVisible(true);
      }

      public static void main(String[] args){
        Fenetre fen = new Fenetre();
      }

    }
```

Vous avez vu que l'objet utilisé afin d'avoir un ascenseur s'appelle un : **JScrollPane**.
Maintenant, vous pouvez écrire aussi loin que vous le voulez, **vers le bas et vers la droite !**
Les ascenseurs apparaissent automatiquement lorsque vous dépassez les limites autorisées. Cependant, vous pouvez redéfinir un comportement grâce aux méthodes :

- `scroll.setHorizontalScrollBarPolicy(int policy)` : permet de définir le comportement du scroll en bas de votre fenêtre ;
- `scroll.setVerticalScrollBarPolicy(int policy)` : permet de définir le comportement du scroll à droite de votre fenêtre.

Le paramètre de ces méthodes est un entier défini dans la classe **JScrollPane**, il peut prendre les valeurs suivantes :

- `JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED` : le scroll vertical n'est visible que s'il est nécessaire, donc s'il y a dépassement de la taille en hauteur ;
- `JScrollPane.VERTICAL_SCROLLBAR_NEVER` : le scroll vertical n'est jamais visible, même si vous dépassez, mais par contre, le conteneur s'allonge tout de même !
- `JScrollPane.VERTICAL_SCROLLBAR_ALWAYS` : le scroll vertical est toujours visible même si vous ne dépassez pas !

Les mêmes entiers existent pour le scroll horizontal, mais vous devrez remplacer `VERTICAL` par `HORIZONTAL` !

Vous devez tout de même savoir que cet objet en utilise un autre : un **JScrollBar**. Les deux barres de défilement sont deux instances de cet objet...

Je pense vous avoir montré le B.A.BA, mais si vous voulez en savoir plus, vous pouvez aller faire un tour ici.

Nous avons vu comment séparer un conteneur, nous avons vu comment agrandir un conteneur, nous allons maintenant voir comment ajouter dynamiquement des conteneurs !

Retour en haut

Avoir plusieurs contenus

Dans ce chapitre, vous allez apprendre à avoir plusieurs "pages" dans votre IHM...
Jusqu'à maintenant, vous ne pouviez pas avoir plusieurs contenus dans votre fenêtre, à moins de leur faire partager l'espace disponible.

Il existe une solution toute simple qui consiste à créer des onglets, et, croyez-moi, c'est aussi très simple à faire.
L'objet à utiliser est un **JTabbedPane**.
Afin d'avoir un exemple plus ludique, j'ai constitué une classe héritée de **JPanel** afin de créer des onglets ayant une couleur de fond différente... Cette classe ne devrait plus vous poser de problèmes :

Code : Java

```
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;

import javax.swing.JPanel;


public class Panneau extends JPanel{

 private Color color = Color.white;
 private static int COUNT = 0;
 private String message = "";

 public Panneau(){}
 public Panneau(Color color){
  this.color = color;
  this.message = "Contenu du panneau N°" + (++COUNT);
 }
 public void paintComponent(Graphics g){
  g.setColor(this.color);
  g.fillRect(0, 0, this.getWidth(), this.getHeight());
  g.setColor(Color.white);
  g.setFont(new Font("Arial", Font.BOLD, 15));
  g.drawString(this.message, 10, 20);
 }
}
```

J'ai utilisé cet objet afin de créer un tableau de **Panneau**. Chaque instance est ensuite ajoutée à mon objet gérant les onglets via sa méthode `add(String title, JComponent comp)` .
Vous voudriez peut-être avoir le code tout de suite...
Le voici :

Code : Java

```
import java.awt.Color;

import javax.swing.JFrame;
import javax.swing.JTabbedPane;

public class Fenetre extends JFrame {

 private JTabbedPane onglet;
 //Vous êtes habitués à cette classe, maintenant... ;)
 public Fenetre(){
  this.setLocationRelativeTo(null);
  this.setTitle("Gérer vos conteneurs");
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setSize(400, 200);

  //Création de plusieurs Panneau
  Panneau[] tPan = { new Panneau(Color.RED),
       new Panneau(Color.GREEN),
```

```
            new Panneau(Color.BLUE)};

        //Création de notre conteneur d'onglets
        onglet = new JTabbedPane();
        int i = 0;
        for(Panneau pan : tPan){
         //Méthode d'ajout d'onglet
         onglet.add("Onglet N°"+(++i), pan);
         //Vous pouvez aussi utiliser la méthode addTab
         //onglet.addTab("Onglet N°"+(++i), pan);

        }
        //on passe ensuite les onglets au contentPane
        this.getContentPane().add(onglet);
        this.setVisible(true);
       }

      public static void main(String[] args){
       Fenetre fen = new Fenetre();
      }

     }
```

Ce qui m'a donné :

Vous constatez que l'utilisation de cet objet est très simple, là aussi...
Je vais tout de même vous montrer quelques méthodes bien utiles. Par exemple, vous pouvez ajouter une image en guise d'icône à côté du titre de l'onglet.
Ce qui pourrait nous donner :

Le code est identique au précédent, à l'exception de ce qu'il y a dans la boucle :

Code : Java

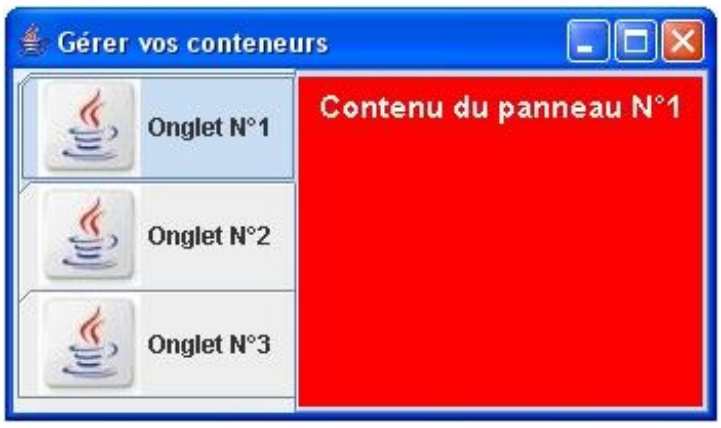
```
for(Panneau pan : tPan){
 //Méthode d'ajout d'onglets
 onglet.add("Onglet N°"+(++i), pan);
        //On ajoute l'image à l'onglet en cours
        //Les index d'onglets fonctionnent comme les tableaux : ils
commencent à 0
 onglet.setIconAt((i - 1), new ImageIcon("java.jpg"));

        //Vous pouvez aussi utiliser la méthode addTab
        //onglet.addTab("Onglet N°"+(++i), new
ImageIcon("java.jpg"), pan);
}
```

Vous avez également la possibilité de changer l'emplacement des entêtes d'onglets en spécifiant le dit emplacement dans le constructeur, comme ceci :

Code : Java

```
//affiche les onglets en bas de la fenêtre
JTabbedPane onglet = new JTabbedPane(JTabbedPane.BOTTOM);

//affiche les onglets à gauche de la fenêtre
JTabbedPane onglet = new JTabbedPane(JTabbedPane.LEFT);

//affiche les onglets à droite de la fenêtre
JTabbedPane onglet = new JTabbedPane(JTabbedPane.RIGHT);
```

Voici ce que vous pouvez obtenir :



Vous pouvez aussi utiliser la méthode `setTabPlacement(JTabbedPane.BOTTOM);` qui a le même effet : ici, la barre d'exploration des onglets sera située en bas du conteneur.

Vous avez aussi la possibilité d'ajouter ou de retirer des onglets. Pour ajouter, vous avez deviné comment procéder !
Pour retirer un onglet, nous allons utiliser la méthode `remove(int index)` . Cette méthode parle d'elle-même, elle va retirer l'onglet ayant pour index le paramètre passé !

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTabbedPane;


public class Fenetre extends JFrame {

 //On déclare notre objet JSplitPane
 private JTabbedPane onglet;
 //Compteur pour le nombre d'onglets
 private int nbreTab = 0;

 //Vous êtes habitués à cette classe, maintenant... ;)
 public Fenetre(){
  this.setLocationRelativeTo(null);
  this.setTitle("Gérer vos conteneurs");
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setSize(400, 200);

  //Création de plusieurs Panneau
  Panneau[] tPan = { new Panneau(Color.RED),
       new Panneau(Color.GREEN),
```

```
            new Panneau(Color.BLUE)};

        //Création de notre conteneur d'onglets
        onglet = new JTabbedPane();
        for(Panneau pan : tPan){
         //Méthode d'ajout d'onglets
         onglet.addTab("Onglet N°"+(++nbreTab), pan);
        }
        //on passe ensuite les onglets au contentPane
        this.getContentPane().add(onglet, BorderLayout.CENTER);

        //Ajout du bouton pour ajouter des onglets
        JButton nouveau = new JButton("Ajouter un onglet");
        nouveau.addActionListener(new ActionListener(){
         public void actionPerformed(ActionEvent e){
          onglet.add("Onglet N°"+(++nbreTab), new
      Panneau(Color.DARK_GRAY));
         }
        });

        //Ajout du bouton pour retirer l'onglet sélectionné
        JButton delete = new JButton("Effacer l'onglet");
        delete.addActionListener(new ActionListener(){
         public void actionPerformed(ActionEvent e){
          //On récupère l'index de l'onglet sélectionné
          int selected = onglet.getSelectedIndex();
          //S'il n'y a plus d'onglet, la méthode ci-dessus retourne -1
          if(selected > -1)onglet.remove(selected);
         }
        });

        JPanel pan = new JPanel();
        pan.add(nouveau);
        pan.add(delete);

        this.getContentPane().add(pan, BorderLayout.SOUTH);
        this.setVisible(true);
       }


       public static void main(String[] args){
        Fenetre fen = new Fenetre();
       }
      }
```

Ce qui peut vous donner :

Voilà : vous venez de voir pas mal de choses, simples et utiles !
Il est temps de se diriger vers le topo...
Retour en haut

Ce qu'il faut retenir

- Vous pouvez scinder le contenu de votre IHM en utilisant un **JSplitPane**.
- Celui-ci vous permet de couper votre IHM dans le sens vertical ou horizontal.
- Vous pouvez spécifier la taille du séparateur grâce à la méthode `split.setDividerSize(int size)` et sa position avec `split.setDividerLocation(int size)` .
- Vous avez la possibilité d'ajouter des ascenseurs à vos conteneurs grâce à l'objet **JScrollPane**.
- Les ascenseurs de cet objet peuvent être affichés ou non selon le mode d'affichage que vous avez choisi avec la méthode `setHorizontalScrollBarPolicy(int policy)` ou / et `setVerticalScrollBarPolicy(int policy)` .
- Vous pouvez utiliser des onglets grâce à l'objet **JTabbedPane**.
- Vous avez la possibilité d'ajouter des onglets grâce à la méthode `addtab()` ou la méthode `add()` .
- Une icône peut être spécifiée à un emplacement donné grâce à la méthode `setIconAt(int index, ImageIcon img)` .
- Vous avez aussi la possibilité de spécifier où doit se mettre la barre de navigation des onglets en le spécifiant dans le constructeur `new JTabbedPane(JTabbedPane.BOTTOM)` ou encore grâce à la méthode `.setTabPlacement(JTabbedPane.BOTTOM);` .
- Des onglets peuvent être retirés en invoquant la méthode `removeAt(int index)` .

Je ne vais pas vous faire l'affront de vous faire un QCM tellement ce chapitre était simple...
Retour en haut
Nos venons de voir comment gérer le contenu de vos fenêtres avec différents objets.
Vous pouvez, bien sûr, les cumuler et mettre des panneaux avec séparateurs dans un onglet...

Vous avez dû remarquer que tous les objets vus dans ce chapitre ont un nom qui se termine avec **Pane**.
Ils appartiennent donc à la famille des conteneurs !

Dans le prochain chapitre, nous allons voir un conteneur bien spécial, lui aussi : **Les arbres**.
Retour en haut

Les arbres

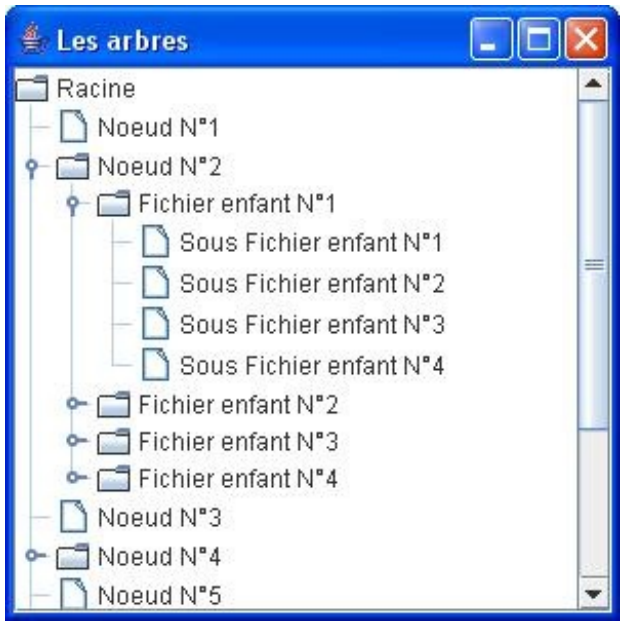
Bon : autant les objets vus dans le chapitre précédent étaient simples, autant celui que nous allons voir est assez compliqué.
Cela n'empêche pas que ce dernier est très pratique et très utilisé.

Vous devez tous déjà avoir vu un arbre, non pas celui du monde végétal, mais celui qui permet d'explorer des dossiers. Nous allons voir comment utiliser et exploiter un tel objet, et interagir avec lui : ne vous inquiétez pas, tout partira de zéro...

Le mieux, c'est encore de rentrer dans le vif du sujet !

La composition des arbres

Tout d'abord, pour ceux qui ne verraient pas de quoi je parle, voici ce qu'on appelle un arbre (**JTree**) :



La chose bien pratique avec cet objet, c'est que, même s'il ne ressemble pas à un chêne ou à un autre arbre, celui-ci est composé de la même façon !

Euh... Qu'est-ce que tu racontes ?

En fait, lorsque vous regardez bien un arbre, celui-ci est constitué de plusieurs sous-ensembles :

- des racines ;
- un tronc ;
- des branches ;
- des feuilles.

L'objet **JTree** se base sur la même architecture, enfin presque. Vous aurez donc :

- une racine : répertoire le plus haut dans la hiérarchie ; ici, seul "**Racine**" est considéré comme une racine ;
- une ou plusieurs branches : un ou plusieurs sous-répertoires, "**Fichier enfant N°1-2-3-4**" sont des branches (ou encore "**Noeud N°2-4-6**") ;
- une ou plusieurs feuilles : éléments se trouvant en bas de la hiérarchie, ici "**Sous Fichier enfant N°1-2-3-4**" ou encore "**Noeud N°1-3-5-7**" sont des feuilles.

Voici le code que j'ai utilisé :

Code : Java

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTree;
import javax.swing.tree.DefaultMutableTreeNode;


public class Fenetre extends JFrame {

 private JTree arbre;

 public Fenetre(){
```

```
        this.setSize(300, 300);
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setTitle("Les arbres");
        //On invoque la méthode de construction de notre arbre
        buildTree();

        this.setVisible(true);
      }

      private void buildTree(){
        //Création d'une racine
        DefaultMutableTreeNode racine = new
DefaultMutableTreeNode("Racine");

        //Nous allons ajouter des branches et des feuilles à notre racine
        for(int i = 1; i < 12; i++){
          DefaultMutableTreeNode rep = new DefaultMutableTreeNode("Noeud N°
"+i);

          //S'il s'agit d'un nombre pair, on rajoute une branche
          if((i%2) == 0){
            //Et une branche en plus ! Une !
            for(int j = 1; j < 5; j++){
              DefaultMutableTreeNode rep2 = new
DefaultMutableTreeNode("Fichier enfant N°" + j);
              //Cette fois, on ajoute nos feuilles
              for(int k = 1; k < 5; k++)
                rep2.add(new DefaultMutableTreeNode("Sous Fichier enfant N°"
+ k));
              rep.add(rep2);
            }
          }
          //On ajoute la feuille ou la branche à la racine
          racine.add(rep);
        }
        //On crée, avec notre hiérarchie, un arbre
        arbre = new JTree(racine);

        //Que nous plaçons sur le ContentPane de notre JFrame à l'aide
d'un scroll
        this.getContentPane().add(new JScrollPane(arbre));
      }

      public static void main(String[] args){
        Fenetre fen = new Fenetre();
      }

    }
```

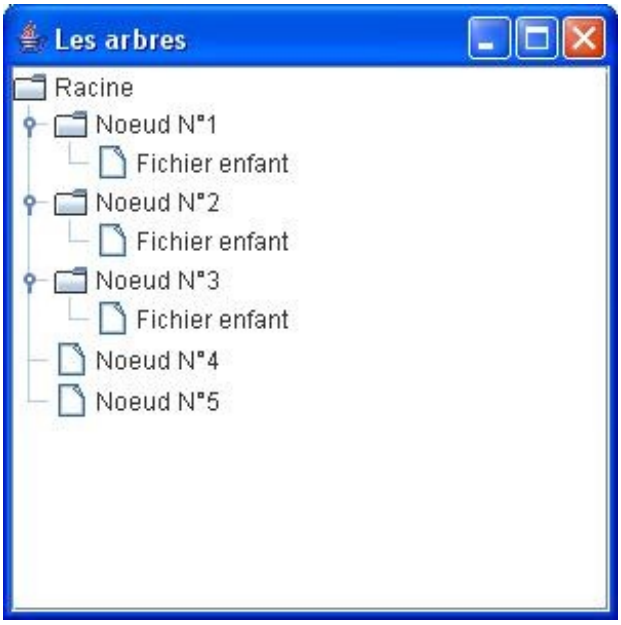
Si vous avez du mal à vous y retrouver, essayez cette version de la méthode `buildTree()` :

Code : Java

```
    private void buildTree(){
      //Création d'une racine
      DefaultMutableTreeNode racine = new
DefaultMutableTreeNode("Racine");

      //Nous allons ajouter des branches et des feuilles à notre racine
      for(int i = 1; i < 6; i++){
        DefaultMutableTreeNode rep = new DefaultMutableTreeNode("Noeud N°
"+i);

        //On rajoute 4 branches
        if(i < 4){
```

```
        DefaultMutableTreeNode rep2 = new DefaultMutableTreeNode("Fichier
enfant");
        rep.add(rep2);
       }
       //On ajoute la feuille ou la branche à la racine
       racine.add(rep);
      }
      //On crée, avec notre hiérarchie, un arbre
      arbre = new JTree(racine);

      //Que nous plaçons sur le ContentPane de notre JFrame à l'aide
d'un scroll
      this.getContentPane().add(new JScrollPane(arbre));
    }
```

Ce qui devrait vous donner :



En ayant manipulé ces deux objets, vous devez vous rendre compte que vous créez une véritable hiérarchie avant de créer votre arbre et d'afficher celui-ci !

Vous devez aussi vous apercevoir que ce type d'objet est tout indiqué pour lister des fichiers ou des répertoires.
D'ailleurs, nous avons vu comment faire ceci lorsque nous avons abordé les flux.
Nous allons utiliser un arbre afin d'afficher notre arborescence de fichiers :

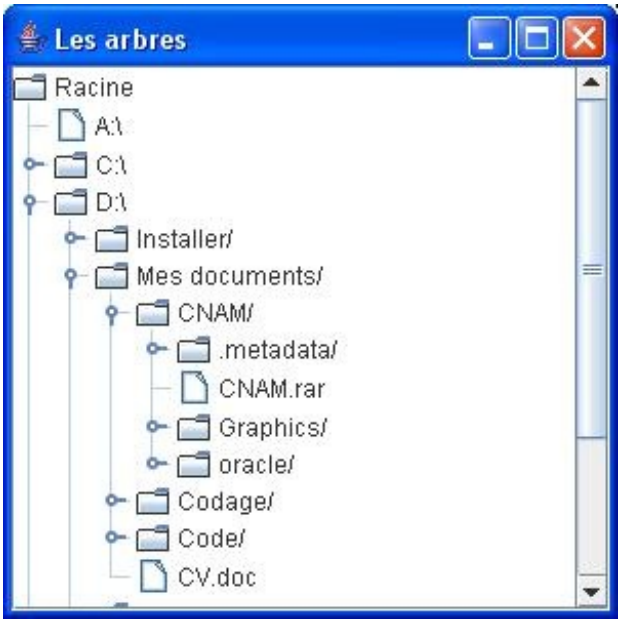
Code : Java

```
import java.io.File;
import java.util.Hashtable;

import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTree;
import javax.swing.event.TreeSelectionEvent;
import javax.swing.event.TreeSelectionListener;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.TreePath;


public class Fenetre extends JFrame {

 private JTree arbre;
 private DefaultMutableTreeNode racine;
 public Fenetre(){
```

```
        this.setSize(300, 300);
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setTitle("Les arbres");
        //On invoque la méthode de construction de notre arbre
        listRoot();

        this.setVisible(true);
      }

      private void listRoot(){

        this.racine = new DefaultMutableTreeNode();

        int count = 0;
        for(File file : File.listRoots())
        {
         DefaultMutableTreeNode lecteur = new
DefaultMutableTreeNode(file.getAbsolutePath());
         try {
          for(File nom : file.listFiles()){
           DefaultMutableTreeNode node = new
DefaultMutableTreeNode(nom.getName()+"\\");

           lecteur.add(this.listFile(nom, node));

          }
         } catch (NullPointerException e) {}

         this.racine.add(lecteur);

         //Si nous avons parcouru plus de 50 dossiers, on sort
         //if(count > 50) {break;}

        }
        //On crée, avec notre hiérarchie, un arbre
        arbre = new JTree(this.racine);

        //Que nous plaçons sur le ContentPane de notre JFrame à l'aide
d'un scroll
        this.getContentPane().add(new JScrollPane(arbre));
      }

      private DefaultMutableTreeNode listFile(File file,
DefaultMutableTreeNode node){
        int count = 0;

        if(file.isFile())
         return new DefaultMutableTreeNode(file.getName());
        else{
         for(File nom : file.listFiles()){
          count++;
          //pas plus de 5 enfants par noeud
          if(count < 5){
           DefaultMutableTreeNode subNode;
           if(nom.isDirectory()){
            subNode = new DefaultMutableTreeNode(nom.getName()+"\\");
            node.add(this.listFile(nom, subNode));
           }else{
            subNode = new DefaultMutableTreeNode(nom.getName());
           }
           node.add(subNode);
          }
         }
         return node;
        }
      }

      public static void main(String[] args){
       Fenetre fen = new Fenetre();
```

```
        }
    }
```

Ce type de code ne devrait plus vous faire peur !
Voici ce que ça me donne, après quelques secondes...



Pas mal, mais du coup, le dossier "**Racine**" ne correspond à rien ici !

Effectivement ; heureusement, il existe une méthode dans l'objet **`JTree`** qui permet de ne pas afficher la racine d'une arborescence : `setRootVisible(Boolean ok);` .
Il vous suffit donc de rajouter cette instruction à la fin de la méthode `listRoot()` de notre objet **`JTree`**, **juste avant d'ajouter notre arbre au ContentPane**.

Bon : vous arrivez à créer et afficher un arbre, maintenant, nous allons voir comment interagir avec celui-ci !
Retour en haut

Des arbres qui vous parlent

Vous connaissez la musique maintenant, nous allons encore implémenter une interface !
Celle-ci se nomme **TreeSelectionListener**.
Elle ne comporte qu'une méthode à redéfinir : `valueChanged(TreeSelectionEvent event)` .

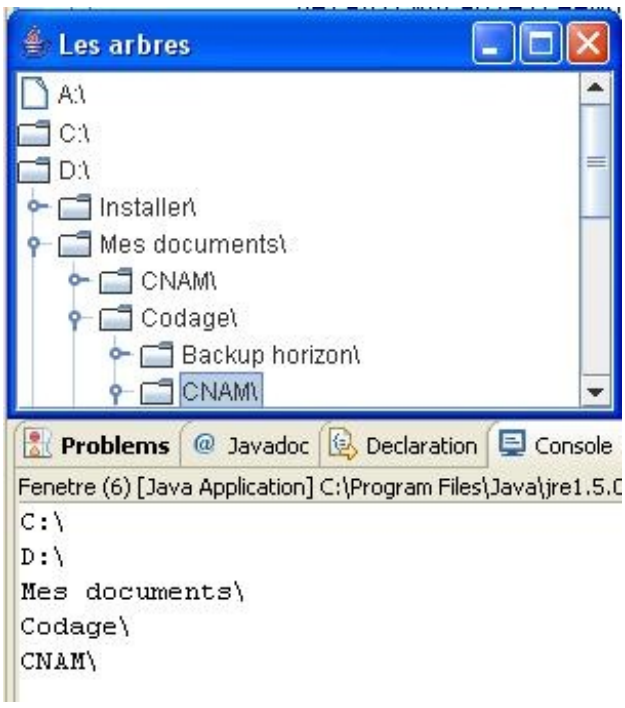
Voici un code utilisant une implémentation de la dite interface :

Code : Java

```
import java.io.File;
import java.util.Hashtable;

import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTree;
import javax.swing.event.TreeSelectionEvent;
import javax.swing.event.TreeSelectionListener;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.TreePath;


public class Fenetre extends JFrame {

 private JTree arbre;
 private DefaultMutableTreeNode racine;
 public Fenetre(){
  this.setSize(300, 200);
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setTitle("Les arbres");
  //On invoque la méthode de construction de notre arbre
  listRoot();

  this.setVisible(true);
 }

 private void listRoot(){

  this.racine = new DefaultMutableTreeNode();

  int count = 0;
  for(File file : File.listRoots())
  {
   DefaultMutableTreeNode lecteur = new
DefaultMutableTreeNode(file.getAbsolutePath());
   try {
    for(File nom : file.listFiles()){
     DefaultMutableTreeNode node = new
DefaultMutableTreeNode(nom.getName()+"\\");

     lecteur.add(this.listFile(nom, node));

    }
   } catch (NullPointerException e) {}

   this.racine.add(lecteur);

   //Si nous avons parcouru plus de 50 dossiers, on sort
   //if(count > 50) {break;}

  }
  //On crée, avec notre hiérarchie, un arbre
  arbre = new JTree(this.racine);
  arbre.setRootVisible(false);
  arbre.addTreeSelectionListener(new TreeSelectionListener(){

   public void valueChanged(TreeSelectionEvent event) {
    if(arbre.getLastSelectedPathComponent() != null){
```

```
        System.out.println(arbre.getLastSelectedPathComponent().toString());
            }
           }
          });
          //Que nous plaçons sur le ContentPane de notre JFrame à l'aide
        d'un scroll
          this.getContentPane().add(new JScrollPane(arbre));
         }

         private DefaultMutableTreeNode listFile(File file,
        DefaultMutableTreeNode node){
          int count = 0;

          if(file.isFile())
           return new DefaultMutableTreeNode(file.getName());
          else{
           for(File nom : file.listFiles()){
            count++;
            //pas plus de 5 enfants par noeud
            if(count < 5){
             DefaultMutableTreeNode subNode;
             if(nom.isDirectory()){
              subNode = new DefaultMutableTreeNode(nom.getName()+"\\");
              node.add(this.listFile(nom, subNode));
             }else{
              subNode = new DefaultMutableTreeNode(nom.getName());
             }
             node.add(subNode);
            }
           }
           return node;
          }
         }

         public static void main(String[] args){
          Fenetre fen = new Fenetre();
         }

        }
```

Ce qui me donne (ben oui, on n'a toujours pas les mêmes dossiers) :



Vous avez maintenant un arbre réactif !
Lorsque vous sélectionnez un dossier, ou un fichier, le nom de ce dernier s'affiche.
Ceci se fait grâce à la méthode `getLastSelectedPathComponent()` qui retourne un **`Object`** correspondant au dernier point de l'arbre qui a été cliqué. Il ne reste plus qu'à utiliser la méthode `toString()` afin de retourner son libellé...

Nous avons réussi à afficher le nom du dernier noeud cliqué, mais nous n'allons pas nous arrêter là... Dans notre cas, il peut être intéressant de connaître le chemin d'accès du noeud dans l'arbre ! Surtout dans notre cas, puisque nous listons le contenu de notre disque.
Nous pouvons donc avoir des informations supplémentaires sur une feuille ou une branche en utilisant un objet **`File`**, par exemple.

Comment fait-on pour connaître le chemin de l'arborescence ?

C'est un tout petit peu plus compliqué, mais rien d'insurmontable...
L'objet **`TreeEvent`** en paramètre de la méthode de l'interface vous donne de précieux renseignements, dont la méthode `getPath()` qui vous retourne un objet **`TreePath`** qui, lui, contient les objets correspondant aux noeuds du chemin d'accès à un point de l'arbre.

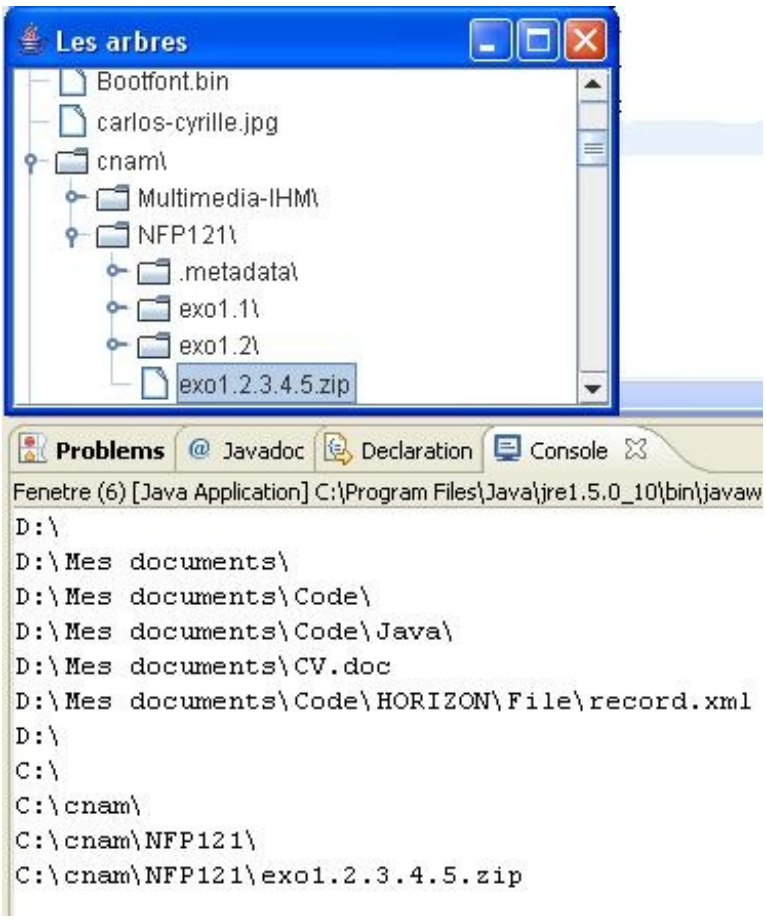
C'est un peu confus, là...

Ne vous inquiétez pas, vous n'avez pas à changer beaucoup de choses pour avoir ce résultat.
En fait, je n'ai modifié que la classe anonyme utilisée pour gérer l'événement déclenché sur l'arbre.
Voici la nouvelle version de votre classe anonyme :

Code : Java

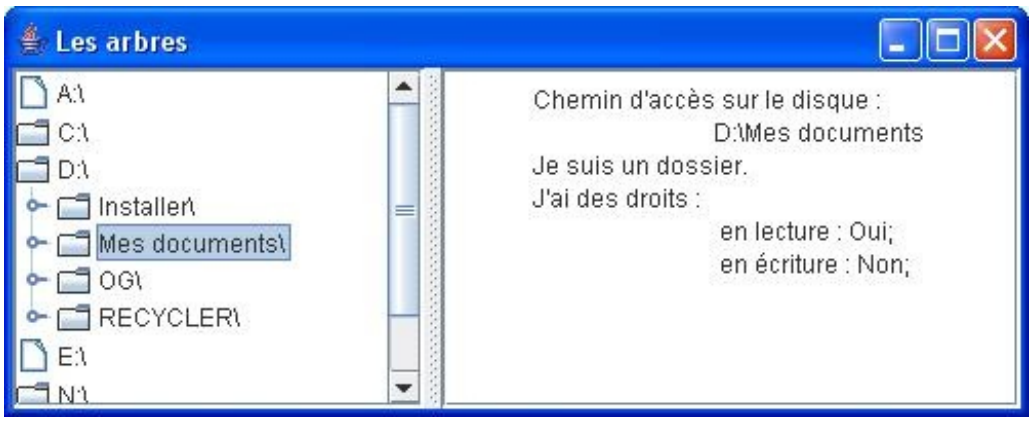
```
arbre.addTreeSelectionListener(new TreeSelectionListener(){

 public void valueChanged(TreeSelectionEvent event) {
  if(arbre.getLastSelectedPathComponent() != null){
   //La méthode getPath retourne un objet TreePath
   System.out.println(getAbsolutePath(event.getPath()));
  }
 }

 private String getAbsolutePath(TreePath treePath){
  String str = "";
  //On balaie le contenu de notre TreePath
  for(Object name : treePath.getPath()){
   //Si l'objet à un nom, on l'ajoute au chemin
   if(name.toString() != null)
    str += name.toString();
  }
  return str;
 }
});
```

Et voici ce que j'ai pu obtenir :



Vous pouvez voir que nous avons maintenant le chemin complet dans notre arbre et, vu que nous utilisons les fichiers de notre système, nous allons pouvoir en savoir plus.
Nous allons donc ajouter un "***coin information***" à droite de notre arbre, dans un conteneur à part.

Essayer de le faire vous-mêmes dans un premier temps, sachant que j'ai obtenu quelque chose comme ça :



Voilà mon code :

Secret (cliquez pour afficher)

Classe Panneau.java

Code : Java

```
import java.awt.Color;
import javax.swing.JPanel;
import javax.swing.JTextArea;
```

```
public class Panneau extends JPanel {
 private String texte = "Racine de l'arbre.";
 private JTextArea jta;

 public Panneau(){
  this.jta = new JTextArea(texte);
  this.setBackground(Color.white);
  this.add(jta);
 }
 public void setTexte(String texte) {
  this.jta.setText(texte);
 }
}
```

Classe Fenetre.java

Code : Java

```
import java.awt.BorderLayout;
import java.io.File;

import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JSplitPane;
import javax.swing.JTree;
import javax.swing.event.TreeSelectionEvent;
import javax.swing.event.TreeSelectionListener;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.TreePath;


public class Fenetre extends JFrame {

 private JTree arbre;
 private DefaultMutableTreeNode racine;
 private Panneau panneau = new Panneau();

 public Fenetre(){
  this.setSize(500, 200);
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setTitle("Les arbres");
  //On invoque la méthode de construction de notre arbre
  listRoot();

  this.setVisible(true);
 }

 private void listRoot(){

  this.racine = new DefaultMutableTreeNode();

  int count = 0;
  for(File file : File.listRoots())
  {
   DefaultMutableTreeNode lecteur = new
DefaultMutableTreeNode(file.getAbsolutePath());
   try {
    for(File nom : file.listFiles()){
     DefaultMutableTreeNode node = new
DefaultMutableTreeNode(nom.getName()+"\\");

     lecteur.add(this.listFile(nom, node));

    }
   } catch (NullPointerException e) {}
```

```
        this.racine.add(lecteur);

        //Si nous avons parcouru plus de 50 dossiers, on sort
        //if(count > 50) {break;}

      }
      //On crée, avec notre hiérarchie, un arbre
      arbre = new JTree(this.racine);
      arbre.setRootVisible(false);
      arbre.addTreeSelectionListener(new TreeSelectionListener(){

       public void valueChanged(TreeSelectionEvent event) {
        if(arbre.getLastSelectedPathComponent() != null){
         //La méthode getPath retourne un objet TreePath
         File file= new File(getAbsolutePath(event.getPath()));
         panneau.setTexte(getDescription(file));
        }
       }

       private String getAbsolutePath(TreePath treePath){
        String str = "";
        //On balaie le contenu de notre TreePath
        for(Object name : treePath.getPath()){
         //Si l'objet à un nom, on l'ajoute au chemin
         if(name.toString() != null)
          str += name.toString();
        }
        return str;
       }
       /**
* Retourne une description d'un objet File
* @param file
* @return
*/
       private String getDescription(File file){
        String str = "Chemin d'accès sur le disque : \n\t" +
file.getAbsolutePath() + "\n";
        str += (file.isFile()) ? "Je suis un fichier.\nJe fais " +
file.length() + " ko\n" : "Je suis un dossier.\n";
        str += "J'ai des droits : \n";
        str += "\t en lecture : " + ((file.canRead()) ? "Oui;" :
"Non;");
        str += "\n\t en écriture : " + ((file.canWrite()) ? "Oui;" :
"Non;");
        return str;
       }
      });
      //On crée un séparateur de conteneur pour réviser
      JSplitPane split = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
new JScrollPane(arbre), new JScrollPane(panneau));
      //On place le séparateur
      split.setDividerLocation(200);
      //On ajoute le tout
      this.getContentPane().add(split, BorderLayout.CENTER);
     }

     private DefaultMutableTreeNode listFile(File file,
DefaultMutableTreeNode node){
      int count = 0;

      if(file.isFile())
       return new DefaultMutableTreeNode(file.getName());
      else{
       for(File nom : file.listFiles()){
        count++;
        //pas plus de 5 enfants par noeud
        if(count < 5){
         DefaultMutableTreeNode subNode;
         if(nom.isDirectory()){
```

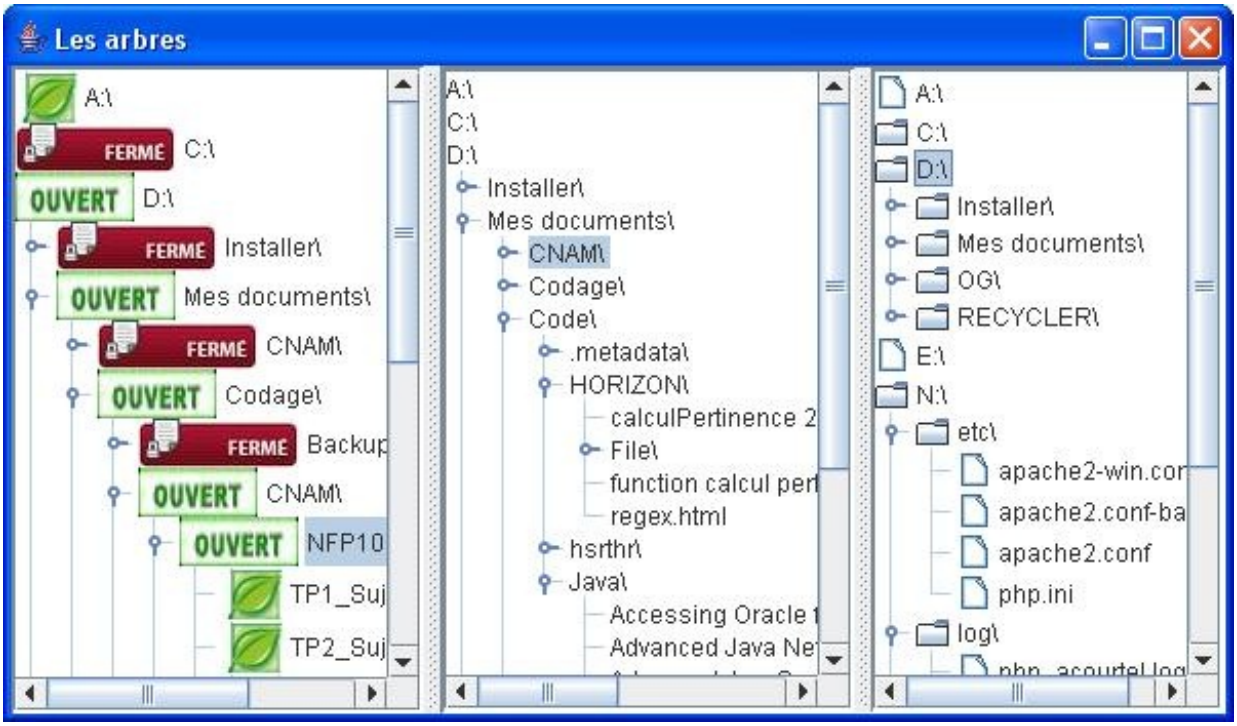
```
            subNode = new DefaultMutableTreeNode(nom.getName()+"\\");
            node.add(this.listFile(nom, subNode));
           }else{
            subNode = new DefaultMutableTreeNode(nom.getName());
           }
           node.add(subNode);
          }
         }
         return node;
        }
       }

       public static void main(String[] args){
        Fenetre fen = new Fenetre();
       }

      }
```

J'espère que vous n'avez pas trop eu de mal à faire ce petit exercice...
Vous devriez maintenant commencer à savoir utiliser ce type d'objet, mais, avant de passer à autre chose, je vous propose de voir comment personnaliser un peu l'affichage de notre arbre !
Retour en haut

Décorez vos arbres

Vous avez la possibilité de changer les icônes pour les répertoires, les fichiers, les icônes d'ouverture, les icônes de fermeture. Cette opération est très simple à réaliser : il vous suffit d'utiliser un objet **`DefaultTreeCellRenderer`** (qui est une sorte de modèle), de définir les icônes pour tous ces cas et, ensuite, de spécifier à votre arbre d'utiliser ce modèle en utilisant la méthode `setCellRenderer(DefaultTreeCellRenderer cellRenderer)` .

Voici un exemple qui vous montre trois rendus distincts :



Et voici le code que j'ai utilisé pour arriver à ce résultat :

Code : Java

```
import java.io.File;

import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JSplitPane;
import javax.swing.JTree;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeCellRenderer;


public class Fenetre extends JFrame {

 private JTree arbre, arbre2, arbre3;
 private DefaultMutableTreeNode racine;
 //On va créer deux modèles d'affichage
 private  DefaultTreeCellRenderer[] tCellRenderer = new
DefaultTreeCellRenderer[3];

 public Fenetre(){
  this.setSize(600, 350);
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setTitle("Les arbres");
  //On invoque la méthode de construction de notre arbre
  initRenderer();
  listRoot();

  this.setVisible(true);
```

```
        }
        /**
       * Méthode qui permet de créer des modèles d'affichage
       */
        private void initRenderer(){
         //Instanciation
         this.tCellRenderer[0] = new  DefaultTreeCellRenderer();
         //Initialisation des images pour les actions fermer, ouvrir et
      pour les feuilles
         this.tCellRenderer[0].setClosedIcon(new
      ImageIcon("img/fermer.jpg"));
         this.tCellRenderer[0].setOpenIcon(new
      ImageIcon("img/ouvert.jpg"));
         this.tCellRenderer[0].setLeafIcon(new
      ImageIcon("img/feuille.jpg"));

         this.tCellRenderer[1] = new  DefaultTreeCellRenderer();
         this.tCellRenderer[1].setClosedIcon(null);
         this.tCellRenderer[1].setOpenIcon(null);
         this.tCellRenderer[1].setLeafIcon(null);
        }


        private void listRoot(){

         this.racine = new DefaultMutableTreeNode();

         int count = 0;
         for(File file : File.listRoots())
         {
          DefaultMutableTreeNode lecteur = new
      DefaultMutableTreeNode(file.getAbsolutePath());
          try {
           for(File nom : file.listFiles()){
            DefaultMutableTreeNode node = new
      DefaultMutableTreeNode(nom.getName()+"\\");

            lecteur.add(this.listFile(nom, node));

           }
          } catch (NullPointerException e) {}

          this.racine.add(lecteur);

          //Si nous avons parcouru plus de 50 dossiers, on sort
          //if(count > 50) {break;}

         }
         //On crée, avec notre hiérarchie, un arbre
         arbre = new JTree(this.racine);
         arbre.setRootVisible(false);
         //On définit le rendu pour cet arbre
         arbre.setCellRenderer(this.tCellRenderer[0]);

         arbre2 = new JTree(this.racine);
         arbre2.setRootVisible(false);
         arbre2.setCellRenderer(this.tCellRenderer[1]);

         arbre3 = new JTree(this.racine);
         arbre3.setRootVisible(false);

         JSplitPane split = new JSplitPane( JSplitPane.HORIZONTAL_SPLIT,
           new JScrollPane(arbre2),
           new JScrollPane(arbre3));
         split.setDividerLocation(200);

         JSplitPane split2 = new JSplitPane( JSplitPane.HORIZONTAL_SPLIT,
           new JScrollPane(arbre),
           split);
         split2.setDividerLocation(200);
```

```
        this.getContentPane().add(split2);
      }

      private DefaultMutableTreeNode listFile(File file,
    DefaultMutableTreeNode node){
        int count = 0;

        if(file.isFile())
         return new DefaultMutableTreeNode(file.getName());
        else{
         for(File nom : file.listFiles()){
          count++;
          //pas plus de 5 enfants par noeud
          if(count < 5){
           DefaultMutableTreeNode subNode;
           if(nom.isDirectory()){
            subNode = new DefaultMutableTreeNode(nom.getName()+"\\");
            node.add(this.listFile(nom, subNode));
           }else{
            subNode = new DefaultMutableTreeNode(nom.getName());
           }
           node.add(subNode);
          }
         }
         return node;
        }
      }

      public static void main(String[] args){
       Fenetre fen = new Fenetre();
      }

    }
```

C'est simple, n'est-ce pas ?
Vous définissez les nouvelles images à utiliser et vous spécifiez à l'arbre quel modèle utiliser !

Il existe une autre façon de changer l'affichage (le design) de votre application.
Chaque système d'exploitation à son propre "***design***", mais vous avez pu constater que vos applications Java ne ressemblent pas du tout à ce que votre OS vous propose d'habitude !

C'est vrai que les couleurs ne sont pas comme d'habitude !

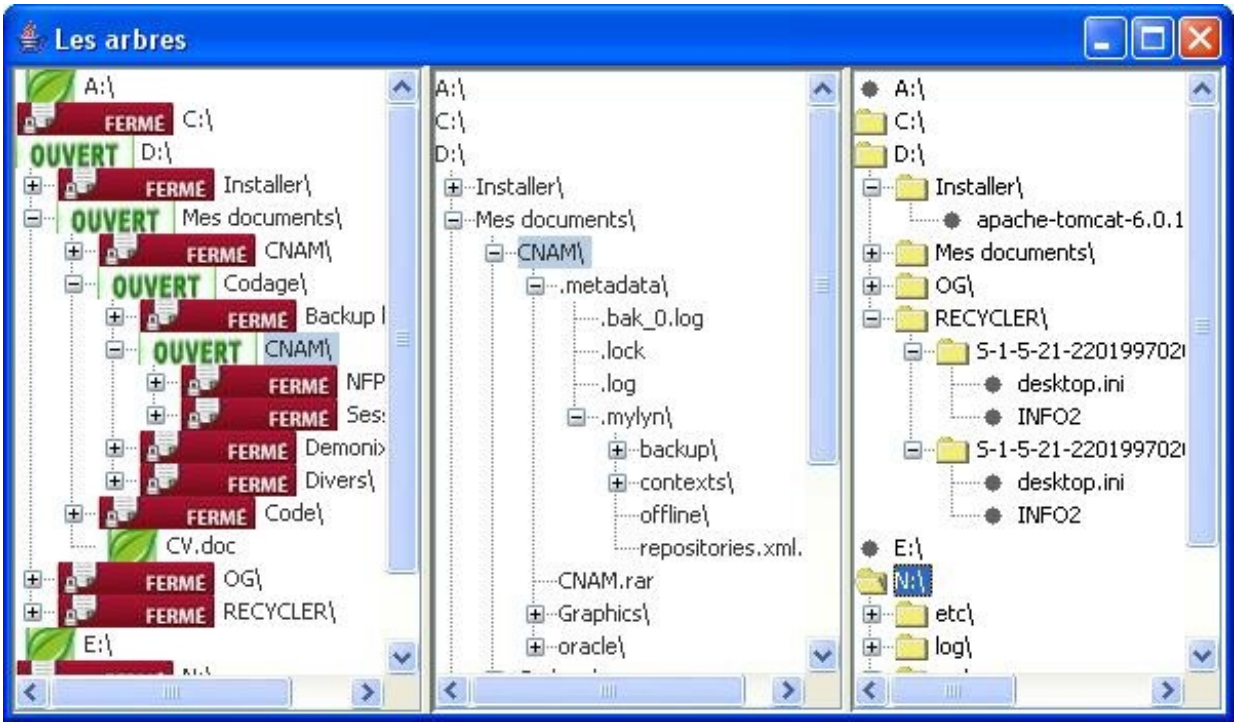
Les couleurs, mais aussi la façon dont sont dessinés vos composants...
Mais il y a un moyen de pallier ce problème : utiliser le "***look and feel***" de votre OS.

J'ai rajouté ces lignes de code dans le constructeur de mon objet, avant l'instruction `setVisible(true)` :

Code : Java

```
    try {
       //On force à utiliser le look and feel du system

    UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
       //Ici on force tous les composants de notre fenêtre (this) à se
    redessiner avec le look and feel du système
       SwingUtilities.updateComponentTreeUI(this);
    } catch (InstantiationException e) {
    } catch (ClassNotFoundException e) {
    } catch (UnsupportedLookAndFeelException e) {
    } catch (IllegalAccessException e) {}
```

Ce qui me donne avec l'exemple ci-dessus :



Vous pouvez, bien sûr, utiliser d'autres "look and feel" que ceux de votre système et de Java. Voici un code qui permet de lister ces types d'affichages et d'instancier un objet **Fenetre** en lui spécifiant quel modèle utiliser :

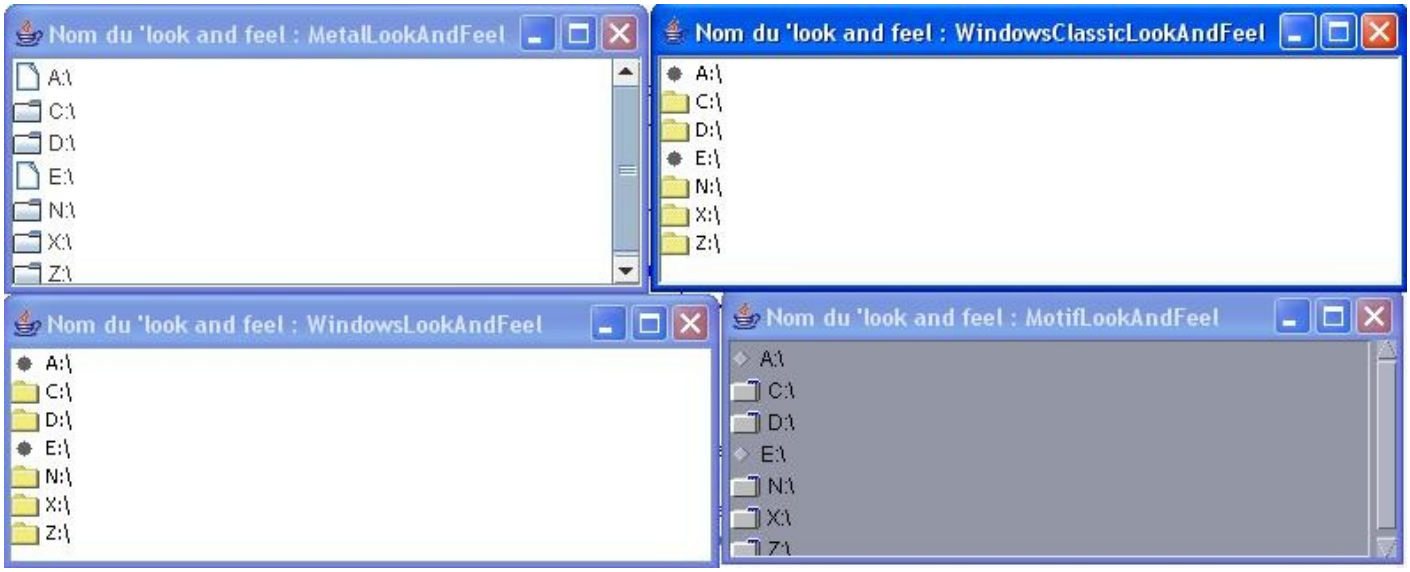
Code : Java

```
import java.io.File;

import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTree;
import javax.swing.SwingUtilities;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;
import javax.swing.tree.DefaultMutableTreeNode;


public class Fenetre extends JFrame {

 private JTree arbre, arbre2, arbre3;
 private DefaultMutableTreeNode racine;

 //On passe maintenant le nom du look and feel à utiliser en
paramètre du constructeur
 public Fenetre(String lookAndFeel){
  this.setSize(200, 300);
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  String title =
(lookAndFeel.split("\\."))[(lookAndFeel.split("\\.").length - 1)];
  this.setTitle("Nom du 'look and feel : " + title);
  //On invoque la méthode de construction de notre arbre

  listRoot();
  //On force l'utilisation
  try {
      UIManager.setLookAndFeel(lookAndFeel);
      SwingUtilities.updateComponentTreeUI(this);
   } catch (InstantiationException e) {
   } catch (ClassNotFoundException e) {
   } catch (UnsupportedLookAndFeelException e) {
```

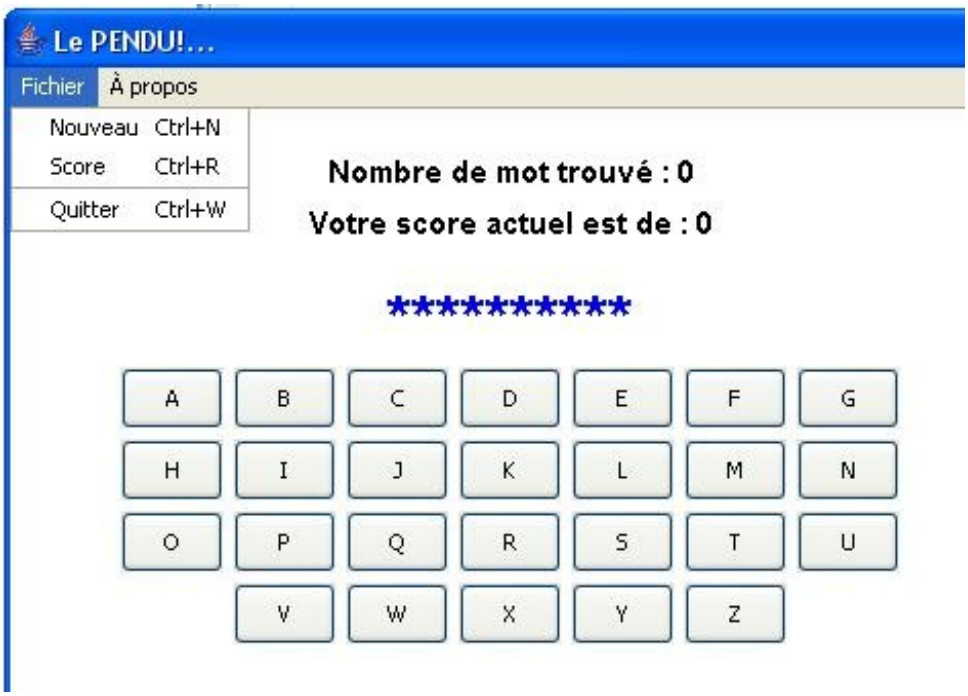
```
        } catch (IllegalAccessException e) {}
      this.setVisible(true);
     }

     private void listRoot(){

      this.racine = new DefaultMutableTreeNode();

      int count = 0;
      for(File file : File.listRoots())
      {
       DefaultMutableTreeNode lecteur = new
    DefaultMutableTreeNode(file.getAbsolutePath());
       try {
        for(File nom : file.listFiles()){
         DefaultMutableTreeNode node = new
    DefaultMutableTreeNode(nom.getName()+"\\");

         lecteur.add(this.listFile(nom, node));

        }
       } catch (NullPointerException e) {}

       this.racine.add(lecteur);

       //Si nous avons parcouru plus de 50 dossiers, on sort
       //if(count > 50) {break;}

      }
      //On crée, avec notre hiérarchie, un arbre
      arbre = new JTree(this.racine);
      arbre.setRootVisible(false);

      this.getContentPane().add(new JScrollPane(arbre));
     }

     private DefaultMutableTreeNode listFile(File file,
    DefaultMutableTreeNode node){
      int count = 0;

      if(file.isFile())
       return new DefaultMutableTreeNode(file.getName());
      else{
       for(File nom : file.listFiles()){
        count++;
        //pas plus de 5 enfants par noeud
        if(count < 5){
         DefaultMutableTreeNode subNode;
         if(nom.isDirectory()){
          subNode = new DefaultMutableTreeNode(nom.getName()+"\\");
          node.add(this.listFile(nom, subNode));
         }else{
          subNode = new DefaultMutableTreeNode(nom.getName());
         }
         node.add(subNode);
        }
       }
       return node;
      }
     }

     public static void main(String[] args){
      //nous allons créer des fenêtres avec des looks différents
      //Cette instruction permet de récupérer tous les looks du système
      UIManager.LookAndFeelInfo[] looks =
    UIManager.getInstalledLookAndFeels();
      Fenetre fen;
      //On parcourt tout le tableau en passant le nom du look à
    utiliser
      for(int i = 0; i < looks.length; i++)
```

```
            fen = new Fenetre(looks[i].getClassName());
        }
    }
```

J'ai capturé les fenêtres obtenues (le nom du look se trouve dans le titre de la fenêtre) :



Voici deux captures d'écran vous montrant le rendu obtenu en appliquant le look par défaut sur le pendu :



Vous pouvez voir que les menus et les boutons ressemblent déjà plus à ce que vous pouvez avoir sous Windows...
Et la boîte de dialogue, c'est flagrant :

Nous allons voir comment jouer un peu avec nos arbres... (non, pas sur nos arbres...)

Retour en haut

Jouons avec nos arbres

C'est maintenant que les choses compliquées vont commencer !

Par contre, il va falloir faire la lumière sur certaines choses...
Vous commencez à connaître les arbres, cependant je vous ai caché certaines choses afin de ne pas surcharger le début de ce chapitre.
Voyons : votre **JTree** est en fait composé de plusieurs objets. Vous vous doutez bien que pour un composant aussi complexe que celui-ci, il y a du monde au balcon...

Voici une liste des objets que vous serez susceptibles d'utiliser avec ce composant (il y a cinq interfaces et une classe concrète...) :

- **TreeModel** : c'est lui qui contient les noeuds de votre arbre ;
- **TreeNode** : noeuds et structure de votre arbre ;
- **TreeSelectionModel** : modèle de sélection de tous vos noeuds ;
- **TreePath** : objet qui vous permet de connaître le chemin d'un noeud dans l'arbre et voici notre classe concrète ;
- **TreeCellRenderer** : interface vous permettant de modifier l'apparence d'un noeud ;
- **TreeCellEditor** : éditeur utilisé lorsqu'un noeud est éditable.

Vous allez voir que, même si ces objets sont nombreux, leur utilisation, avec un peu de pratique, n'est pas si compliquée que ça...
Nous allons commencer par quelque chose d'assez simple : modifier le libellé d'un noeud !

C'est bon, on a déjà trouvé ! Il suffit d'invoquer la méthode `setEditable(Boolean bool)` de notre **JTree** !

Tout à fait !
Cependant, vous serez peut-être amenés à sauvegarder le nouveau nom de votre noeud... Il vous faudra donc déclencher un traitement lors de la modification d'un noeud.
Pour faire cela, nous allons utiliser l'objet **TreeModel** que nous allons écouter afin de déterminer ce qui se passe sur notre arbre !

Voici un exemple de code utilisant un **DefaultTreeModel** (classe implémentant l'interface **TreeModel**) ainsi qu'une implémentation de l'interface **TreeModelListener** afin d'écouter le dit objet :

Code : Java

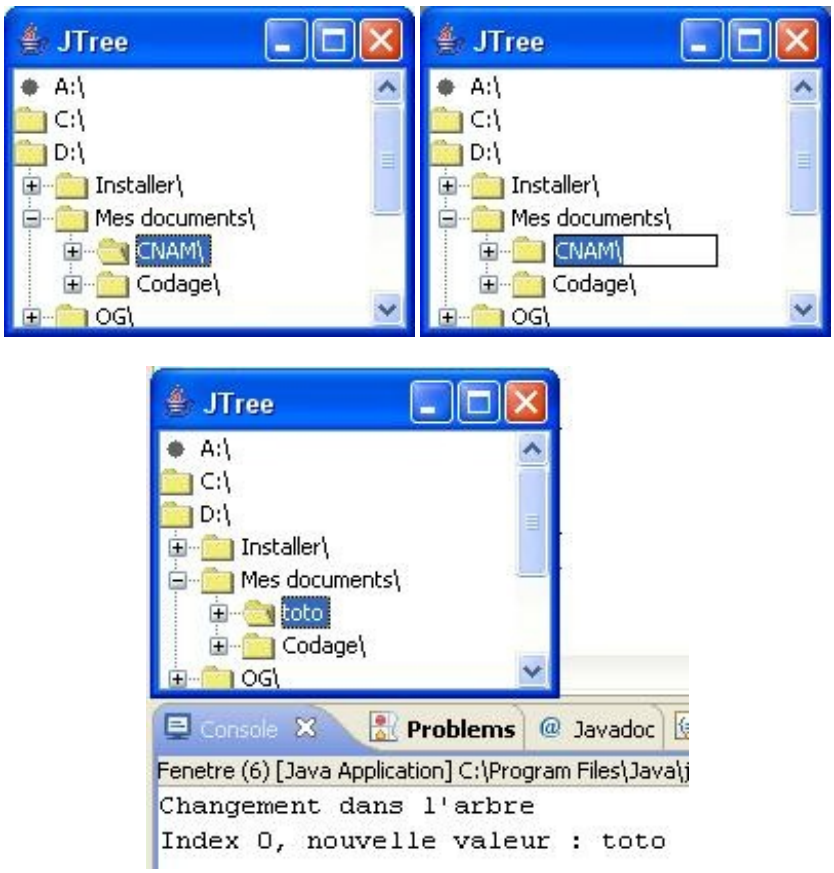
```
import java.awt.BorderLayout;
import java.io.File;

import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTree;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;
import javax.swing.event.TreeModelEvent;
import javax.swing.event.TreeModelListener;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;


public class Fenetre extends JFrame {

 private JTree arbre;
 private DefaultMutableTreeNode racine;
 //Notre objet modèle
 private DefaultTreeModel model;

 //On passe maintenant le nom du look and feel à utiliser en
paramètre du constructeur
 public Fenetre(){
  this.setSize(200, 300);
```

```
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setTitle("JTree");
        listRoot();
        this.setVisible(true);
      }

      private void listRoot(){

        this.racine = new DefaultMutableTreeNode();

        int count = 0;
        for(File file : File.listRoots())
        {
         DefaultMutableTreeNode lecteur = new
DefaultMutableTreeNode(file.getAbsolutePath());
         try {
          for(File nom : file.listFiles()){
           DefaultMutableTreeNode node = new
DefaultMutableTreeNode(nom.getName()+"\\");

           lecteur.add(this.listFile(nom, node));

          }
         } catch (NullPointerException e) {}
         this.racine.add(lecteur);
        }

        //On crée notre modèle
        this.model = new DefaultTreeModel(this.racine);
        //Et nous allons écouter ce que notre modèle a à nous dire !
        this.model.addTreeModelListener(new TreeModelListener() {
         /**
* Méthode appelée lorsqu'un noeud a changé
*/
          public void treeNodesChanged(TreeModelEvent evt) {
            System.out.println("Changement dans l'arbre");
            Object[] listNoeuds = evt.getChildren();
            int[] listIndices = evt.getChildIndices();
            for (int i = 0; i < listNoeuds.length; i++) {
              System.out.println("Index " + listIndices[i] + ",
nouvelle valeur : "
                  + listNoeuds[i]);
            }
          }
          /**
* Méthode appelée lorsqu'un noeud est inséré
*/
          public void treeNodesInserted(TreeModelEvent event) {
          System.out.println("Un noeud a été inséré !");

         }

          /**
* Méthode appelée lorsqu'un noeud est supprimé
*/
         public void treeNodesRemoved(TreeModelEvent event) {
          System.out.println("Un noeud a été retiré !");
         }
         /**
* Méthode appelée lorsque la structure d'un noeud a été modifiée
*/
         public void treeStructureChanged(TreeModelEvent event) {
          System.out.println("La structure d'un noeud a changé !");
         }
              });
        //On crée, avec notre hiérarchie, un arbre
        arbre = new JTree();
        //On passe notre modèle à notre arbre
        //===> on pouvait aussi passer l'objet TreeModel au constructeur
```

```
        du JTree
          arbre.setModel(model);
          arbre.setRootVisible(false);
          //On rend notre arbre éditable
          arbre.setEditable(true);
          this.getContentPane().add(new JScrollPane(arbre),
        BorderLayout.CENTER);
         }

         private DefaultMutableTreeNode listFile(File file,
        DefaultMutableTreeNode node){
          int count = 0;

          if(file.isFile())
           return new DefaultMutableTreeNode(file.getName());
          else{
           for(File nom : file.listFiles()){
            count++;
            //pas plus de 5 enfants par noeud
            if(count < 3){
             DefaultMutableTreeNode subNode;
             if(nom.isDirectory()){
              subNode = new DefaultMutableTreeNode(nom.getName()+"\\");
              node.add(this.listFile(nom, subNode));
             }else{
              subNode = new DefaultMutableTreeNode(nom.getName());
             }
             node.add(subNode);
            }
           }
           return node;
          }
         }

         public static void main(String[] args){
          //nous allons créer des fenêtres avec des looks différents
          //Cette instruction permet de récupérer tous les looks du système

          try {

        UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
          } catch (InstantiationException e) {
          } catch (ClassNotFoundException e) {
          } catch (UnsupportedLookAndFeelException e) {
          } catch (IllegalAccessException e) {}
          Fenetre fen = new Fenetre();

         }
        }
```

Afin de pouvoir changer le nom d'un noeud, vous devez double-cliquer sur un noeud avec un espace d'environ 1/2 seconde entre chaque clic... Si vous double-cliquez trop vite, vous déplierez le noeud ! !

Ce qui m'a donné chez moi :



Le dossier '**toto**' s'appelait '**CNAM/**' : vous pouvez voir que lorsque nous changeons le nom d'un noeud, la méthode `treeNodesChanged(TreeModelEvent evt)` est invoquée !

Vous voyez que, mis à part le fait que plusieurs objets sont utilisés, ce n'est pas si compliqué que ça...
Maintenant, je vous propose de voir comment ajouter des noeuds à notre arbre !
Pour ce faire, nous allons utiliser un bouton qui va nous demander de spécifier le nom du nouveau noeud, ceci via un **`JOptionPane`**.

Voici un code d'exemple :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.io.File;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPopupMenu;
import javax.swing.JScrollPane;
import javax.swing.JTree;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;
import javax.swing.event.TreeModelEvent;
import javax.swing.event.TreeModelListener;
import javax.swing.event.TreeSelectionEvent;
import javax.swing.event.TreeSelectionListener;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;
import javax.swing.tree.TreeNode;
import javax.swing.tree.TreePath;
```

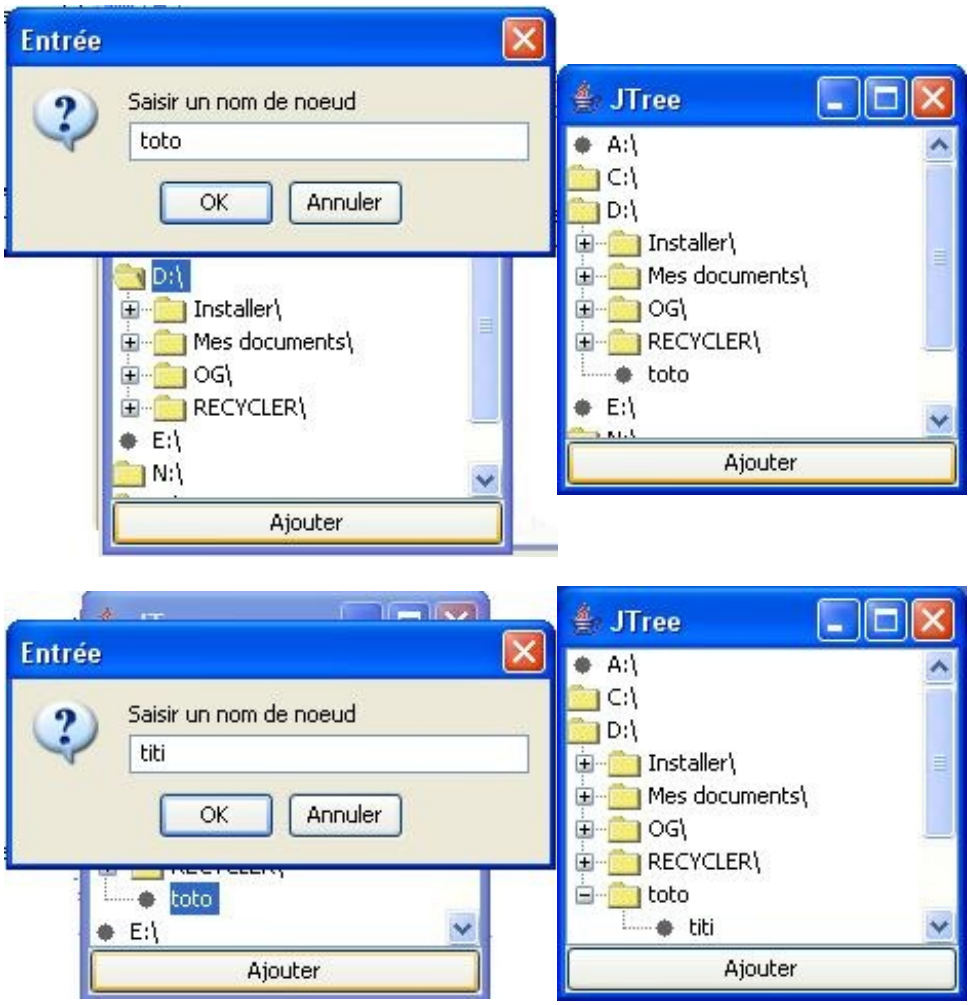
```
public class Fenetre extends JFrame {

 private JTree arbre;
 private DefaultMutableTreeNode racine;
 private DefaultTreeModel model;
 private JButton bouton = new JButton("Ajouter");

 //On passe maintenant le nom du look and feel à utiliser en
paramètre du constructeur
 public Fenetre(){
  this.setSize(200, 300);
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  this.setTitle("JTree");
  //On invoque la méthode de construction de notre arbre

  listRoot();
  bouton.addActionListener(new ActionListener(){

   public void actionPerformed(ActionEvent event) {
    if(arbre.getLastSelectedPathComponent() != null){
     JOptionPane jop = new JOptionPane();
     String nodeName = jop.showInputDialog("Saisir un nom de
noeud");

     if(nodeName != null && !nodeName.trim().equals("")){

      DefaultMutableTreeNode parentNode =
(DefaultMutableTreeNode)arbre.getLastSelectedPathComponent();
      DefaultMutableTreeNode childNode = new
DefaultMutableTreeNode(nodeName);
      parentNode.add(childNode);
      model.insertNodeInto(childNode, parentNode,
parentNode.getChildCount()-1);
      model.nodeChanged(parentNode);
     }
    }
    else{
     System.out.println("AUCUNE SELECTION ! ! !");
    }
   }
  });
  this.getContentPane().add(bouton, BorderLayout.SOUTH);
  this.setVisible(true);
 }

 private void listRoot(){

  this.racine = new DefaultMutableTreeNode();

  int count = 0;
  for(File file : File.listRoots())
  {
   DefaultMutableTreeNode lecteur = new
DefaultMutableTreeNode(file.getAbsolutePath());
   try {
    for(File nom : file.listFiles()){
     DefaultMutableTreeNode node = new
DefaultMutableTreeNode(nom.getName()+"\\");
     lecteur.add(this.listFile(nom, node));
    }
   } catch (NullPointerException e) {}

   this.racine.add(lecteur);

  }
  //On crée, avec notre hiérarchie, un arbre
  arbre = new JTree();
  this.model = new DefaultTreeModel(this.racine);
```

```
        arbre.setModel(model);
        arbre.setRootVisible(false);
        arbre.setEditable(true);
        arbre.getModel().addTreeModelListener(new TreeModelListener() {

                public void treeNodesChanged(TreeModelEvent evt) {
                  System.out.println("Changement dans l'arbre");
                  Object[] listNoeuds = evt.getChildren();
                  int[] listIndices = evt.getChildIndices();
                  for (int i = 0; i < listNoeuds.length; i++) {
                    System.out.println("Index " + listIndices[i] + ", noeud
déclencheur : "
                        + listNoeuds[i]);
                  }
                }
                public void treeNodesInserted(TreeModelEvent event) {
           System.out.println("Un noeud a été inséré !");
          }
          public void treeNodesRemoved(TreeModelEvent event) {
           System.out.println("Un noeud a été retiré !");
          }
          public void treeStructureChanged(TreeModelEvent event) {
           System.out.println("La structure d'un noeud a changé !");
          }
               });

        this.getContentPane().add(new JScrollPane(arbre),
BorderLayout.CENTER);
       }

       private DefaultMutableTreeNode listFile(File file,
DefaultMutableTreeNode node){
        int count = 0;

        if(file.isFile())
         return new DefaultMutableTreeNode(file.getName());
        else{
         for(File nom : file.listFiles()){
          count++;
          //pas plus de 5 enfants par noeud
          if(count < 3){
           DefaultMutableTreeNode subNode;
           if(nom.isDirectory()){
            subNode = new DefaultMutableTreeNode(nom.getName()+"\\");
            node.add(this.listFile(nom, subNode));
           }else{
            subNode = new DefaultMutableTreeNode(nom.getName());
           }
           node.add(subNode);
          }
         }
         return node;
        }
       }

       public static void main(String[] args){
        //nous allons créer des fenêtres avec des looks différents
        //Cette instruction permet de récupérer tous les looks du système

        try {

UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
         } catch (InstantiationException e) {
         } catch (ClassNotFoundException e) {
         } catch (UnsupportedLookAndFeelException e) {
         } catch (IllegalAccessException e) {}
        Fenetre fen = new Fenetre();

       }
```

```
    }
```

Vous remarquerez que nous avons ajouté des variables d'instances afin d'y avoir accès dans toute notre classe !

Voici quelques captures d'écran :



Là non plus, rien d'extraordinairement compliqué, mis à part cette portion de code :

Code : Java

```
parentNode =
(DefaultMutableTreeNode)arbre.getLastSelectedPathComponent();
DefaultMutableTreeNode childNode = new
DefaultMutableTreeNode(nodeName);
DefaultMutableTreeNode parentNode.add(childNode);
model.insertNodeInto(childNode, parentNode,
parentNode.getChildCount()-1);
model.nodeChanged(parentNode);
```

Tout d'abord, nous récupérons le dernier noeud sélectionné avec `parentNode = (DefaultMutableTreeNode)arbre.getLastSelectedPathComponent();` ; ensuite, nous créons un nouveau noeud avec `DefaultMutableTreeNode childNode = new DefaultMutableTreeNode(nodeName);` et nous l'ajoutons dans notre noeud parent avec l'instruction `parentNode.add(childNode);` , mais nous devons spécifier à notre modèle que celui-ci a un nouveau noeud et donc, qu'il a changé, ceci avec les instructions :

Code : Java

```
model.insertNodeInto(childNode, parentNode,
parentNode.getChildCount()-1);
model.nodeChanged(parentNode);
```

Et voilà !

Avant de vous laisser, il me reste encore à vous montrer... un code !
Celui-ci permet de retirer dynamiquement un noeud d'un arbre !
Inutile de le commenter, celui-ci est très clair :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.io.File;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JPopupMenu;
import javax.swing.JScrollPane;
import javax.swing.JTree;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;
import javax.swing.event.TreeModelEvent;
import javax.swing.event.TreeModelListener;
import javax.swing.event.TreeSelectionEvent;
import javax.swing.event.TreeSelectionListener;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;
import javax.swing.tree.TreeNode;
import javax.swing.tree.TreePath;


public class Fenetre extends JFrame {

 private JTree arbre;
 private DefaultMutableTreeNode racine;
 private DefaultTreeModel model;
 private JButton bouton = new JButton("Ajouter"),  effacer = new
JButton("Effacer");

 //On passe maintenant le nom du look and feel à utiliser en
paramètre du constructeur
 public Fenetre(){
  this.setSize(200, 300);
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  this.setTitle("JTree");
  //On invoque la méthode de construction de notre arbre

  listRoot();
  bouton.addActionListener(new ActionListener(){

   public void actionPerformed(ActionEvent event) {
    if(arbre.getLastSelectedPathComponent() != null){
     JOptionPane jop = new JOptionPane();
     String nodeName = jop.showInputDialog("Saisir un nom de
```

```
noeud");

      if(nodeName != null && !nodeName.trim().equals("")){
       DefaultMutableTreeNode parentNode =
(DefaultMutableTreeNode)arbre.getLastSelectedPathComponent();
       DefaultMutableTreeNode childNode = new
DefaultMutableTreeNode(nodeName);
       parentNode.add(childNode);
       model.insertNodeInto(childNode, parentNode,
parentNode.getChildCount()-1);
       model.nodeChanged(parentNode);
      }
     }
     else{
      System.out.println("AUCUNE SELECTION ! ! !");
     }
    }
   });

   effacer.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent event) {
     if(arbre.getLastSelectedPathComponent() != null){
      DefaultMutableTreeNode node =
(DefaultMutableTreeNode)arbre.getLastSelectedPathComponent();
      DefaultMutableTreeNode parentNode =
(DefaultMutableTreeNode)node.getParent();
      model.removeNodeFromParent(node);
      model.nodeChanged(parentNode);
     }
     else{
      System.out.println("AUCUNE SELECTION ! ! !");
     }
    }
   });

   JPanel pan = new JPanel();
   pan.add(bouton);
   pan.add(effacer);

   this.getContentPane().add(pan, BorderLayout.SOUTH);
   this.setVisible(true);
  }

  private void listRoot(){

   this.racine = new DefaultMutableTreeNode();

   int count = 0;
   for(File file : File.listRoots())
   {
    DefaultMutableTreeNode lecteur = new
DefaultMutableTreeNode(file.getAbsolutePath());
    try {
     for(File nom : file.listFiles()){
      DefaultMutableTreeNode node = new
DefaultMutableTreeNode(nom.getName()+"\\");
      lecteur.add(this.listFile(nom, node));
     }
    } catch (NullPointerException e) {}

    this.racine.add(lecteur);

   }
   //On crée, avec notre hiérarchie, un arbre
   arbre = new JTree();
   this.model = new DefaultTreeModel(this.racine);

   arbre.setModel(model);
   arbre.setRootVisible(false);
   arbre.setEditable(true);
```

```
        arbre.getModel().addTreeModelListener(new TreeModelListener() {

              public void treeNodesChanged(TreeModelEvent evt) {

              }
              public void treeNodesInserted(TreeModelEvent event) {
          System.out.println("Un noeud a été inséré !");
         }
         public void treeNodesRemoved(TreeModelEvent event) {
          System.out.println("Un noeud a été retiré !");
         }
         public void treeStructureChanged(TreeModelEvent event) {
          System.out.println("La structure d'un noeud a changé !");
         }
              });

        this.getContentPane().add(new JScrollPane(arbre),
      BorderLayout.CENTER);
       }

       private DefaultMutableTreeNode listFile(File file,
      DefaultMutableTreeNode node){
        int count = 0;

        if(file.isFile())
         return new DefaultMutableTreeNode(file.getName());
        else{
         for(File nom : file.listFiles()){
          count++;
          //pas plus de 5 enfants par noeud
          if(count < 3){
           DefaultMutableTreeNode subNode;
           if(nom.isDirectory()){
            subNode = new DefaultMutableTreeNode(nom.getName()+"\\");
            node.add(this.listFile(nom, subNode));
           }else{
            subNode = new DefaultMutableTreeNode(nom.getName());
           }
           node.add(subNode);
          }
         }
         return node;
        }
       }

       public static void main(String[] args){
        //nous allons créer des fenêtres avec des looks différents
        //Cette instruction permet de récupérer tous les looks du système

        try {

      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
         } catch (InstantiationException e) {
         } catch (ClassNotFoundException e) {
         } catch (UnsupportedLookAndFeelException e) {
         } catch (IllegalAccessException e) {}
        Fenetre fen = new Fenetre();

       }
      }
```

Ce qui me donne :

Avant toute suppression

Après suppression du dossier "Installer/"

Après suppression du dossier "D/"

Avant de clore ce chapitre, voici un petit cadeau.

Secret (cliquez pour afficher)

Voici comment utiliser un menu contextuel avec vos arbres.
Je n'ai implémenté qu'un menu contextuel pour effacer un noeud, mais la démarche est la même pour d'autres actions...

Code : Java

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
```

```
import java.io.File;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JPopupMenu;
import javax.swing.JScrollPane;
import javax.swing.JTree;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;
import javax.swing.event.TreeModelEvent;
import javax.swing.event.TreeModelListener;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;


public class Fenetre extends JFrame {

 private JTree arbre;
 private DefaultMutableTreeNode racine;
 private DefaultTreeModel model;
 private JButton bouton = new JButton("Ajouter"),  effacer = new
JButton("Effacer");
 private JMenuItem eraseMenu ;

 //On passe maintenant le nom du look and feel à utiliser en paramètre du
constructeur
 public Fenetre(){
  this.setSize(200, 300);
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  this.setTitle("JTree");
  //On invoque la méthode de construction de notre arbre

  listRoot();
  bouton.addActionListener(new ActionListener(){

   public void actionPerformed(ActionEvent event) {
    if(arbre.getLastSelectedPathComponent() != null){
     JOptionPane jop = new JOptionPane();
     String nodeName = jop.showInputDialog("Saisir un nom de noeud");

     if(nodeName != null && !nodeName.trim().equals("")){
      DefaultMutableTreeNode parentNode =
(DefaultMutableTreeNode)arbre.getLastSelectedPathComponent();
      DefaultMutableTreeNode childNode = new
DefaultMutableTreeNode(nodeName);
      parentNode.add(childNode);
      model.insertNodeInto(childNode, parentNode,
parentNode.getChildCount()-1);
      model.nodeChanged(parentNode);
     }
    }
    else{
     System.out.println("AUCUNE SELECTION ! ! !");
    }
   }
  });

  effacer.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent event) {
    if(arbre.getLastSelectedPathComponent() != null){
     DefaultMutableTreeNode node =
(DefaultMutableTreeNode)arbre.getLastSelectedPathComponent();
     DefaultMutableTreeNode parentNode =
(DefaultMutableTreeNode)node.getParent();
     model.removeNodeFromParent(node);
```

```
            model.nodeChanged(parentNode);
           }
           else{
            System.out.println("AUCUNE SELECTION ! ! !");
           }
          }
         });

         JPanel pan = new JPanel();
         pan.add(bouton);
         pan.add(effacer);

         this.getContentPane().add(pan, BorderLayout.SOUTH);
         this.setVisible(true);
        }

        private void listRoot(){

         this.racine = new DefaultMutableTreeNode();

         int count = 0;
         for(File file : File.listRoots())
         {
          DefaultMutableTreeNode lecteur = new
DefaultMutableTreeNode(file.getAbsolutePath());
          try {
           for(File nom : file.listFiles()){
            DefaultMutableTreeNode node = new
DefaultMutableTreeNode(nom.getName()+"\\");
            lecteur.add(this.listFile(nom, node));
           }
          } catch (NullPointerException e) {}

          this.racine.add(lecteur);

         }
         //On crée, avec notre hiérarchie, un arbre
         arbre = new JTree();
         this.model = new DefaultTreeModel(this.racine);

         arbre.setModel(model);
         arbre.setRootVisible(false);
         arbre.setEditable(true);
         arbre.getModel().addTreeModelListener(new TreeModelListener() {

                public void treeNodesChanged(TreeModelEvent evt) {

                }
                public void treeNodesInserted(TreeModelEvent event) {
           System.out.println("Un noeud a été inséré !");
          }
          public void treeNodesRemoved(TreeModelEvent event) {
           System.out.println("Un noeud a été retiré !");
          }
          public void treeStructureChanged(TreeModelEvent event) {
           System.out.println("La structure d'un noeud a changé !");
          }
               });

         /**
        * Effacer un noeud avec un menu contextuel ! !
        */
         arbre.addMouseListener(new MouseAdapter(){
          //Lorsque nous cliquons avec la souris
          public void mousePressed(MouseEvent event) {
           //Lors d'un clic droit
           if(event.getButton() == MouseEvent.BUTTON3){

            System.out.println("Clic droit détecté ! !");
```

```
        //Si on a cliqué sur l'arbre, on peut récupérer l'indice de la ligne
    avec cette méthode
        //Sinon, la méthode retourne -1
        if(arbre.getRowForLocation(event.getX(), event.getY()) != -1){

         //On peut récupérer le chemin du noeud qui a généré l'événement
         arbre.setSelectionPath(arbre.getPathForLocation(event.getX(),
    event.getY()));
         System.out.println("Noeud sélectionné : " +
    ((DefaultMutableTreeNode)arbre.getLastSelectedPathComponent()).toString());

         //On peut donc en déduire le noeud et le parent
         DefaultMutableTreeNode node =
    (DefaultMutableTreeNode)arbre.getLastSelectedPathComponent();
         DefaultMutableTreeNode parentNode =
    (DefaultMutableTreeNode)node.getParent();

         //On n'a plus qu'à générer notre menu contextuel !
         JPopupMenu jpm = new JPopupMenu();
         eraseMenu = new JMenuItem("Effacer ce noeud");
         eraseMenu.addActionListener(new EraseMenuListener(parentNode, node));
         jpm.add(eraseMenu);
         jpm.show(arbre, event.getX(), event.getY());
         //Et le tour est joué ! ! !
        }
       }
      }
     });

     this.getContentPane().add(new JScrollPane(arbre), BorderLayout.CENTER);
    }

    private DefaultMutableTreeNode listFile(File file, DefaultMutableTreeNode
    node){
     int count = 0;

     if(file.isFile())
      return new DefaultMutableTreeNode(file.getName());
     else{
      for(File nom : file.listFiles()){
       count++;
       //pas plus de 5 enfants par noeud
       if(count < 3){
        DefaultMutableTreeNode subNode;
        if(nom.isDirectory()){
         subNode = new DefaultMutableTreeNode(nom.getName()+"\\");
         node.add(this.listFile(nom, subNode));
        }else{
         subNode = new DefaultMutableTreeNode(nom.getName());
        }
        node.add(subNode);
       }
      }
      return node;
     }
    }

    /**
    * Listener personnalisé pour le menu contextuel
    * @author CHerby
    */
    class EraseMenuListener implements ActionListener{

     //Nous allons nous servir de ces deux objets
     DefaultMutableTreeNode  parentNode, node;
     public EraseMenuListener(DefaultMutableTreeNode parent,
    DefaultMutableTreeNode node){
      this.parentNode = parent;
      this.node = node;
     }
```

```
        //Ici, vous connaissez !
        public void actionPerformed(ActionEvent e) {
         model.removeNodeFromParent(this.node);
         model.nodeChanged(this.parentNode);
        }
       }


      public static void main(String[] args){
       //nous allons créer des fenêtres avec des looks différents
       //Cette instruction permet de récupérer tous les looks du système

       try {
           UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (InstantiationException e) {
        } catch (ClassNotFoundException e) {
        } catch (UnsupportedLookAndFeelException e) {
        } catch (IllegalAccessException e) {}
       Fenetre fen = new Fenetre();

      }
     }
```

Le résultat est sympa :

Voilà : je pense que vous en savez assez pour utiliser les arbres dans vos futures applications !
Je vous propose donc un petit tour sur le topo des familles.
Retour en haut

Ce que vous devez retenir

- Les arbres sont une combinaison entre des **`DefaultMutableTreeNode`** et des **`JTree`**.
- Vous pouvez masquer le répertoire "***racine***" en invoquant la méthode `setRootVisible(Boolean ok)` .
- Afin d'intercepter les événements sur un tel composant, vous devrez implémenter l'interface **`TreeSelectionListener`**.
- Cette interface n'a qu'une méthode à redéfinir : `public void valueChanged(TreeSelectionEvent event)` .
- L'affichage des différents éléments constituant un arbre peut être modifié en utilisant un **`DefaultTreeCellRenderer`**. Définissez et affectez cet objet à l'arbre pour personnaliser l'affichage de votre arbre.
- Vous pouvez aussi changer le "***look and feel***" et utiliser celui de votre OS.

Retour en haut

Plus de peur que de mal, même si c'est un objet compliqué à utiliser.
Mais maintenant que vous êtes chaud, nous continuons avec un objet tout aussi compliqué (si ce n'est pas plus) avec : **les tableaux**.

Retour en haut

Les tableaux, les vrais

Nous continuons notre descente aux enfers avec un autres composant assez complexe.
Il a été une de mes bêtes noires pendant quelques temps : j'espère donc que ce chapitre vous permettra de vous simplifier son apprentissage.

Bon, je vous propose de commencer dès maintenant !
Je vous le promets : ça ne fera pas mal... (ou peut-être un peu).

Premiers pas

Bon. Les tableaux sont des composants qui permettent d'afficher des données de façon structurée : dans un tableau.

Tout le monde sait ce qu'est un tableau.
Non ? Vous me charriez, là... En voici un :

Le code source de ce programme est :

Code : Java

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;


public class Fenetre extends JFrame {

 public Fenetre(){
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setTitle("JTable");
  this.setSize(300, 120);

                //Les données du tableau
  Object[][] data = { {"Cysboy", "28 ans", "1.80 cm"},
    {"BZHHydde", "28 ans", "1.80 cm"},
    {"IamBow", "24 ans", "1.90 cm"},
    {"FunMan", "32 ans", "1.85 cm"}
  };
        //Les titres des colonnes
  String  title[] = {"Pseudo", "Age", "Taille"};
  JTable tableau = new JTable(data, title);
                //On ajoute notre tableau à notre contentPane dans
un scroll
                //Sinon les titres des colonnes ne s'afficheront pas
! !
  this.getContentPane().add(new JScrollPane(tableau));
 }


 public static void main(String[] args){
  Fenetre fen = new Fenetre();
  fen.setVisible(true);
 }

}
```

Code simple et clair !
Vous instanciez un objet **`JTable`** en lui passant en paramètres les données qu'il doit utiliser.

Attention : le tableau correspondant aux titres des colonnes de votre tableau peuvent être des **`String`** ou de type **`Object`** ! ! Tandis que les données, elles, sont des **`Object`**.

Pourquoi les données doivent-elles être de type `Object` ?

Vous allez voir, un peu plus loin, que nous pouvons mettre plusieurs types d'éléments dans un tableau ! Mais l'heure n'est pas encore venue pour ce genre de choses... Il vous faut voir comment fonctionne cet objet avant cela !

Je comprends que vous soyez pressés mais tout vient à point à qui sait attendre.

Bon, d'accord. Mais pourquoi tu as mis le tableau dans un scroll ?

En fait, ceux qui ont essayé d'ajouter le tableau dans le `contentPane` sans scroll ont dû voir que **les titres de colonnes n'apparaissent pas**...
Le scroll indique automatiquement à notre tableau où il doit afficher ses titres !
Si vous ne faites pas ceci, **vous êtes OBLIGÉS de spécifier où mettre l'entête du tableau**, comme ceci :

Code : Java

```
//On indique que l'entête doit être au nord, donc au-dessus
this.getContentPane().add(tableau.getTableHeader(),
BorderLayout.NORTH);
//Et le corps au centre !
this.getContentPane().add(tableau, BorderLayout.CENTER);
```

Bon, je pense qu'on a fait le tour des préliminaires... Rentrons dans le vif du sujet !

Retour en haut

Les cellules, c'est la vie

Vos tableaux sont composés de ce qu'on appelle : **des cellules !**
Vous pouvez les voir facilement, elles sont encadrées par des bordures noires et contiennent les données que vous avez mises dans le tableau d'`Object` et de `String`.

Celles-ci peuvent être retrouvées par des coordonnées (x, y) où x correspond au numéro de ligne et y au numéro de la colonne ! Une cellule est donc une intersection d'une ligne et d'une colonne.

Donc, afin de pouvoir modifier une cellule, nous allons devoir récupérer la ligne et la colonne auxquelles elle appartient ?

Tout à fait... Et là... Vous devez avoir à peu près cette tête-là :
Ne vous inquiétez pas, nous allons prendre tout ça point par point.

Tout d'abord, nous allons commencer par changer la taille d'une colonne et d'une ligne, ce qui nous donne, au final, quelque chose comme ça :

Vous allez voir que le code utilisé est simple comme tout, encore fallait-il que vous sachiez quelles méthodes et quels objets utiliser... Voici le code permettant d'obtenir ce résultat :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.TableColumn;


public class Fenetre extends JFrame {

 private JTable tableau;
 private JButton change = new JButton("Changer la taille");
 private JButton retablir = new JButton("Rétablir");

 public Fenetre(){
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setTitle("JTable");
  this.setSize(300, 240);

  Object[][] data = { {"Cysboy", "28 ans", "1.80 cm"},
```

```
        {"BZHHydde", "28 ans", "1.80 cm"},
        {"IamBow", "24 ans", "1.90 cm"},
        {"FunMan", "32 ans", "1.85 cm"}
      };

      String  title[] = {"Pseudo", "Age", "Taille"};
      this.tableau = new JTable(data, title);

      JPanel pan = new JPanel();

      change.addActionListener(new ActionListener(){
       public void actionPerformed(ActionEvent arg0) {
        changeSize(200, 80);
        change.setEnabled(false);
        retablir.setEnabled(true);
       }
      });

      retablir.addActionListener(new ActionListener(){
       public void actionPerformed(ActionEvent arg0) {

        changeSize(75, 16);
        change.setEnabled(true);
        retablir.setEnabled(false);
       }
      });

      retablir.setEnabled(false);
      pan.add(change);
      pan.add(retablir);

      //On remplace cette ligne
      this.getContentPane().add(new JScrollPane(tableau),
    BorderLayout.CENTER);
      this.getContentPane().add(pan, BorderLayout.SOUTH);

     }

     /**
    * Change la taille d'une ligne et d'une colonne
    * J'ai mis deux boucles afin que vous puissiez voir comment
    parcourir les colonnes et les lignes
    * @param width
    * @param height
    */
     public void changeSize(int width, int height){
      //On crée un objet TableColumn afin de travailler sur notre
    colonne
      TableColumn col;
      for(int i = 0; i < tableau.getColumnCount(); i++){
       if(i == 1){
        //On récupère le modèle de la colonne
        col = tableau.getColumnModel().getColumn(i);
        //On lui affecte la nouvelle valeur
        col.setPreferredWidth(width);
       }
      }
      for(int i = 0; i < tableau.getRowCount(); i++){
       //On affecte la taille de la ligne à l'indice spécifié !
       if(i == 1)
        tableau.setRowHeight(i, height);
      }
     }

     public static void main(String[] args){
      Fenetre fen = new Fenetre();
      fen.setVisible(true);
     }

    }
```

Tout comme pour les tableaux vus dans la première partie de ce tuto, les indices de colonnes et de lignes d'un **JTable** commencent à 0 ! !

J'ai mis deux boucles afin que vous puissiez voir comment parcourir les colonnes et les lignes de votre objet mais, ici, nous aurions pu nous contenter d'un indice mis en dur dans notre code...

Ceci mis à part, vous constatez que la ligne et la colonne concernées changent bien de tailles lors du clic sur les boutons ! Vous venez donc de voir comment changer la taille des cellules de façon dynamique.

Je dis ça parce que, au cas où vous ne l'auriez pas remarqué, vous pouvez changer la taille des colonnes manuellement. Il vous suffit de cliquer sur un séparateur de colonne, de maintenir le clic et de déplacer le dit séparateur :

D'accord, on a bien compris !
Par contre, cette instruction nous semble bizarre : `tableau.getColumnModel().getColumn(i);` . Tu pourrais creuser un peu ?

J'allais y venir, mais le fait que vous ayez remarqué ceci est bien la preuve que vous commencez à vous poser les bonnes questions !

En fait, vous devez savoir qu'il y a un objet qui fait le lien entre votre tableau et vos données. Celui-ci est un ce qu'on appelle un modèle de tableau (ça vous rappelle les modèles d'arbres, non ?).
L'objet en question s'appelle **JTableModel** et vous allez voir qu'il permet de faire des choses très intéressantes ! !

Par exemple, vous avez vu que vous pouvez mettre des données de n'importe quel type héritant de la classe **Object**.

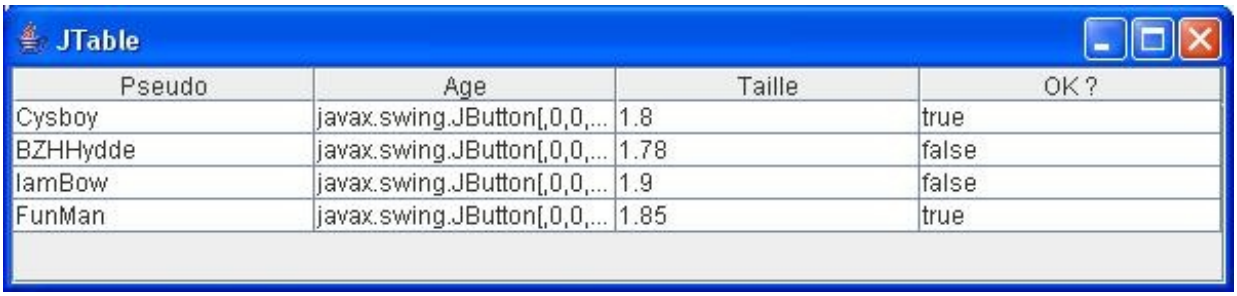
Essayez ce morceau de code :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.TableColumn;


public class Fenetre extends JFrame {

 private JTable tableau;
 private JButton change = new JButton("Changer la taille");

 public Fenetre(){
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setTitle("JTable");
  this.setSize(600, 140);

  Object[][] data = { {"Cysboy", new JButton("6boy"), new
Double(1.80), new Boolean(true)}
```

```
Double(1.80), new Boolean(true)},
        {"BZHHydde", new JButton("BZH"), new Double(1.78), new
Boolean(false)},
        {"IamBow", new JButton("BoW"), new Double(1.90), new
Boolean(false)},
        {"FunMan", new JButton("Year"), new Double(1.85), new
Boolean(true)}
    };

    String  title[] = {"Pseudo", "Age", "Taille", "OK ?"};
    this.tableau = new JTable(data, title);
    this.getContentPane().add(new JScrollPane(tableau),
BorderLayout.CENTER);

  }

  public static void main(String[] args){
    Fenetre fen = new Fenetre();
    fen.setVisible(true);
  }
}
```

Ce code doit vous donner ceci :



Maintenant, vous allez créer votre modèle de tableau !
Pour cela, il vous suffit de créer une classe héritant de **`AbstractTableModel`** qui, vous l'avez sûrement deviné, est une classe abstraite...

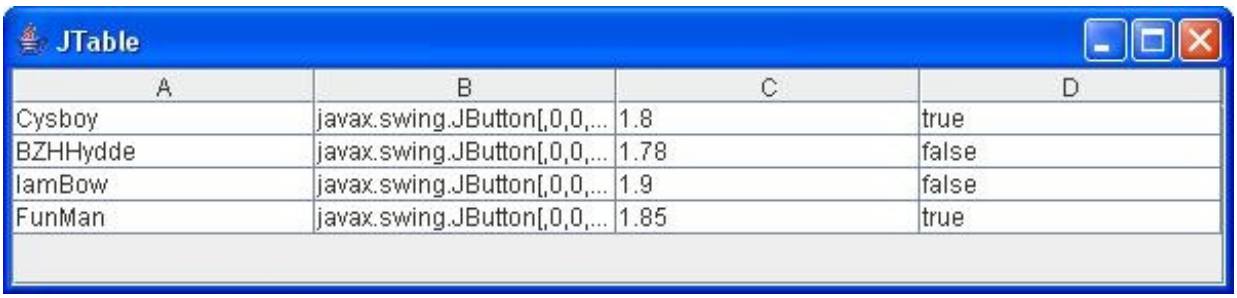
Voici donc un code mettant en pratique mes dires :

Code : Java

```
import java.awt.BorderLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;


public class Fenetre extends JFrame {

 private JTable tableau;
 private JButton change = new JButton("Changer la taille");

 public Fenetre(){
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setTitle("JTable");
  this.setSize(600, 140);

    Object[][] data = { {"Cysboy", new JButton("6boy"), new
Double(1.80), new Boolean(true)},
```

```
            {"BZHHydde", new JButton("BZH"), new Double(1.78), new
        Boolean(false)},
            {"IamBow", new JButton("BoW"), new Double(1.90), new
        Boolean(false)},
            {"FunMan", new JButton("Year"), new Double(1.85), new
        Boolean(true)}
          };

          String  title[] = {"Pseudo", "Age", "Taille", "OK ?"};

          ZModel model = new ZModel(data, title);
          System.out.println("Nombre de colonne : " +
        model.getColumnCount());
          System.out.println("Nombre de ligne : " + model.getRowCount());
          this.tableau = new JTable(model);
          this.getContentPane().add(new JScrollPane(tableau),
        BorderLayout.CENTER);

         }

                //CLASSE MODÈLE PERSONNALISÉE
         class ZModel extends AbstractTableModel{

          private Object[][] data;
          private String[] title;
          /**
        * Constructeur
        * @param data
        * @param title
        */
          public ZModel(Object[][] data, String[] title){
           this.data = data;
           this.title = title;
          }

          /**
        * Retourne le nombre de colonnes
        */
          public int getColumnCount() {
           return this.title.length;
          }

          /**
        * Retourne le nombre de lignes
        */
          public int getRowCount() {
           return this.data.length;
          }

          /**
        * Retourne la valeur à l'emplacement spécifié
        */
          public Object getValueAt(int row, int col) {
           return this.data[row][col];
          }

         }

         public static void main(String[] args){
          Fenetre fen = new Fenetre();
          fen.setVisible(true);
         }
        }
```

Ce qui nous donne ceci :



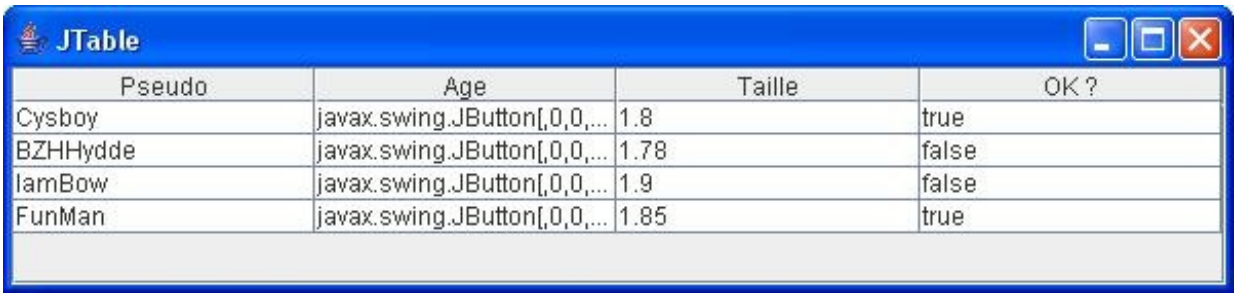
Eh ! ! On n'a même plus les titres de nos colonnes !

Effectivement. Ceci est dû au fait que vous n'avez redéfini que les méthodes abstraites de la classe **`AbstractTableModel`**. Afin de voir nos titres de colonnes apparaître, il faut redéfinir la méthode `getColumnName(int col)` , elle devrait ressembler à ceci :

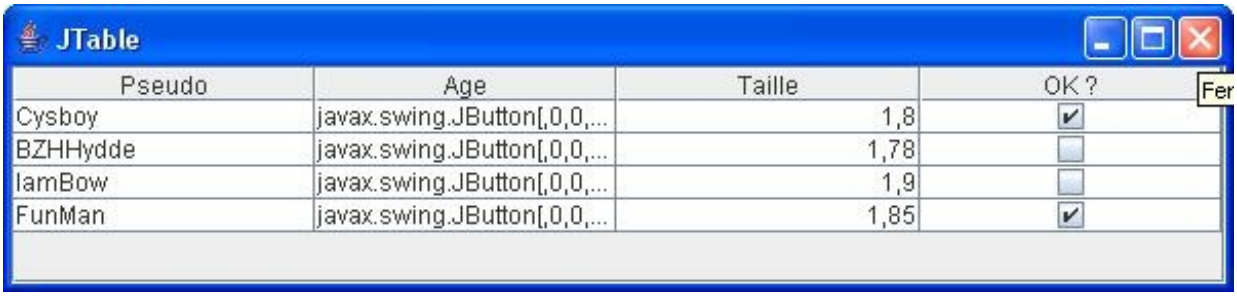
Code : Java

```
/**
* Retourne le titre de la colonne à l'indice spécifé
*/
public String getColumnName(int col) {
  return this.title[col];
}
```

Ré-exécutez votre code, après avoir rajouté cette méthode dans votre objet **`ZModel`**, et vous devriez avoir ceci :



Vous commencez à faire des choses sympas !
Je vais vous montrer un autre truc marrant avec les modèles.
Regardez ce que vous pouvez obtenir :



Vous avez vu ? Vos booléens se sont transformés en cases à cocher !
Les booléens valant *vrai* sont devenus des cases cochées et les booléens valant *faux* sont maintenant des cases non cochées !

Comment tu as fait ça ?

D'une manière toute simple, j'ai redéfini une méthode dans mon modèle et c'est automatique !
Cette méthode permet de retourner la classe du type de valeur d'un modèle et de transformer vos booléens en cases à cocher...
Au moment où notre objet crée le rendu des cellules, il invoque cette méthode et se sert de cette dernière pour créer certaines choses, comme ce que vous venez de voir.

Pour avoir ce rendu, il vous suffit de redéfinir la méthode `getColumnClass(int col)` . Cette méthode retourne un objet **`Class`**.
Je vous laisse réfléchir un peu sur le "***comment renvoyer cette valeur***".
Pour voir comment faire, c'est juste en dessous :

Secret (cliquez pour afficher)

Code : Java

```
/**
* Retourne la classe de la donnée de la colonne
* @param col
*/
public Class getColumnClass(int col){
 //On retourne le type de la cellule à la colonne demandée
 //On se moque de la ligne puisque les données sur chaque ligne sont les mêmes
 //On choisit donc la première ligne
 return this.data[0][col].getClass();
}
```

Je ne sais pas si vous avez remarqué mais vos cellules ne sont plus éditables !

Oh non ! Qu'est-ce qui se passe ?

Je vous avait prévenus que ce composant était pénible...
En fait, vous devez aussi dire à votre modèle d'avertir votre **`JTable`** que certaines cellules peuvent être éditées et d'autres non (comme le bouton, par exemple).

Afin de réussir ceci, vous devez redéfinir la méthode `isCellEditable(int row, int col)` qui, dans la classe mère retourne tout le temps **`false`** ...
Rajouter donc ce morceau de code dans notre modèle :

Code : Java

```
/**
* Retourne vrai si la cellule est éditable : celle-ci sera donc éditable
* @return boolean
*/
public boolean isCellEditable(int row, int col){
 return true;
}
```

Vos cellules sont de nouveau éditables, cependant, vous n'avez pas dit à votre modèle que la cellule contenant votre bouton ne devait pas être éditable...

T'es gentil, mais comment on fait ça ?

Grâce à la méthode `getClass()` de tout objet Java !

Vous pouvez déterminer de quelle classe est issu votre objet et, pour faire la comparaison avec une classe, on utilise le mot clé **instanceof**.
Regardez comment on procède, c'est ~~clore comme de l'eau de ruche~~ clair comme de l'eau de roche :

Code : Java

```
/**
* Retourne vrai si la cellule est éditable : celle-ci sera donc
éditable
* @return boolean
*/
public boolean isCellEditable(int row, int col){
 //On appelle la méthode getValueAt qui retourne la valeur d'une
cellule
 //Et on fait un traitement spécifique si c'est un JButton
 if(getValueAt(0, col) instanceof JButton)
  return false;
 return true;
}
```

Voilà ! ! !
Victoire ! Les cellules sont de nouveau éditables sauf le **JButton** !
Par contre, je vois que certains d'entre vous attendent toujours de voir apparaître un zoli bouton...
Pour réussir à faire ceci, nous n'allons pas utiliser un modèle de tableau, mais un objet qui s'occupe de gérer le contenu et la façon dont ce contenu est affiché.

Les modèles servent à faire un pont entre ce qu'affiche **JTable** et ce que fait l'utilisateur. Pour changer la façon dont sont affichées les cellules, nous devrons utiliser **DefaultCellRenderer**.
Retour en haut

Contrôlez l'affichage de vos cellules

Bon, je ne vais pas vous faire languir plus longtemps...
Cependant, vous devrez faire la distinction entre un **TableModel** et un **DefaultTableCellRenderer**.
Le premier fait le lien entre vous et le tableau tandis que le second s'occupe de gérer l'affichage dans les cellules !

Le but du jeu est de définir une nouvelle façon de dessiner les composants dans les cellules.
En définitive, nous n'allons pas vraiment faire ça, mais nous allons dire à notre tableau que la valeur qu'il a dans telle ou telle cellule est un composant (bouton ou autre).

Rien de plus simple à faire, encore fallait-il le savoir.
Il suffit de créer une classe héritant de **DefaultTableCellRenderer** et de redéfinir la méthode `public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column)` .

Waw ! Il y en a des paramètres !

En effet, mais dans le cas qui nous intéresse, nous n'en avons besoin que d'un seul : **value**.
Vous voyez que cette méthode retourne un objet **Component**.
Nous allons seulement spécifier de quel type d'objet il s'agit suivant le cas...

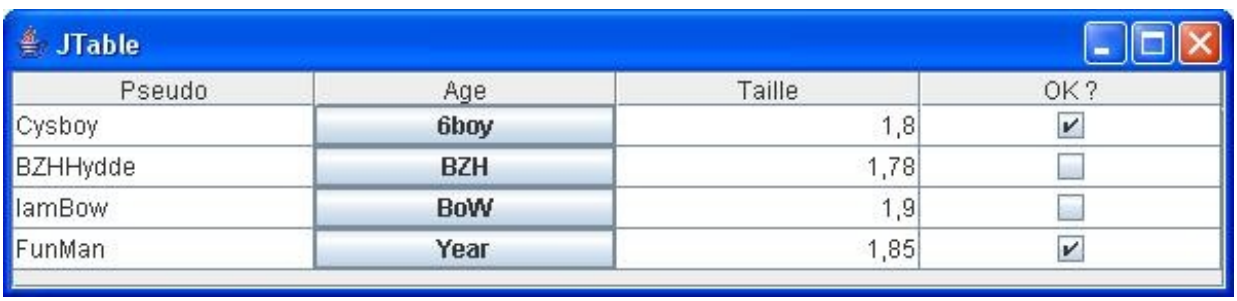
Regarder notre classe héritée :

Code : Java

```
import java.awt.Component;
import javax.swing.JButton;
import javax.swing.JTable;
import javax.swing.table.DefaultTableCellRenderer;

public class TableComponent extends DefaultTableCellRenderer {

 public Component getTableCellRendererComponent(JTable table,
   Object value, boolean isSelected, boolean hasFocus, int row,
   int column) {
  //Si la valeur de la cellule est un JButton, on transtype notre
valeur
  if (value instanceof JButton){
   return (JButton) value;
  }
  else
   return this;
 }
}
```

Une fois cette classe créée, il suffit juste de dire à notre tableau d'utiliser ce rendu de cellules, avec cette instruction :
`this.tableau.setDefaultRenderer(JButton.class, new TableComponent());` .
Le premier paramètre permet de dire à notre tableau de faire attention à ce type d'objet et enfin, le second lui dit d'utiliser ce modèle de cellules.

Ce qui nous donne :



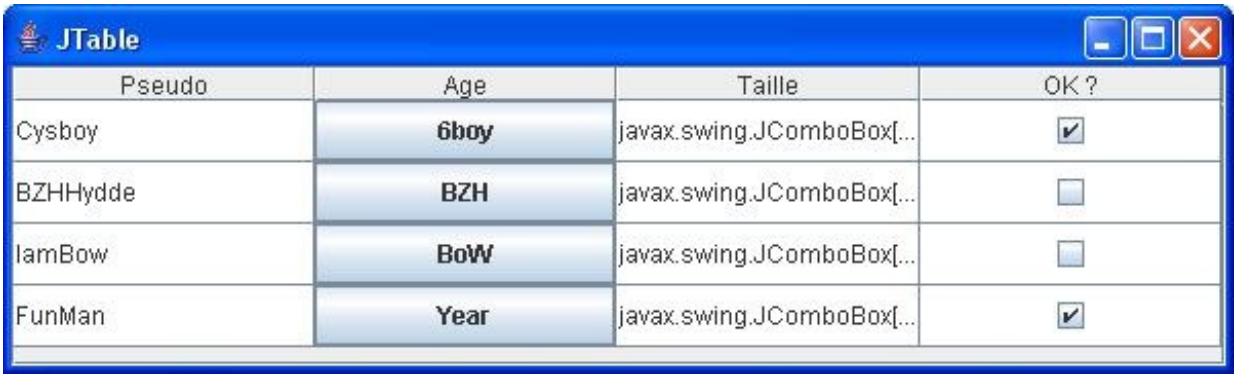
Voilà notre bouton en chair et en os !
Je me doute bien que les plus taquins d'entre vous ont dû essayer de mettre plus d'un type de composant dans le tableau... Et vous vous retrouvez le bec dans l'eau car il ne prend en compte que les boutons pour le moment...

En fait, une fois que vous avez défini une classe héritée afin de gérer le rendu de vos cellules, vous avez fait le plus gros du travail...
Tenez, si, par exemple, nous avons ce genre de données à mettre dans notre tableau :

Code : Java

```
Object[][] data = { {"Cysboy", new JButton("6boy"), new
JComboBox(new String[]{"toto", "titi", "tata"}), new Boolean(true)},
   {"BZHHydde", new JButton("BZH"), new JComboBox(new
String[]{"toto", "titi", "tata"}), new Boolean(false)},
   {"IamBow", new JButton("BoW"), new JComboBox(new String[]{"toto",
"titi", "tata"}), new Boolean(false)},
   {"FunMan", new JButton("Year"), new JComboBox(new
String[]{"toto", "titi", "tata"}), new Boolean(true)}
  };
```

et si nous conservons l'objet de rendu de cellule tel qu'il est actuellement, nous aurons ceci :



J'ai augmenté un peu la taille de la fenêtre et des lignes... Mis à part ça, rien n'a changé !

Les boutons s'affichent toujours mais pas les combos !

Comment faire, alors ?

Je sais que certains d'entre vous ont presque trouvé la solution, j'en suis même sûr.
Vous n'auriez pas ajouté ce qui suit dans votre objet de rendu de cellule ?

Code : Java

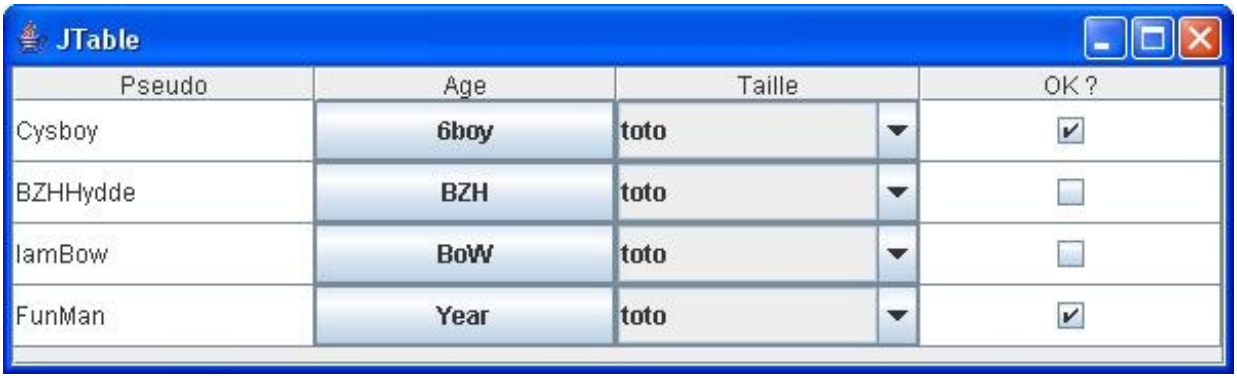
```
import java.awt.Component;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JTable;
import javax.swing.table.DefaultTableCellRenderer;


public class TableComponent extends DefaultTableCellRenderer {

 public Component getTableCellRendererComponent(JTable table,
   Object value, boolean isSelected, boolean hasFocus, int row,
   int column) {
```

```
        if (value instanceof JButton){
         return (JButton) value;
        }
        //LIGNE RAJOUTÉE
        else if(value instanceof JComboBox){
         return (JComboBox) value;
        }
        else
         return this;
      }
    }
```

Ceux qui ont fait ceci ont trouvé la première partie de la solution !
Vous avez bien spécifié à votre objet de retourner une valeur *castée* en cas de rencontre avec une combo !
Seulement, et j'en suis quasiment sûr, **vous avez dû oublier de dire à votre tableau de faire attention aux boutons ET aux combos !**
Rappelez-vous cette instruction : `this.tableau.setDefaultRenderer(JButton.class, new TableComponent());` .
Votre tableau ne fait attention qu'aux boutons ! !

D'accord. Comment fait-on, alors ?

Tout simplement en changeant `JButton.class` en `JComponent.class` .
Après avoir fait ces deux modifs, vous devriez avoir ceci :



Alors ! Elle n'est pas belle, la vie ?
Vous devez avoir saisi comment on fait pour utiliser les modèles de tableaux et les rendus de cellules, maintenant...

Cependant, vous aurez aussi constaté que vos composants sont inertes !
Ceci est dû au fait que vous allez devoir gérer la façon de réagir de vos cellules... Avant d'en arriver là, nous allons voir une autre façon d'afficher correctement des composants dans un **JTable**.
Nous allons pouvoir laisser de côté notre classe servant de modèle et nous concentrer sur les composants.

Nous allons commencer par revenir à un code plus sobre :

Code : Java

```
import java.awt.BorderLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;


public class Fenetre extends JFrame {

 private JTable tableau;
 private JButton change = new JButton("Changer la taille");
```

```
 public Fenetre(){
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setTitle("JTable");
  this.setSize(600, 180);

  Object[][] data = { {"Cysboy", "6boy", "Combo", new
Boolean(true)},
       {"BZHHydde", "BZH", "Combo", new Boolean(false)},
       {"IamBow", "BoW", "Combo", new Boolean(false)},
       {"FunMan", "Year", "Combo", new Boolean(true)}
  };

  String  title[] = {"Pseudo", "Age", "Taille", "OK ?"};

  this.tableau = new JTable(data, title);
  this.tableau.setRowHeight(30);
  this.getContentPane().add(new JScrollPane(tableau),
BorderLayout.CENTER);

 }
  public static void main(String[] args){
  Fenetre fen = new Fenetre();
  fen.setVisible(true);
 }
}
```

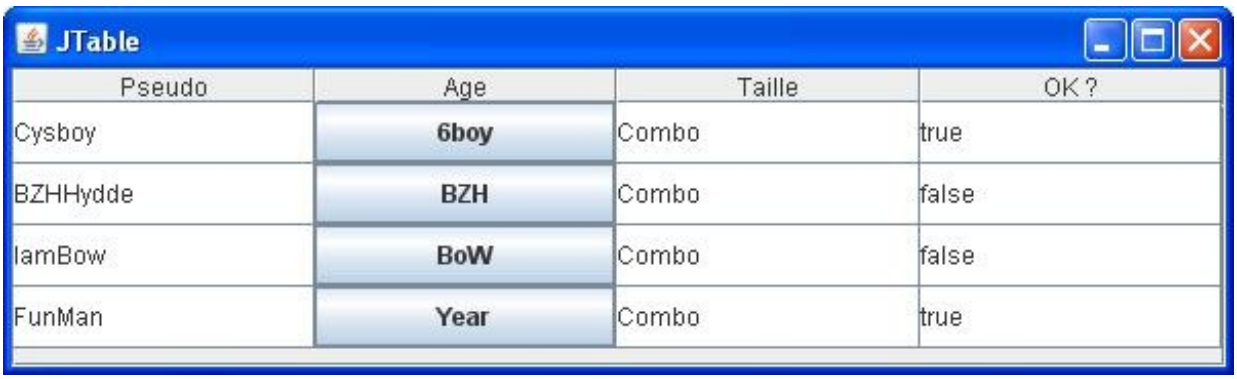
De là, nous allons créer une classe qui va permettre d'afficher un bouton dans les cellules de la seconde colonne et une combo dans les cellules de la troisième colonne...
Le principe est simple : nous allons créer une classe qui héritera de la classe **JButton** et qui implémentera l'interface **TableCellRenderer**.
Nous allons ensuite dire à notre tableau d'utiliser ce type de rendu pour la seconde colonne.

Voici notre classe **ButtonRenderer** :

Code : Java

```
import java.awt.Component;
import javax.swing.JButton;
import javax.swing.JTable;
import javax.swing.table.TableCellRenderer;

public class ButtonRenderer extends JButton implements
TableCellRenderer{

 public Component getTableCellRendererComponent( JTable table,
Object value,
             boolean isSelected, boolean isFocus,
             int row, int col) {
  //On écrit dans le bouton ce que contient la cellule
  setText((value != null) ? value.toString() : "");
  //on retourne notre bouton
  return this;
 }
}
```

Il nous suffit maintenant de mettre à jour le tableau grâce à cette méthode `this.tableau.getColumn("Age").setCellRenderer(new ButtonRenderer());` : on recupère la colonne grâce à son titre et on applique le rendu !
Résultat :



Votre bouton est de nouveau éditable, mais ce problème sera réglé par la suite...

Pour le rendu de la cellule numéro 3, je vous laisse un peu chercher, ce n'est pas très difficile maintenant que vous avez appris cette méthode.

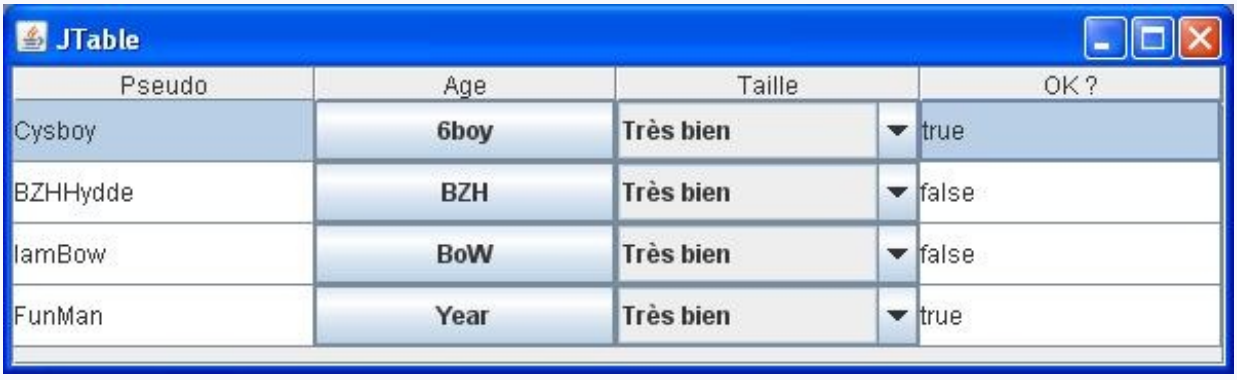
Voilà le code et le résultat :

Secret (cliquez pour afficher)

Code : Java

```
import java.awt.Component;

import javax.swing.JComboBox;
import javax.swing.JTable;
import javax.swing.table.TableCellRenderer;


public class ComboRenderer extends JComboBox implements
TableCellRenderer {

 public Component getTableCellRendererComponent(JTable table,
Object value,
   boolean isSelected, boolean isFocus, int row, int col) {
  this.addItem("Très bien");
  this.addItem("Bien");
  this.addItem("Mal");
  return this;
 }
}
```

Résultat :



La suite logique est donc la gestion des événements...
Retour en haut

Des tableaux très actifs !

Bon, la dernière ligne droite avant la fin du chapitre...

Comme je l'ai déjà dit, ici aussi, c'est simple : le tout est d'acquérir les connaissances requises !

Nous allons commencer par le plus difficile et nous terminerons par le plus simple !
Je vous le donne en mille : le composant le plus difficile à utiliser dans un tableau, entre un bouton et une combo c'est : **le bouton !**

Eh oui, vous verrez que la combo est presque gérée automatiquement, alors que vous allez devoir dire quoi faire à vos boutons...
Afin de réussir à faire ceci, nous allons donc créer une classe que notre tableau va utiliser afin de pouvoir utiliser les boutons pour faire des actions spécifiques.

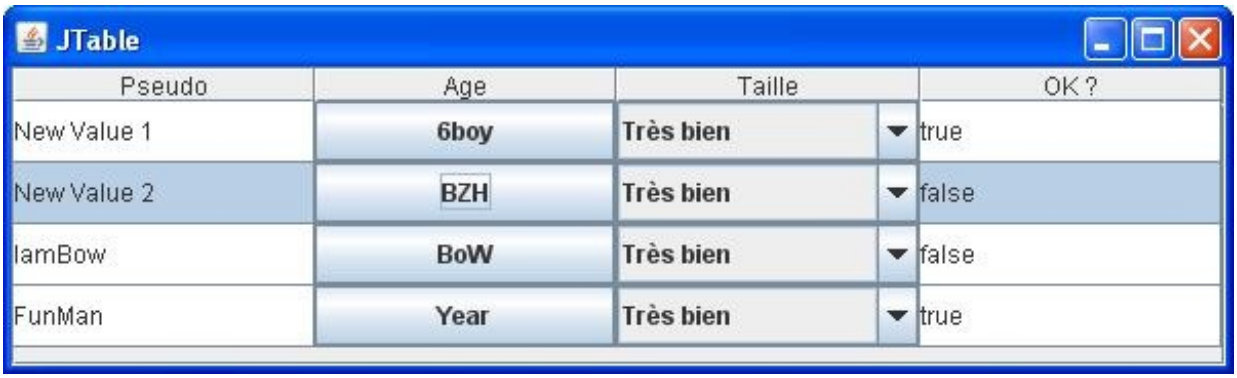
Code : Java

```
import java.awt.Component;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.DefaultCellEditor;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JTable;

public class ButtonEditor extends DefaultCellEditor {

 protected JButton button;
 private boolean   isPushed;
 private ButtonListener bListener = new ButtonListener();

 /**
* Constructeur avec une checkBox
* @param checkBox
* @param count
*/
 public ButtonEditor(JCheckBox checkBox) {
  //Par défaut, ce type d'objet travaille avec un JCheckBox
  super(checkBox);
     //On crée à nouveau notre bouton
  button = new JButton();
     button.setOpaque(true);
     //On lui attribue un listener
     button.addActionListener(bListener);
 }

 public Component getTableCellEditorComponent(JTable table, Object
value,
                   boolean isSelected, int row, int column) {
  //On définit le numéro de ligne à notre listener
  bListener.setRow(row);
  //Idem pour le numéro de colonne
  bListener.setColumn(column);
  //On passe aussi le tableau pour des actions potentielles
  bListener.setTable(table);

  //On réaffecte le libellé au bouton
  button.setText( (value ==null) ? "" : value.toString() );
  //On renvoie le bouton
     return button;
 }

 /**
* Notre listener pour le bouton
* @author CHerby
*
*/
```

```
class ButtonListener implements ActionListener{

    private int column, row;
    private JTable table;
    private int nbre = 0;

    public void setColumn(int col){this.column = col;}
    public void setRow(int row){this.row = row;}
    public void setTable(JTable table){this.table = table;}

    public void actionPerformed(ActionEvent event) {
   //On affiche un Zoli message mais vous pourriez faire les
traitements que vous voulez
   System.out.println("coucou du bouton : " +
((JButton)event.getSource()).getText() );
   //On affecte un nouveau libellé à une cellule de la ligne
   table.setValueAt("New Value " + (++nbre), this.row, (this.column
-1));
     }
   }

 }
```

Ce code n'est pas très difficile à comprendre... Vous commencez à avoir l'habitude de manipuler ce genre d'objet.
Maintenant, afin d'utiliser cet objet avec votre tableau, nous allons lui spécifier cet éditeur dans la colonne correspondante avec cette instruction : `this.tableau.getColumn("Age").setCellEditor(new ButtonEditor(new JCheckBox()));` .

Le rendu est très probant :



Vous pouvez voir que lorsque vous cliquez sur un bouton, la valeur dans la cellule juste à gauche est modifiée.
L'utilisation est donc très simple. Imaginez par conséquent que la gestion des combos est d'autant plus simple !

Un peu plus tôt, je vous ai fait développer une classe afin de pouvoir afficher la combo normalement. Cependant, il y a beaucoup plus simple... Vous avez pu voir que la classe **DefaultCellEditor** pouvait prendre un objet en paramètre : dans l'exemple utilisé dans le cas du **JButton**, il utilisait un **JCheckBox**.
Vous devez savoir que cet objet peut prendre d'autres types de paramètres :

- un **JComboBox** ;
- un **JTextField**.

Nous allons donc pouvoir utiliser l'objet **DefaultCellEditor** directement en lui passant une combo en paramètre...
Nous allons aussi enlever l'objet servant à afficher correctement la combo afin que vous puissiez juger de l'efficacité de cette méthode.

Voici le nouveau code du constructeur de notre fenêtre :

Code : Java

```
import java.awt.BorderLayout;

import javax.swing.DefaultCellEditor;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;
import javax.swing.table.DefaultTableCellRenderer;


public class Fenetre extends JFrame {

 private JTable tableau;
 private JButton change = new JButton("Changer la taille");
 //Contenu de notre combo
 private String[] comboData = {"Très bien", "Bien", "Mal"};

 public Fenetre(){
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setTitle("JTable");
  this.setSize(600, 180);
  //Données de notre tableau
  Object[][] data = { {"Cysboy", "6boy", comboData[0], new
Boolean(true)},
        {"BZHHydde", "BZH", comboData[0], new Boolean(false)},
        {"IamBow", "BoW", comboData[0], new Boolean(false)},
        {"FunMan", "Year", comboData[0], new Boolean(true)}
  };
  //titre du tableau
  String  title[] = {"Pseudo", "Age", "Taille", "OK ?"};
  //Combo à utiliser
  JComboBox combo = new JComboBox(comboData);

  this.tableau = new JTable(data, title);
  this.tableau.setRowHeight(30);
  this.tableau.getColumn("Age").setCellRenderer(new
ButtonRenderer());
  this.tableau.getColumn("Age").setCellEditor(new ButtonEditor(new
JCheckBox()));
  //On définit l'éditeur par défaut pour la cellule
  //en lui spécifiant quel type d'affichage prendre en compte
  this.tableau.getColumn("Taille").setCellEditor(new
DefaultCellEditor(combo));
  this.getContentPane().add(new JScrollPane(tableau),
BorderLayout.CENTER);
 }

  public static void main(String[] args){
  Fenetre fen = new Fenetre();
  fen.setVisible(true);
 }
}
```

Vous pouvez voir que c'est d'une simplicité enfantine !
Le résultat est, en plus, très convaincant :

Votre cellule se "***transforme***" en combo lorsque vous cliquez dessus !
En fait, lorsque le tableau sent une action sur cette cellule, celui-ci utilise l'éditeur que vous avez spécifié pour celle-ci et, comme nous en avons spécifié un afin d'afficher une combo, la dite combo est affichée !

Si vous préférez que la combo soit affichée directement même sans clic de souris, il vous suffit de laisser l'objet gérant l'affichage et le tour est joué !
De même, pour le bouton, si vous enlevez l'objet de rendu du tableau, celui-ci s'affiche comme un bouton lors du clic sur la cellule !

Il ne nous reste plus qu'à voir comment rajouter des informations dans notre tableau et l'affaire est entendue...

Certains d'entre-vous l'auront remarqué, les boutons ont un drôle de comportement...
Ceci est dû au fait que vous avez défini des comportements spéciaux à votre tableau... Il vous faut donc définir un modèle à utiliser afin de bien définir tous les points comme l'affichage, la mise à jour...

Nous allons donc utiliser un modèle de tableau perso dont les actions seront définis par nous !
Voici la classe **Fenetre** modifiée en conséquence :

Code : Java

```
package com.sdz.jtable2;
import java.awt.BorderLayout;

import javax.swing.DefaultCellEditor;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;
import javax.swing.table.DefaultTableCellRenderer;



public class Fenetre extends JFrame {

 private JTable tableau;
 private JButton change = new JButton("Changer la taille");
 //Contenu de notre combo
 private String[] comboData = {"Très bien", "Bien", "Mal"};

 public Fenetre(){
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setTitle("JTable");
  this.setSize(600, 180);
  //Données de notre tableau
```

```
        Object[][] data = { {"Cysboy", "6boy", comboData[0], new
Boolean(true)},
              {"BZHHydde", "BZH", comboData[0], new Boolean(false)},
              {"IamBow", "BoW", comboData[0], new Boolean(false)},
              {"FunMan", "Year", comboData[0], new Boolean(true)}
        };
        //titre du tableau
        String  title[] = {"Pseudo", "Age", "Taille", "OK ?"};
        //Combo à utiliser
        JComboBox combo = new JComboBox(comboData);

        //Nous devons utiliser un modèle d'affichage spécifique afin de
pallier aux bugs d'affichage !
        ZModel zModel = new ZModel(data, title);

        this.tableau = new JTable(zModel);
        this.tableau.setRowHeight(30);
        this.tableau.getColumn("Age").setCellRenderer(new
ButtonRenderer());
        this.tableau.getColumn("Age").setCellEditor(new ButtonEditor(new
JCheckBox()));
        //On définit l'éditeur par défaut pour la cellule
        //en lui spécifiant quel type d'affichage prendre en compte
        this.tableau.getColumn("Taille").setCellEditor(new
DefaultCellEditor(combo));
        this.getContentPane().add(new JScrollPane(tableau),
BorderLayout.CENTER);
       }


       class ZModel extends AbstractTableModel{

        private Object[][] data;
        private String[] title;
        /**
* Constructeur
* @param data
* @param title
*/
        public ZModel(Object[][] data, String[] title){
         this.data = data;
         this.title = title;
        }
        /**
* Retourne le titre de la colonne à l'indice spécifé
*/
        public String getColumnName(int col) {
          return this.title[col];
        }

        /**
* Retourne le nombre de colonnes
*/
        public int getColumnCount() {
         return this.title.length;
        }

        /**
* Retourne le nombre de lignes
*/
        public int getRowCount() {
         return this.data.length;
        }

        /**
* Retourne la valeur à l'emplacement spécifié
*/
        public Object getValueAt(int row, int col) {
         return this.data[row][col];
        }
```

```
    /**
   * Défini la valeur à l'emplacement spécifié
   */
    public void setValueAt(Object value, int row, int col) {
     //On interdit la modification sur certaine colonne !
     if(!this.getColumnName(col).equals("Age") &&
  !this.getColumnName(col).equals("Suppression"))
       this.data[row][col] = value;
    }

    /**
   * Retourne la classe de la donnée de la colonne
   * @param col
   */
    public Class getColumnClass(int col){
     //On retourne le type de la cellule à la colonne demandée
     //On se moque de la ligne puisque les données sur chaque ligne
  sont les mêmes
     //On choisit donc la première ligne
     return this.data[0][col].getClass();
    }

    public boolean isCellEditable(int row, int col){
     return true;
    }
   }




    public static void main(String[] args){
    Fenetre fen = new Fenetre();
    fen.setVisible(true);
   }
  }
```

Vous aurez remarqué que nous construisons notre tableau par le biais de notre modèle, ceci implique que nous allons aussi passer par note modèle afin de modifier notre tableau !
La conséquence direct de ceci : nous allons devoir modifier un peu notre classe **ButtonEditor**.

Voici la classe ButtonEditor utilisant le modèle de tableau pour gérer la modification des valeurs :

Code : Java

```
package com.sdz.jtable2;

import java.awt.Component;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.DefaultCellEditor;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;

public class ButtonEditor extends DefaultCellEditor {

 protected JButton button;
 private boolean   isPushed;
 private ButtonListener bListener = new ButtonListener();

 /**
* Constructeur avec une checkBox
```

```
     * @param checkBox
     * @param count
     */
      public ButtonEditor(JCheckBox checkBox) {
       //Par défaut, ce type d'objet travaille avec un JCheckBox
       super(checkBox);
          //On crée à nouveau notre bouton
       button = new JButton();
          button.setOpaque(true);
          //On lui attribue un listener
          button.addActionListener(bListener);
      }

      public Component getTableCellEditorComponent(JTable table, Object
     value,
                       boolean isSelected, int row, int column) {
       //On définit le numéro de ligne à notre listener
       bListener.setRow(row);
       //Idem pour le numéro de colonne
       bListener.setColumn(column);
       //On passe aussi le tableau pour des actions potentielles
       bListener.setTable(table);

       //On réaffecte le libellé au bouton
       button.setText( (value ==null) ? "" : value.toString() );
       //On renvoie le bouton
          return button;
      }

      /**
     * Notre listener pour le bouton
     * @author CHerby
     *
     */
      class ButtonListener implements ActionListener{

         private int column, row;
         private JTable table;
         private int nbre = 0;
         private JButton button;

         public void setColumn(int col){this.column = col;}
         public void setRow(int row){this.row = row;}
         public void setTable(JTable table){this.table = table;}

         public JButton getButton(){return this.button;}

         public void actionPerformed(ActionEvent event) {
        //On affiche un Zoli message mais vous pourriez faire les
     traitements que vous voulez
        System.out.println("coucou du bouton : " +
     ((JButton)event.getSource()).getText() );
        //On affecte un nouveau libellé à une celulle de la ligne
        ((AbstractTableModel)table.getModel()).setValueAt("New Value " +
     (++nbre), this.row, (this.column -1));
        //Permet de dire à notre tableau qu'une valeur a changé
        //à l'emplacement déterminé par les valeur passée en paramètre

     ((AbstractTableModel)table.getModel()).fireTableCellUpdated(this.row,
     this.column - 1);
        this.button = ((JButton)event.getSource());
         }
        }

      }
```

Voilà, après ceci, le bug d'affichage devrait avoir disparu !

Je vous propose donc de continuer dans notre lancée.
Retour en haut

Ajouter des lignes et des colonnes

Je vais profiter de ce point pour vous montrer une autre façon d'initialiser un tableau :

Code : Java

```
//data et title sont toujours nos tableaux d'objets !
JTable tableau = new JTable(new DefaultTableModel(data, title));
```

Et l'intérêt de faire comme ça ?

C'est très simple : l'ajout et la suppression dynamiques d'entrées dans un tableau se fait via un modèle de rendu, mais vous n'avez pas l'obligation de recréer le vôtre, celui-ci fera très bien l'affaire !

Dans un premier temps, nous allons ajouter et retirer des lignes à notre tableau.
Nous allons garder le même code que précédemment avec deux - trois ajouts :

- le bouton pour ajouter un ligne ;
- le bouton pour effacer une ligne.

Grâce au modèle par défaut défini lors de la création du tableau, nous allons pouvoir avoir accès à deux méthodes fort utiles :

- `addRow(Object[] lineData)` : permet d'ajouter une ligne au modèle et met automatiquement à jour le tableau ;
- `removeRow(int row)` : efface une ligne du modèle et met automatiquement à jour le tableau.

Afin de pouvoir utiliser le dit modèle, nous allons devoir récupérer celui-ci.
En fait, c'est notre tableau qui va nous le fournir en invoquant la méthode `getModel()` qui retourne un objet **`TableModel`**.

Un cast sera nécessaire afin de pouvoir utiliser l'objet retourné !

Au final, voici que nous obtiendrons :

au lancement du programme

après ajout d'une ligne

après suppression d'une ligne

Vous constatez que j'ai ajouté un bouton "**Ajouter une ligne**" ainsi qu'un bouton "**Supprimer la ligne**" ; mis à part ça, l'IHM n'a pas changé.

Voilà les codes source utilisés dans cet exemple. Ceux-ci sont assez documentés et, mis à part l'utilisation de `addRow(Object[] lineData)` et de `removeRow(int row)` , rien n'a changé.

Souvenez-vous bien que nous passons par un modèle de tableau perso !
Nous allons devoir gérer les méthodes d'ajout et de suppression de ligne dans ce modèle.

De ce fait, c'est nous qui allons devoir déterminer et coder les actions d'ajout et de suppression de lignes dans notre tableau !

Fenetre.java

Code : Java

```
package com.sdz.jtable;

import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.DefaultCellEditor;
import javax.swing.JButton;
```

```
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;
import javax.swing.table.DefaultTableCellRenderer;
import javax.swing.table.DefaultTableModel;


public class Fenetre extends JFrame {

 private JTable tableau;
 private JButton change = new JButton("Changer la taille");
 private String[] comboData = {"Très bien", "Bien", "Mal"};
 private String supp = "Supprimer la ligne";
 private JComboBox combo;

 public Fenetre(){
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setTitle("JTable");
  this.setSize(600, 250);
  this.createContent();
 }

 private void createContent(){
  //Données de notre tableau
  Object[][] data = { {"Cysboy", "6boy", comboData[0], new
Boolean(true), supp},
        {"BZHHydde", "BZH", comboData[0], new Boolean(false), supp},
        {"IamBow", "BoW", comboData[0], new Boolean(false), supp},
        {"FunMan", "Year", comboData[0], new Boolean(true), supp}
  };

  //titre du tableau
  String  title[] = {"Pseudo", "Age", "Taille", "OK ?",
"Suppression"};
  //Combo à utiliser
  combo = new JComboBox(comboData);

  //Nous devons utiliser un modèle d'affichage spécifique afin de
pallier aux bugs d'affichage !
  ZModel zModel = new ZModel(data, title);

  this.tableau = new JTable(zModel);
  this.tableau.setRowHeight(30);
  this.tableau.getColumn("Age").setCellRenderer(new
ButtonRenderer());
  this.tableau.getColumn("Age").setCellEditor(new ButtonEditor(new
JCheckBox()));

  //On définit l'éditeur par défaut pour la cellule
  //en lui spécifiant quel type d'affichage prendre en compte
  this.tableau.getColumn("Taille").setCellEditor(new
DefaultCellEditor(combo));
  DefaultTableCellRenderer dcr = new DefaultTableCellRenderer();
  this.tableau.getColumn("Taille").setCellRenderer(dcr);

  //On définit un éditeur pour la colonne "supprimer"
  this.tableau.getColumn("Suppression").setCellEditor(new
DeleteButtonEditor(new JCheckBox()));

  //On ajoute le tableau
  this.getContentPane().add(new JScrollPane(tableau),
BorderLayout.CENTER);

  JButton ajouter = new JButton("Ajouter une ligne");
  ajouter.addActionListener(new MoreListener());
```

```
        this.getContentPane().add(ajouter, BorderLayout.SOUTH);
    }

    class ZModel extends AbstractTableModel{

     private Object[][] data;
     private String[] title;
     /**
* Constructeur
* @param data
* @param title
*/
     public ZModel(Object[][] data, String[] title){
      this.data = data;
      this.title = title;
     }
     /**
* Retourne le titre de la colonne à l'indice spécifé
*/
     public String getColumnName(int col) {
       return this.title[col];
     }

     /**
* Retourne le nombre de colonnes
*/
     public int getColumnCount() {
      return this.title.length;
     }

     /**
* Retourne le nombre de lignes
*/
     public int getRowCount() {
      return this.data.length;
     }

     /**
* Retourne la valeur à l'emplacement spécifié
*/
     public Object getValueAt(int row, int col) {
      return this.data[row][col];
     }

     /**
* Défini la valeur à l'emplacement spécifié
*/
     public void setValueAt(Object value, int row, int col) {
      //On interdit la modification sur certaine colonne !
      if(!this.getColumnName(col).equals("Age") &&
!this.getColumnName(col).equals("Suppression"))
       this.data[row][col] = value;
     }

     /**
* Retourne la classe de la donnée de la colonne
* @param col
*/
     public Class getColumnClass(int col){
      //On retourne le type de la cellule à la colonne demandée
      //On se moque de la ligne puisque les données sur chaque ligne
sont les mêmes
      //On choisit donc la première ligne
      return this.data[0][col].getClass();
     }

     /**
* Méthode permettant de retirer une ligne du tableau
* @param position
*/
```

```
  public void removeRow(int position){

   int indice = 0, indice2 = 0, nbRow = this.getRowCount()-1, nbCol
= this.getColumnCount();
   Object temp[][] = new Object[nbRow][nbCol];

   for(Object[] value : this.data){
    if(indice != position){
     temp[indice2++] = value;
    }
    System.out.println("Indice = " + indice);
    indice++;
   }
   this.data = temp;
   temp = null;
   //Cette méthode permet d'avertir le tableau que les données ont
été modifiées
   //Ce qui permet une mise à jours complète du tableau
   this.fireTableDataChanged();
  }

  /**
* Permet d'ajouter une ligne dans le tableau
* @param data
*/
  public void addRow(Object[] data){
   int indice = 0, nbRow = this.getRowCount(), nbCol =
this.getColumnCount();

   Object temp[][] = this.data;
   this.data = new Object[nbRow+1][nbCol];

   for(Object[] value : temp)
    this.data[indice++] = value;


   this.data[indice] = data;
   temp = null;
   //Cette méthode permet d'avertir le tableau que les données ont
été modifiées
   //Ce qui permet une mise à jours complète du tableau
   this.fireTableDataChanged();
  }


  public boolean isCellEditable(int row, int col){
   return true;
  }
 }


 class MoreListener implements ActionListener{
  public void actionPerformed(ActionEvent event) {
   Object[] donnee = new Object[]{"Angelo", "Rennais", comboData[0],
new Boolean(false), supp};
   ((ZModel)tableau.getModel()).addRow(donnee);
  }
 }

 public static void main(String[] args){
  Fenetre fen = new Fenetre();
  fen.setVisible(true);
 }
}
```

ButtonRenderer.java

Code : Java

```
import java.awt.Component;

import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JTable;
import javax.swing.table.TableCellRenderer;

public class ButtonRenderer extends JButton implements
TableCellRenderer{

 public Component getTableCellRendererComponent( JTable table,
Object value,
             boolean isSelected, boolean isFocus,
             int row, int col) {
  //On écrit dans le bouton avec la valeur de la cellule
  setText((value != null) ? value.toString() : "");
  //on retourne notre bouton
  return this;
 }
}
```

ButtonEditor.java

Code : Java

```
import java.awt.Component;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.DefaultCellEditor;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JTable;
import javax.swing.table.DefaultTableModel;

public class ButtonEditor extends DefaultCellEditor {

 protected JButton button;
 private ButtonListener bListener = new ButtonListener();

 /**
* Constructeur avec une checkBox
* @param checkBox
* @param count
*/
 public ButtonEditor(JCheckBox checkBox) {
  //Par défaut, ce type d'objet travaille avec un JCheckBox
  super(checkBox);
     //On crée à nouveau notre bouton
  button = new JButton();
     button.setOpaque(true);
     //On lui attribue un listener
     button.addActionListener(bListener);
 }

 public Component getTableCellEditorComponent(JTable table, Object
value,
                  boolean isSelected, int row, int column) {
  //On définit le numéro de lignes à notre listener
```

```
        bListener.setRow(row);
        //Idem pour le numéro de colonnes
        bListener.setColumn(column);
        //On passe aussi le tableau pour des actions potentielles
        bListener.setTable(table);
        //On réaffecte le libellé au bouton
        button.setText( (value ==null) ? "" : value.toString() );
        //On renvoie le bouton
           return button;
      }

      /**
     * Notre listener pour le bouton
     * @author CHerby
     *
     */
      class ButtonListener implements ActionListener{

         private int column, row;
         private JTable table;
         private int nbre = 0;
         private JButton button;

         public void setColumn(int col){this.column = col;}
         public void setRow(int row){this.row = row;}
         public void setTable(JTable table){this.table = table;}
         public JButton getButton(){return this.button;}

         public void actionPerformed(ActionEvent event) {
        //On affiche un Zoli message mais vous pourriez faire les
     traitements que vous voulez
        System.out.println("coucou du bouton : " +
     ((JButton)event.getSource()).getText() );
        //On affecte un nouveau libellé à une celulle de la ligne
        ((AbstractTableModel)table.getModel()).setValueAt("New Value " +
     (++nbre), this.row, (this.column -1));
        //Permet de dire à notre tableau qu'une valeur a changé
        //à l'emplacement déterminé par les valeur passée en paramètre

     ((AbstractTableModel)table.getModel()).fireTableCellUpdated(this.row,
     this.column - 1);
        this.button = ((JButton)event.getSource());
         }
      }
     }
```

DeleteButtonEditor.java

Code : Java

```
     import java.awt.Component;
     import java.awt.event.ActionEvent;
     import java.awt.event.ActionListener;

     import javax.swing.DefaultCellEditor;
     import javax.swing.JButton;
     import javax.swing.JCheckBox;
     import javax.swing.JTable;
     import javax.swing.table.DefaultTableModel;

     public class DeleteButtonEditor extends DefaultCellEditor {

      protected JButton button;
      private DeleteButtonListener bListener = new
```

```
DeleteButtonListener();

 /**
* Constructeur avec une checkBox
* @param checkBox
* @param count
*/
 public DeleteButtonEditor(JCheckBox checkBox) {
  //Par défaut, ce type d'objet travaille avec un JCheckBox
  super(checkBox);
     //On crée à nouveau notre bouton
  button = new JButton();
     button.setOpaque(true);
     //On lui attribue un listener
     button.addActionListener(bListener);
 }

 public Component getTableCellEditorComponent(JTable table, Object
value,
                    boolean isSelected, int row, int column) {
  //On définit le numéro de lignes à notre listener
  bListener.setRow(row);
  //On passe aussi le tableau pour des actions potentielles
  bListener.setTable(table);
  //On réaffecte le libellé au bouton
  button.setText( (value ==null) ? "" : value.toString() );
  //On renvoie le bouton
     return button;
 }

 /**
* Notre listener pour le bouton
* @author CHerby
*
*/
 class DeleteButtonListener implements ActionListener{

    private int row;
    private JTable table;

    public void setRow(int row){this.row = row;}
    public void setTable(JTable table){this.table = table;}

    public void actionPerformed(ActionEvent event) {
   if(table.getRowCount() > 0){
    //On affiche un Zoli message mais vous pourriez faire les
traitements que vous voulez
    System.out.println("coucou du bouton : " +
((JButton)event.getSource()).getText() );
    //On affecte un nouveau libellé à une celulle de la ligne
    ((ZModel)table.getModel()).removeRow(this.row);

   }
    }
   }
 }
```

ComboRenderer.java

Code : Java

```
import java.awt.Component;
import javax.swing.JComboBox;
import javax.swing.JTable;
```

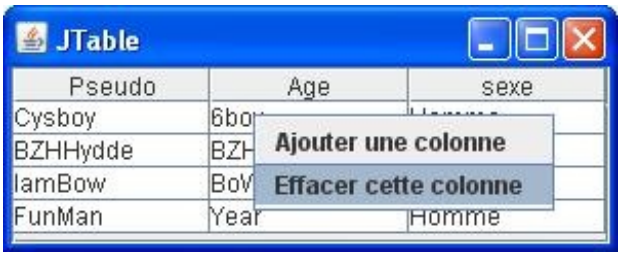
```
import javax.swing.table.TableCellRenderer;


public class ComboRenderer extends JComboBox implements
TableCellRenderer {

 public Component getTableCellRendererComponent(JTable table, Object
value,
   boolean isSelected, boolean isFocus, int row, int col) {

  this.addItem("Très bien");
  this.addItem("Bien");
  this.addItem("Mal");
  return this;
 }
}
```

C'est assez simple finalement, mais c'est tout de même assez galère lorsqu'on ne sait pas par où commencer !

Maintenant que vous savez faire tout ceci, ajouter ou retirer des colonnes ne devrait pas vous faire sourciller...
Nous allons toujours utiliser notre modèle de rendu afin d'ajouter ou retirer des colonnes.
Cette fois, j'ai utilisé des menus contextuels afin de faire ceci : faut bien faire varier les plaisirs ! C'est surtout que, comme ça, vous verrez le rapprochement entre les arbres et les tableaux.

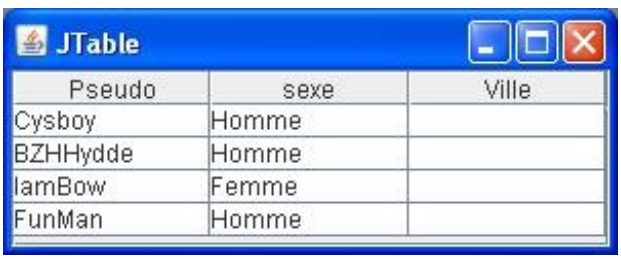
Voilà ce que j'ai obtenu :

avant effacement d'une colonne



après effacement d'une colonne

après ajout d'une colonne nommée "Ville"



Pour avoir ce rendu, j'ai refait une classe épurée... On commençait à ne plus s'y retrouver...

Voici le code de cette classe :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Point;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

import javax.swing.JFrame;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPopupMenu;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableColumn;


public class TableFrame extends JFrame {
 private JTable tableau;
 private String supp = "Supprimer la ligne";

 public TableFrame(){
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setTitle("JTable");
  this.setSize(300, 120);
  this.createContent();
 }

 private void createContent(){
  Object[][] data = { {"Cysboy", "6boy", "Homme"},
        {"BZHHydde", "BZH", "Homme"},
        {"IamBow", "BoW", "Femme"},
        {"FunMan", "Year", "Homme"}
  };

  String  title[] = {"Pseudo", "Age", "sexe"};
  this.tableau = new JTable(new DefaultTableModel(data, title));
  this.tableau.getTableHeader().addMouseListener(new MouseAdapter(){
   public void mouseReleased(MouseEvent event){
    if(event.getButton() == event.BUTTON3){
     //Si on est dans le tableau
     if(tableau.columnAtPoint(new Point(event.getX(), event.getY())) != -1){

      //ajouter une colonne
      JMenuItem ajouter = new JMenuItem("Ajouter une colonne");
      ajouter.addActionListener(new AddListener());

      //Le menu effacer
      JMenuItem erase = new JMenuItem("Effacer cette colonne");
      erase.addActionListener(new EraseListener(new Point(event.getX(), event.g

      //Ajout du menu contextuel
      JPopupMenu menu = new JPopupMenu();
      menu.add(ajouter);
      menu.add(erase);
      menu.show(tableau, event.getX(), event.getY());
     }
    }
   }
  });
  this.getContentPane().add(new JScrollPane(tableau), BorderLayout.CENTER);
 }

 /**
```

```
        * Classe gérant la suppression d'une colonne
        * @author CHerby
        */
         class EraseListener implements ActionListener{
          private Point point = null;
          private Object[][] data;
          private Object[] title;
          public EraseListener(Point col){
           this.point = col;
          }
          public void actionPerformed(ActionEvent event) {
           //On récupère l'indice de la colonne
           int col = tableau.columnAtPoint(this.point);
           //On en déduit la colonne sur laquelle on se trouve
           TableColumn column = tableau.getColumn(tableau.getColumnName(col));

           //On avertit le modèle que le tableau a perdu une colonne
           initNewData(column);
           //On donne les nouvelles données au modèle
           ((DefaultTableModel)tableau.getModel()).setDataVector(this.data, this.title)

          }
          /**
        * Méthode qui génère un nouveau contenu au modèle
        * @param column
        * @return
        */
          private void initNewData(TableColumn column){
           this.data = new Object[tableau.getRowCount()][tableau.getColumnCount()-1];
           this.title = new Object[tableau.getColumnCount()-1];

           //On parcourt toutes les lignes
           for(int i = 0; i < tableau.getRowCount(); i++){

            //Toutes les colonnes
            for(int j = 0, k = 0; j < tableau.getColumnCount(); j++){
             //Si la colonne concernée n'est pas celle à effacer

        if(!((DefaultTableModel)tableau.getModel()).getColumnName(j).equals(column.getH

              //On récupère les titres de colonnes au premier passage
              if(i == 0)this.title[k] = ((DefaultTableModel)tableau.getModel()).getColu
              //on récupère les données
              this.data[i][k] = tableau.getValueAt(i, j);
              k++;
             }
            }
           }
          }
         }

         /**
        * Permet d'ajouter une colonne au tableau, enfin, au modèle du tableau
        * @author CHerby
        */
         class AddListener implements ActionListener{
          public void actionPerformed(ActionEvent event) {
           //On affiche une pop-up
           JOptionPane jop = new JOptionPane(), jop2 = new JOptionPane();
           String nom = jop.showInputDialog(null, "Saisissez le nom de la nouvelle colo
        d'une colonne", JOptionPane.QUESTION_MESSAGE);
           //Si les contrôles d'usage sont bons
           if(nom != null){
            if(!nom.trim().equals("")){
             //On ajoute une colonne au modèle
             ((DefaultTableModel)tableau.getModel()).addColumn(nom);
            }
           }
          }
         }
```

```
    public static void main(String[] args){
     TableFrame fen = new TableFrame();
     fen.setVisible(true);
    }
}
```

Voilà un chapitre rondement mené.
Je crois qu'il est temps de faire un topo.

Retour en haut

Ce qu'il faut retenir

- Un tableau est en fait un composant appelé `JTable`.
- Celui-ci prend en paramètre un tableau d'objets à deux dimensions (les données) et un tableau de chaînes de caractères (le titre des colonnes).
- Vous pouvez utiliser, afin de gérer vous-mêmes le contenu du tableau, un modèle de données (`TableModel`).
- Afin de pouvoir ajouter ou retirer des lignes dans un tableau, il faut passer par un modèle de données. Ainsi, l'affichage est mis à jour automatiquement.
- Il en va de même pour l'ajout et la suppression de colonnes.
- La gestion de l'affichage brut (hors édition) des cellules peut se gérer colonne par colonne en utilisant une classe dérivant de `TableCellRenderer`.
- La gestion de l'affichage brut lors de l'édition d'une cellule peut se gérer colonne par colonne en utilisant une classe dérivant de `DefaultCellEditor`.
- Pour un tableau contenant plusieurs types de données, il peut être préférable de gérer entièrement la façon dont le tableau traite ses cellules.

Maintenant, le petit QCM des familles.

Retour en haut

Ouf !
Une bonne chose de faite...

Je vous avais dit que ce chapitre allait être assez complexe, comme le précédent !
Vous vous en êtes sortis indemnes, ou presque...

Retour en haut

Ce que vous pouvez voir en plus

Il faut être honnête, vous avez vu une bonne partie des objets graphiques proposés par Java. Il en reste bien d'autres, mais le fonctionnement de ceux-ci ne devraient pas vous poser trop de problèmes maintenant que vous êtes initiés à la programmation événementielle...

Je vous propose de faire un tour d'horizon de certains autres objets dont vous pourriez avoir besoin.
Ceux-ci seront accompagnés d'un code facile à comprendre ainsi que d'une brève description !

En avant pour **le dernier chapitre de ce tuto Java** !

D'autres conteneurs graphiques

Voici quelques objets que vous pourrez trouver sur votre passage.

Le `JWindow`

Pour faire simple, c'est une `JFrame` mais sans les contours de fenêtre permettant de **réduire, fermer ou agrandir** ! Souvent utiliser pour faire des *splash screen* : la même chose que vous avez au lancement d'Eclipse...

Code source :

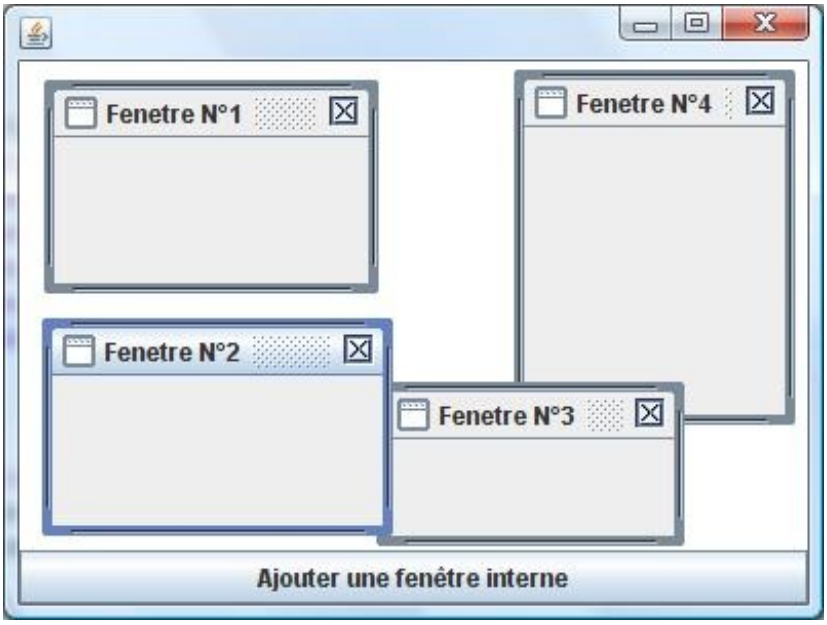
Secret (cliquez pour afficher)

Code : Java

```
import java.awt.Color;

import javax.swing.BorderFactory;
import javax.swing.ImageIcon;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JWindow;
import javax.swing.border.BevelBorder;


public class Window extends JWindow{

 public static void main(String[] args){
  Window wind = new Window();
  wind.setVisible(true);
 }

 public Window(){

  setSize(220, 165);
  setLocationRelativeTo(null);

  JPanel pan = new JPanel();
  JLabel img = new JLabel(new ImageIcon("planète.jpeg"));
  img.setVerticalAlignment(JLabel.CENTER);
  img.setHorizontalAlignment(JLabel.CENTER);

  pan.setBorder(BorderFactory.createLineBorder(Color.blue));
  pan.add(img);
  getContentPane().add(pan);
 }
}
```

Le `JDesktopPane` combiné à des `JInternalFrame`

Ces deux objets sont très souvent associés et permettent de faire des applications multi-fenêtres !



Code source :

Secret (cliquez pour afficher)

Code : Java

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JDesktopPane;
import javax.swing.JFrame;
import javax.swing.JInternalFrame;
import javax.swing.JInternalFrame;


public class Bureau extends JFrame{

 private static int nbreFenetre = 0;
 private JDesktopPane desktop = new JDesktopPane();
 private static int xy = 10;

 public Bureau(){
  this.setSize(400, 300);
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  JButton ajouter = new JButton("Ajouter une fenêtre interne");
  ajouter.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent event){
    ++nbreFenetre;
    xy += 2;
    desktop.add(new MiniFenetre(nbreFenetre), nbreFenetre);
   }
  });

  this.getContentPane().add(desktop, BorderLayout.CENTER);
  this.getContentPane().add(ajouter, BorderLayout.SOUTH);

 }

 class MiniFenetre extends JInternalFrame{
```

```
        public MiniFenetre(int nbre){
         this.setTitle("Fenetre N°"+nbre);
         this.setClosable(true);
         this.setResizable(true);
         this.setSize(150, 80);
         this.setLocation(xy, xy);
         this.setVisible(true);
        }
       }

       public static void main(String[] args){
        Bureau bureau = new Bureau();
        bureau.setVisible(true);
       }

      }
```

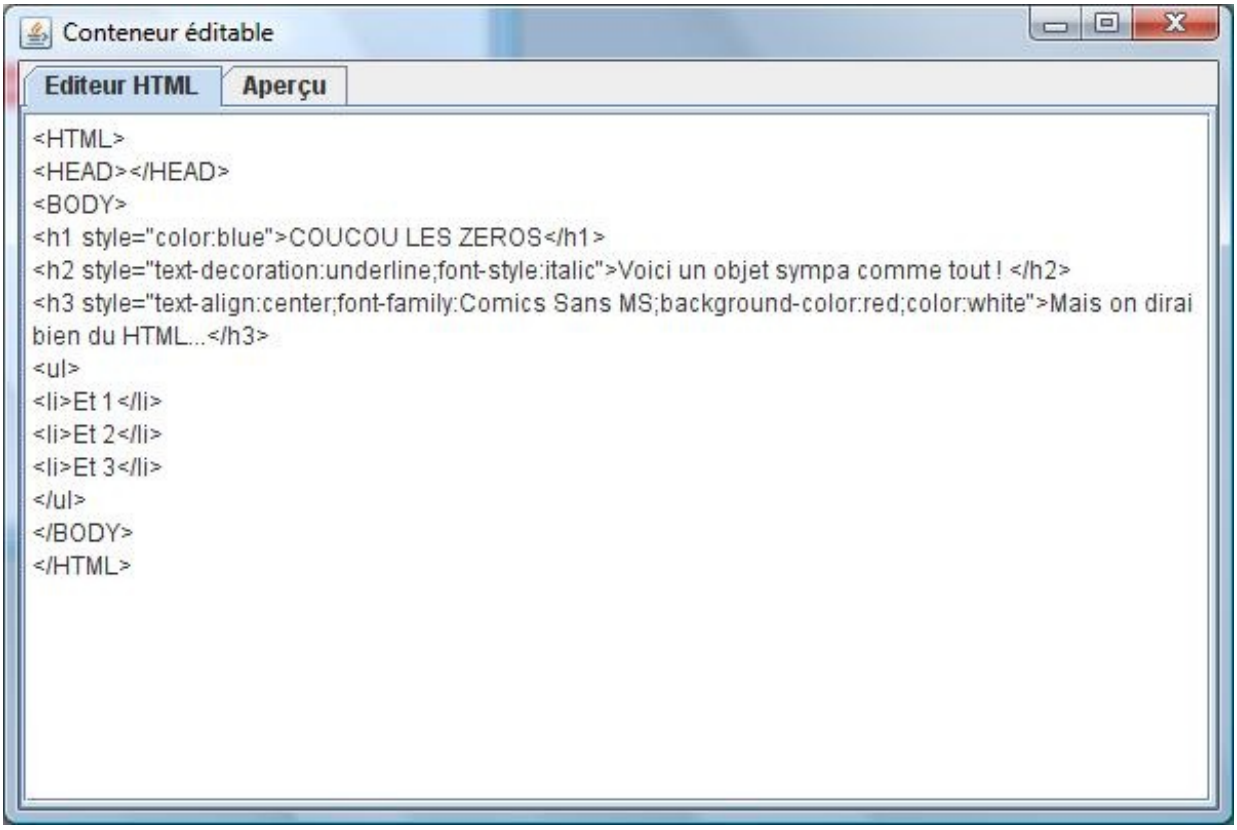
Retour en haut

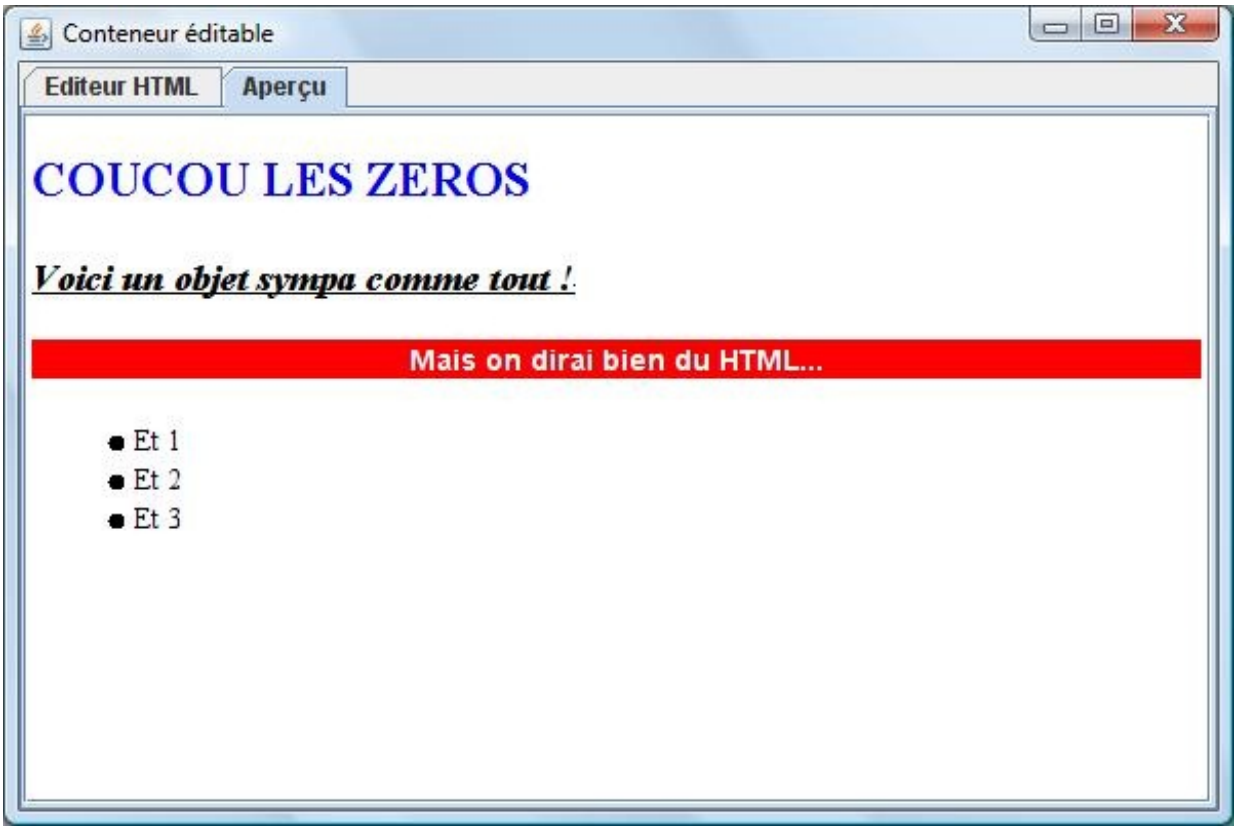
D'autres objets graphiques

Voici un objet sympa mais quelque peu limité par la façon dont il gère son contenu HTML !

Le `JEditorPane`

Il permet de réaliser des textes riches avec mise en page.
Il y a aussi le **`JTextPane`** qui vous permet de faire un mini-éditeur de texte très facilement (enfin, tout est relatif...).





Code source de cette fenêtre :

Secret (cliquez pour afficher)

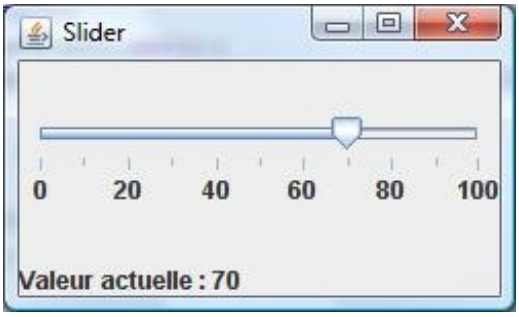
Code : Java

```
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

import javax.swing.JEditorPane;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTabbedPane;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
import javax.swing.text.html.HTMLEditorKit;


public class Fenetre extends JFrame {

 private JEditorPane editorPane, apercu;
 private JTabbedPane onglet = new JTabbedPane();

 public Fenetre(){

  this.setSize(600, 400);
  this.setTitle("Conteneur éditable");
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  editorPane = new JEditorPane();

editorPane.setText("<HTML>\n<HEAD></HEAD>\n<BODY>\n\n</BODY>\n</HTML>");

  apercu = new JEditorPane();
  apercu.setEditable(false);

  onglet.addTab("Editeur HTML", new JScrollPane(editorPane));
  onglet.addTab("Aperçu", new JScrollPane(apercu));
  onglet.addChangeListener(new ChangeListener(){

   public void stateChanged(ChangeEvent e) {

    FileWriter fw = null;

    try {
     fw = new FileWriter(new File("tmp/tmp.html"));
     fw.write(editorPane.getText());
     fw.close();
    } catch (FileNotFoundException e1) {
     // TODO Auto-generated catch block
     e1.printStackTrace();
    } catch (IOException e1) {
     // TODO Auto-generated catch block
     e1.printStackTrace();
    }
    //*
    try {
     File file = new File("tmp/tmp.html");
     apercu.setEditorKit(new HTMLEditorKit());
     apercu.setPage(file.toURL());

    } catch (IOException e1) {
     // TODO Auto-generated catch block
```

```
            e1.printStackTrace();
           }

          }
         });

         this.getContentPane().add(onglet, BorderLayout.CENTER);
         this.setVisible(true);
        }

       public static void main(String[] args){
        Fenetre fen = new Fenetre();
       }

      }
```

Le JSlider

Celui-ci est un outil qui vous permet d'utiliser un système de mesure pour une application : re-dimensionner une image, choisir le tempo d'un morceau de musique, l'opacité d'une couleur...



Code source :

Secret (cliquez pour afficher)

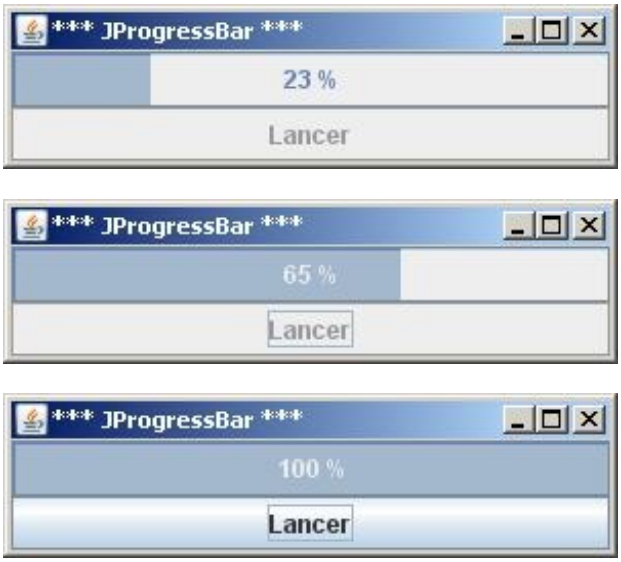
Code : Java

```
import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JSlider;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;


public class Slide extends JFrame{

 private JLabel label = new JLabel("Valeur actuelle : 30");

 public Slide(){
  this.setSize(250, 150);
  this.setTitle("Slider");
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  JSlider slide = new JSlider();

  slide.setMaximum(100);
  slide.setMinimum(0);
  slide.setValue(30);
  slide.setPaintTicks(true);
  slide.setPaintLabels(true);
```

```
        slide.setMinorTickSpacing(10);
        slide.setMajorTickSpacing(20);
        slide.addChangeListener(new ChangeListener(){
         public void stateChanged(ChangeEvent event){
          label.setText("Valeur actuelle : " +
      ((JSlider)event.getSource()).getValue());
         }
        });

        this.getContentPane().add(slide, BorderLayout.CENTER);
        this.getContentPane().add(label, BorderLayout.SOUTH);

       }

       public static void main(String[] args){
        Slide slide = new Slide();
        slide.setVisible(true);
       }

      }
```

La `JProgessBar`

Elle vous permet de réaliser une barre de progression pour des traitements longs.



Code source :

Secret (cliquez pour afficher)

Code : Java

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JProgressBar;

public class Progress extends JFrame{

 private Thread t;
 private JProgressBar bar;
```

```
    private JButton launch ;

    public Progress(){

     this.setSize(300, 80);
     this.setTitle("*** JProgressBar ***");
     this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     this.setLocationRelativeTo(null);


     t = new Thread(new Traitement());
     bar  = new JProgressBar();
     bar.setMaximum(500);
     bar .setMinimum(0);
     bar.setStringPainted(true);

     this.getContentPane().add(bar, BorderLayout.CENTER);

     launch = new JButton("Lancer");
     launch.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent event){
       t = new Thread(new Traitement());
       t.start();

      }
     });
     this.getContentPane().add(launch, BorderLayout.SOUTH);
     t.start();
     this.setVisible(true);
    }

    class Traitement implements Runnable{

     public void run(){
      launch.setEnabled(false);

      for(int val = 0; val <= 500; val++){
       bar.setValue(val);
       try {
        t.sleep(10);
       } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
       }
      }
      launch.setEnabled(true);
     }
    }

    public static void main(String[] args){
     Progress p = new Progress();
    }

   }
```

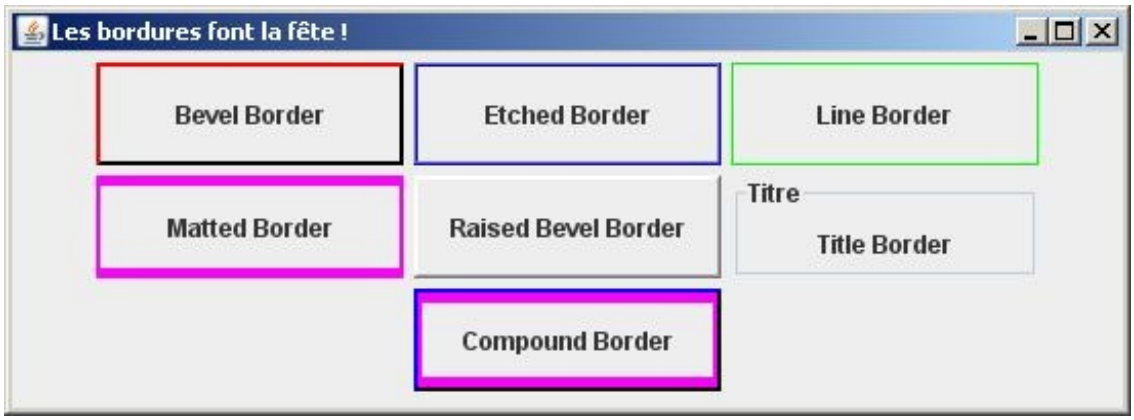
La modification des valeurs de cet objet doit se faire dans un thread, sinon vous aurez la barre vide, un temps d'attente puis la barre remplie, mais sans défilement des valeurs !

Retour en haut

Enjoliver vos IHM

Nous n'avons pas beaucoup abordé ce point tout au long du tuto, mais je vous laisse découvrir les joyeusetés qu'offre Java en la matière... C'est encore en cherchant les infos dont on a besoin qu'on les retient le mieux !

Voici comment ajouter des bordures à vos composants :



Code source :

Secret (cliquez pour afficher)

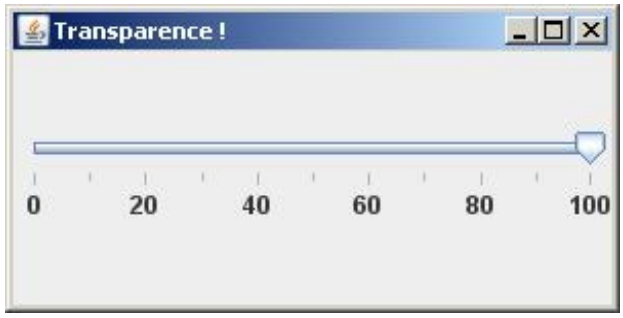
Code : Java

```
import java.awt.Color;
import java.awt.Dimension;

import javax.swing.BorderFactory;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.border.BevelBorder;
import javax.swing.border.Border;
import javax.swing.border.EtchedBorder;


public class BorderDemo extends JFrame{

 private String[] list = { "Bevel Border",
        "Etched Border",
        "Line Border",
        "Matted Border",
        "Raised Bevel Border",
        "Title Border",
        "Compound Border"
       };

 private Border[] listBorder = {
BorderFactory.createBevelBorder(BevelBorder.LOWERED, Color.black,
Color.red),
         BorderFactory.createEtchedBorder(Color.BLUE, Color.GRAY),
         BorderFactory.createLineBorder(Color.green),
         BorderFactory.createMatteBorder(5, 2, 5, 2,
Color.MAGENTA),
         BorderFactory.createRaisedBevelBorder(),
         BorderFactory.createTitledBorder("Titre"),
         BorderFactory.createCompoundBorder(
           BorderFactory.createBevelBorder(BevelBorder.LOWERED,
Color.black, Color.blue),
           BorderFactory.createMatteBorder(5, 2, 5, 2,
Color.MAGENTA)
          )
```

```
            };

        public BorderDemo(){

         this.setTitle("Les bordures font la fête !");
         this.setLocationRelativeTo(null);
         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         this.setSize(550, 200);

         JPanel pan = new JPanel();
         for(int i = 0; i < list.length; i++){

          JLabel lib = new JLabel(list[i]);
          lib.setPreferredSize(new Dimension(150, 50));
          lib.setBorder(listBorder[i]);
          lib.setAlignmentX(JLabel.CENTER);
          lib.setHorizontalAlignment(JLabel.CENTER);
          pan.add(lib);
         }

         this.getContentPane().add(pan);
        }



        public static void main(String[] args){
         BorderDemo demo = new BorderDemo();
         demo.setVisible(true);
        }

       }
```

Cadeau du chef pour ceux pour auront installé la version 1.6 du JDK

Jouer sur l'opacité de vos composants :



Code source :

Secret (cliquez pour afficher)

Code : Java

```
import java.awt.BorderLayout;
import java.lang.reflect.Method;

import javax.swing.JFrame;
import javax.swing.JSlider;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

import com.sun.awt.AWTUtilities;


public class OpacityWindow extends JFrame{

 private JSlider slide = new JSlider();

 public OpacityWindow(){

  this.setTitle("Transparence !");
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setSize(300, 150);

  slide.setMaximum(100);
  slide.setMinimum(0);
  slide.setValue(100);
  slide.setPaintTicks(true);
  slide.setPaintLabels(true);
  slide.setMinorTickSpacing(10);
  slide.setMajorTickSpacing(20);
  slide.addChangeListener(new Changed(this));


  this.getContentPane().add(slide, BorderLayout.CENTER);
  this.setVisible(true);
  AWTUtilities.setWindowOpacity(this, ((float) slide.getValue()) /
100.0f);
 }


 public class Changed implements ChangeListener{
  private JFrame frame;
  public Changed(JFrame frame){this.frame = frame;}
  public void stateChanged(ChangeEvent event){
   AWTUtilities.setWindowOpacity(this.frame, ((float)
slide.getValue()) / 100.0f);
  }
 }


 public static void main(String[] args){
```

```
        OpacityWindow ow = new OpacityWindow();
      }
    }
```

Si vous voulez en savoir plus, il y a un article très complet sur le sujet à cette adresse !
Retour en haut
J'espère que ce dernier chapitre vous a plu !
Vous avez encore du temps pour apprendre tout ça, ne vous découragez pas.

Voilà : la partie événementielle est terminée... Bonne continuation à tous les ZérOs qui sont arrivés jusqu'ici et qui ont réussi à suivre mon tuto !
Retour en haut

Vous en avez fini avec la partie événementielle.
Et le tuto s'arrête ici !

J'espère que vous avez appris tout plein de choses et que vous commencez à faire des choses sympa avec ce langage de programmation.

Il y a de quoi faire niveau IHM... Vous devez vous en rendre compte.
Retour en haut

Partie 4 : Les Design patterns

Ceci est en fait un complément du tuto Java mais vous devez savoir que les chapitres qui suivent peuvent s'adapter à beaucoup de langages (PHP, C#...).

Dans cette partie, vous pourrez voir et utiliser ce qu'on appelle des ***design patterns*** ou, en français, des modèles de conceptions. La conception est la phase qui intervient avant le codage, lorsque vous réfléchissez :

- aux classes dont vous aurez besoin ;
- à savoir lier telle classe avec telle autre ;
- au type d'interfaces que vous souhaitez créer ;
- au type de classes abstraites que vous désirez coder ;
- ...

Afin de schématiser ceci, vous pouvez faire des dessins, mais il y a mieux !
Dans la partie 2, je vous ai initiés ttrrèèèssssss rapidement à UML : nous allons l'utiliser de nouveau ici afin de schématiser les situations.

Vous constaterez qu'il y a de nombreux DP (j'utiliserai cette abréviation à partir de maintenant) et qu'ils peuvent être extrêmement pratiques.

Dîtes-vous bien que, malgré le temps que vous passez à coder, quelqu'un d'autre a rencontré, à un moment donné, les mêmes problèmes de conception que vous. Et vous en avez peut-être un en ce moment, sans même le savoir.
Rassurez-vous, car d'autres ont trouvé une solution :

- évolutive ;
- souple ;
- résistante.

L'un des buts des DP est de vous fournir des solutions afin de rendre vos codes moins assujettis à la modification. Je pense qu'après le premier chapitre vous comprendrez mieux ce que je veux dire... Mais il va y avoir une constante, dans cette partie !

Nous partirons du postulat que votre programme est amené à connaître des changements, majeurs ou mineurs. Ces changements pourront être de plusieurs natures mais au final, le but est d'avoir un code réutilisable et non modifiable : on dit aussi "*hermétique à la modification*".

Je me doute que vous devez être sceptique... Je vous propose donc de commencer !
Retour en haut

Les limites de l'héritage : le pattern strategy

Comme je vous le disais dans l'introduction de cette partie, nous allons partir du postulat que vous avez un code qui fonctionne, et par là j'entends un ensemble de classes objets liées par héritage, ou autre.

Nous allons voir dans ce chapitre que, malgré la toute puissance de l'héritage, celui-ci trouve ses limites lorsque vous êtes amenés à modifier vos hiérarchies de classes afin de répondre à une demande (votre chef, un client...).
Et le fait de toucher à votre hiérarchie peut amener des erreurs non désirables (si une erreur peut l'être...), et même des absurdités. Tout ceci dû au fait que vous allez changer une structure qui fonctionne à cause des contraintes que vous subissez.

Pour remédier à cela, il y a un concept simple : en fait, il s'agit d'un des fondements de la programmation orientée objet : **l'encapsulation !**

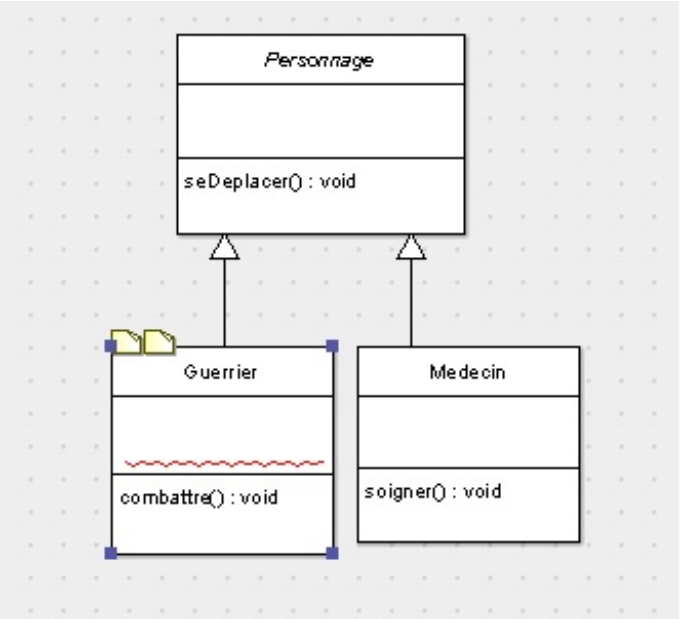
Dans ce chapitre nous parlerons de cette solution, que vous avez déjà vue sans le savoir : ceci si vous avez suivi la partie 3 du tuto.

Bon, il est temps d'y aller !

Posons le problème

Voici le tableau

Vous êtes un jeune et ambitieux développeur d'une toute nouvelle société qui crée des jeux vidéos.
Le dernier titre en date, Z-Army, un jeu de guerre très réaliste, a été un succès interplanétaire ! Votre patron est content et vous aussi.
Vous vous êtes pourtant basé sur une architecture vraiment simple afin de créer et utiliser des personnages (guerrier, médecin...).
D'ailleurs, la voici :

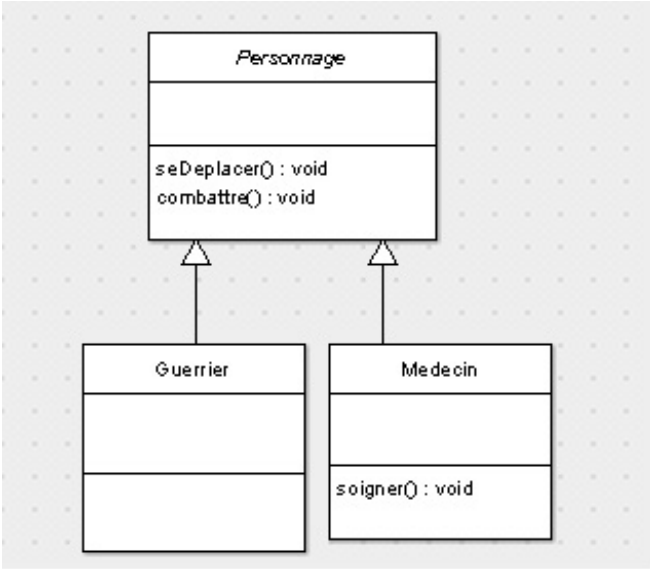


Pour ceux qui seraient totalement étrangers à UML, qu'ils fassent un tour dans le chapitre consacré à cet effet.
Bon, vous constatez que votre hiérarchie est très simple : la classe `Personnage` est une classe abstraite dont héritent les classes `Guerrier` et `Medecin`.
Les guerriers savent se battre tandis que les médecins soignent les blessés sur le champs de bataille !

Les ennuis commencent maintenant !
Votre patron vous a confié le projet ***Z-Army2 "The return of the revenge"*** .
Vous vous dites : yes !... Mon architecture fonctionne à merveille, je la garde.
Et vous commencez à créer le second volet du jeu.

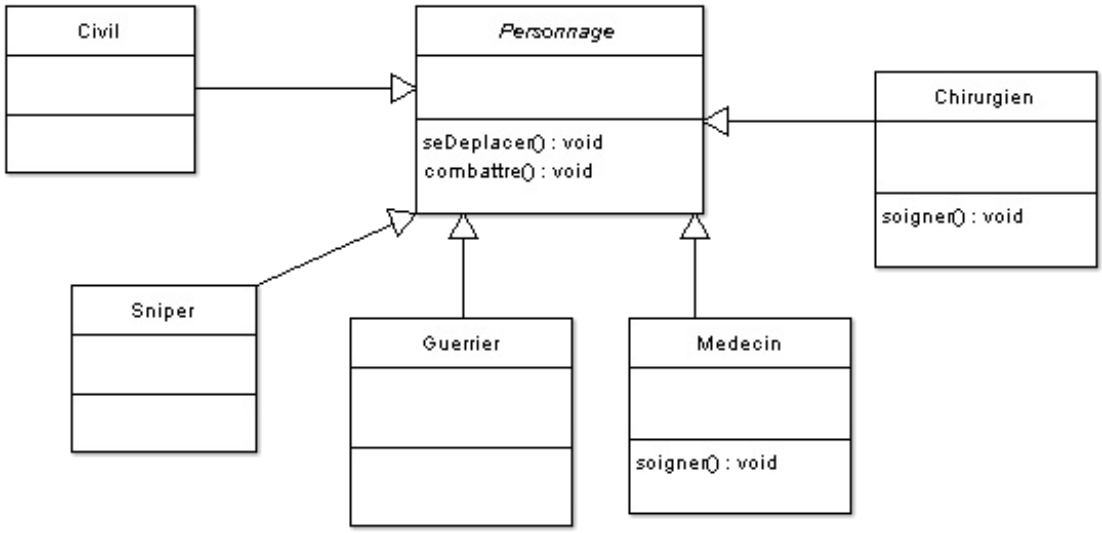
Un mois plus tard, votre patron vous convoque dans son bureau et vous dit : "***Nous avons fait une étude de marché, et il semblerait que les joueurs aimeraient se battre aussi avec les médecins !*** ".
Vous trouvez l'idée séduisante et vous avez déjà pensé à une solution : déplacer la méthode `combattre()` dans la super-classe `Personnage`, afin de pouvoir la redéfinir dans la classe `Medecin` et jouir du polymorphisme !

Votre diagramme de classe ressemble donc à ceci :



À la seconde étude de marché, votre patron vous annonce que vous allez devoir créer des civils, des snipers, des chirurgiens... Toute une panoplie de personnages spécialisés dans leur domaine !

Voici à présent votre diagramme de classe :



Le code source de ces classes

Personnage.java

Code : Java

```
public abstract class Personnage {

 /**
* Méthode de déplacement de personnage
*/
 public abstract void seDeplacer();
 /**
* Méthode que les combattants utilisent
*/
 public abstract void combattre();
}
```

Guerrier.java

Code : Java

```
public class Guerrier extends Personnage {

 public void combattre() {
  System.out.println("Fusil, pistolet, couteau ! Tout ce que tu veux
!");
 }

 public void seDeplacer() {
  System.out.println("Je me déplace à pied.");
 }
}
```

Medecin.java

Code : Java

```
public class Medecin extends Personnage{
 public void combattre() {
  System.out.println("Vive le scalpel !");
 }

 public void seDeplacer() {
  System.out.println("Je me déplace à pied.");
 }

        public void soigner(){
  System.out.println("Je soigne les blessures.");
 }
}
```

Civil.java

Code : Java

```
public class Civil extends Personnage{
 public void combattre() {
  System.out.println("Je ne combats PAS !");
 }

 public void seDeplacer() {
  System.out.println("Je me déplace à pied.");
 }
}
```

Chirurgien.java

Code : Java

```
public class Chirurgien extends Personnage{
 public void combattre() {
  System.out.println("Je ne combats PAS !");
 }

 public void seDeplacer() {
  System.out.println("Je me déplace à pied.");
 }

 public void soigner(){
  System.out.println("Je fais des opérations.");
 }
}
```

Sniper.java

Code : Java

```
public class Sniper extends Personnage{
 public void combattre() {
  System.out.println("Je me sers de mon fusil à lunette !");
 }

 public void seDeplacer() {
  System.out.println("Je me déplace à pied.");
 }
}
```

À ce stade, vous devriez remarquer que :

- le code contenu dans la méthode **`seDeplacer()`** est dupliqué dans toutes les classes ! Il est identique dans toutes celles citées ci-dessus ;
- le code de la méthode **`combattre()`** de la classe **`Chirurgien`** et **`Civil`** est lui aussi dupliqué !

La duplication de code est l'une des choses qui peuvent générer des problèmes dans le futur !
Je m'explique.
Pour le moment, votre chef ne vous a demandé que de créer quelques classes supplémentaires. Qu'en sera-t-il si plusieurs classes, qui n'ont que le seul lien d'héritage existant, ont ce même code dupliqué ? Il ne manquerait plus que votre chef vous demande de modifier à nouveau la façon de se déplacer de ces objets pour en oublier un, voire même plusieurs ! Et voilà les incohérences qui pointent le bout de leur nez...

No problemo ! Tu vas voir... Il suffit de mettre un comportement par défaut pour le déplacement et pour le combat dans la super-classe **`Personnage`**.

Effectivement, votre idée se tient. Donc, ceci nous donne ce qui suit...

Personnage.java

Code : Java

```
public abstract class Personnage {

 /**
* Méthode de déplacement de personnage
*/
 public void seDeplacer(){
  System.out.println("Je me déplace à pied.");
 }
 /**
* Méthode que les combattants utilisent
*/
 public void combattre(){
  System.out.println("Je ne combats PAS !");
 }
}
```

Guerrier.java

Code : Java

```
public class Guerrier extends Personnage {

 public void combattre() {
  System.out.println("Fusil, pistolet, couteau ! Tout ce que tu veux
!");
 }
}
```

Medecin.java

Code : Java

```
public class Medecin extends Personnage{
 public void combattre() {
  System.out.println("Vive le scalpel !");
 }

        public void soigner(){
  System.out.println("Je soigne les blessures.");
 }
}
```

Civil.java

Code : Java

```
public class Civil extends Personnage{
}
```

Chirurgien.java

Code : Java

```
public class Chirurgien extends Personnage{
 public void soigner(){
  System.out.println("Je fais des opérations.");
 }
}
```

Sniper.java

Code : Java

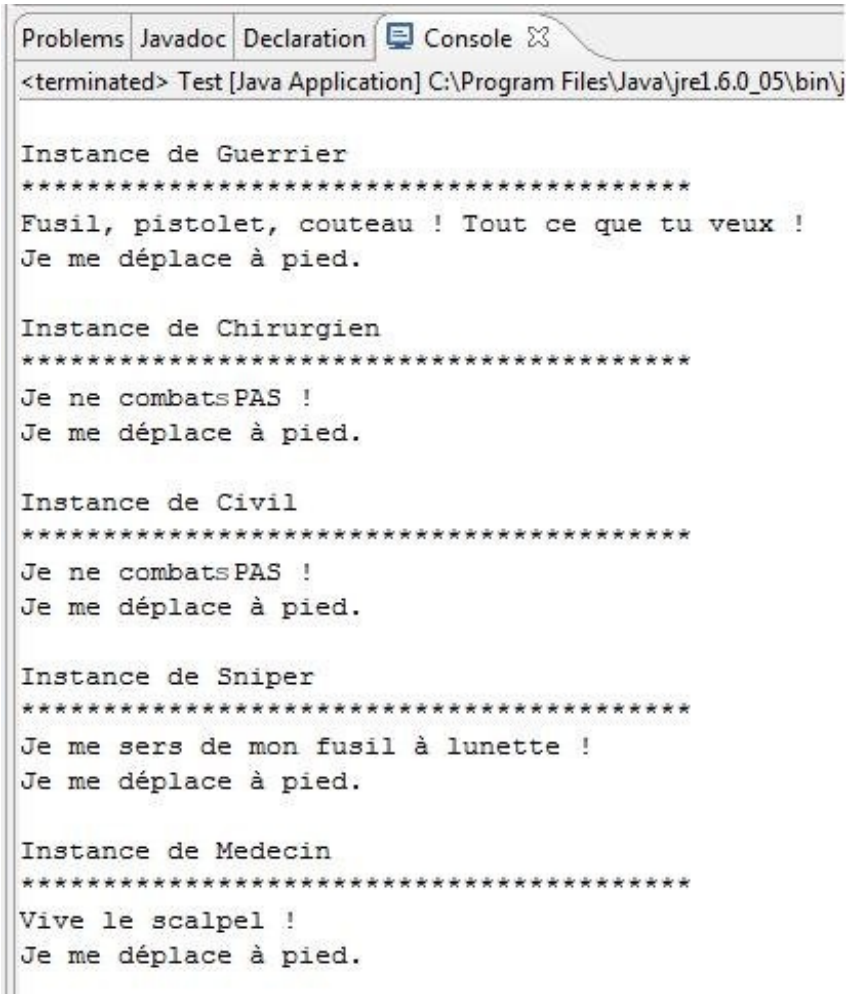
```
public class Sniper extends Personnage{
 public void combattre() {
  System.out.println("Je me sers de mon fusil à lunette !");
 }
}
```

Voici une classe contenant un petit programme afin de tester nos classes :

Code : Java

```
public static void main(String[] args) {
  Personnage[] tPers = {new Guerrier(), new Chirurgien(), new
Civil(), new Sniper(), new Medecin()};
  for(Personnage p : tPers){
   System.out.println("\nInstance de " + p.getClass().getName());
   System.out.println("*************************************");
   p.combattre();
   p.seDeplacer();
  }
 }
```

Et le résultat de ce code :



Apparemment, ce code vous donne ce que vous voulez !
Plus de redondance... Mais, personnellement, un problème me chiffonne. Vous ne pouvez pas utiliser les classes **Medecin** et **Chirurgien** de façon polymorphe, vu que la méthode **soigner()** leur est propre !

Alors, on définit un comportement par défaut (ne pas soigner) dans la super-classe **Personnage** et le tour est joué !

Code : Java

```
public abstract class Personnage {

 /**
* Méthode de déplacement de personnage
*/
 public void seDeplacer(){
  System.out.println("Je me déplace à pied.");
 }
 /**
* Méthode que les combattants utilisent
*/
 public void combattre(){
  System.out.println("Je ne combats PAS !");
 }
        /**
* Méthode de soin
*/
 public void soigner(){
  System.out.println("Je ne soigne pas.");
 }
}
```

Au même moment, votre chef rentre dans votre bureau et vous dit :
"***Nous avons bien réfléchi, et il serait de bon ton que nos guerriers puissent administrer les premiers soins*** ".
À ce moment précis, vous vous délectez de votre capacité d'anticipation ! Vous savez que maintenant, il vous suffit de redéfinir la méthode `soigner()` dans la classe concernée et tout le monde est content !

Seulement voilà ! Votre chef n'avait pas fini son speech...
"***Au fait, il faudrait adapter un comportement différent à nos personnages selon leurs armes, leurs habits, leurs trousses de soin... Enfin tu vois ! Les comportements figés pour des personnages de jeux, de nos jours... c'est un peu ringard !***"

Vous commencez à voir ce dont il retourne ! Vous allez apporter des modifications à votre code, encore et encore...

Problème : à chaque modification de comportement de vos personnages, vous êtes obligés de modifier le code source de la classe concernée !

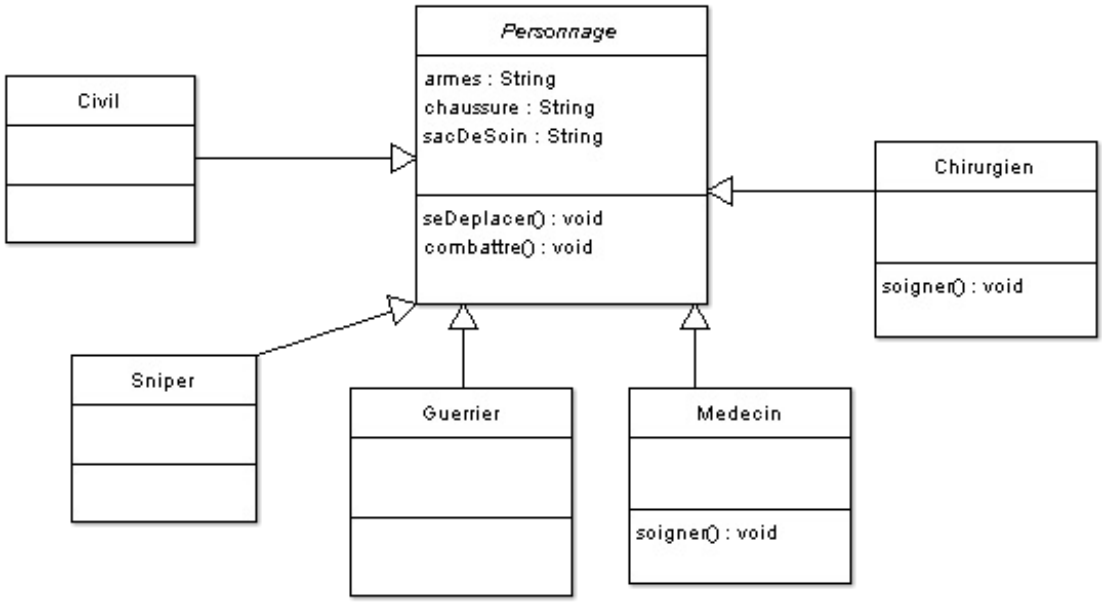
Bon : pour un programmeur, ceci est le train-train quotidien, j'en conviens.
Cependant, si nous suivons les consignes de notre chef et que nous continuons sur notre lancée, les choses vont se compliquer... Voyons voir.
Retour en haut

Un problème supplémentaire

Attelons-nous à appliquer les modifications dans notre programme.
Si nous suivons les consignes de notre chef, et c'est ce que nous allons faire, nous allons devoir gérer des comportements différents selon les accessoires de nos personnages.
En fait, nous pouvons utiliser des variables d'instance et utiliser celles-ci pour appliquer tel ou tel comportement.

Afin de simplifier l'exemple, nous n'allons utiliser que des objets `String`.

Voici le diagramme de classes de notre programme :



Vous avez remarqué que nos personnages vont avoir des accessoires. Selon ceux-ci, nos personnages feront des choses différentes.
Voici les recommandations de notre chef bien-aimé :

- le guerrier devra pouvoir utiliser un couteau, un pistolet ou un fusil de sniper ;
- le sniper peut utiliser son fusil de sniper mais aussi un fusil à pompe ;
- le médecin a une trousse simple pour soigner mais peut utiliser un pistolet;
- le chirurgien a une grosse trousse médicale mais ne peut pas utiliser d'arme ;
- le civil, quant à lui, peut utiliser un couteau seulement quand il en a un ;
- tous les personnages hormis le chirurgien peuvent avoir des baskets pour courir.

Il va nous falloir des accesseurs pour ces variables, mettons-les dans la super-classe !

Inutile de mettre les méthodes de renvoi (`get`), nous ne nous servirons que des mutateurs !

Bon, les modifications sont faites, les caprices de notre cher et tendre chef sont satisfaits ? Voyons ça tout de suite.

Personnage.java

Code : Java

```
public abstract class Personnage {

 protected String armes = "", chaussure = "", sacDeSoin = "";
 /**
* Méthode de déplacement de personnage
*/
```

```
 public void seDeplacer(){
  System.out.println("Je me déplace à pied.");
 }
 /**
* Méthode que les combattants utilisent
*/
 public void combattre(){
  System.out.println("Je ne combats PAS !");
 }
 /**
* Méthode de soin
*/
 public void soigner(){
  System.out.println("Je ne soigne pas.");
 }

 protected void setArmes(String armes) {
  this.armes = armes;
 }
 protected void setChaussure(String chaussure) {
  this.chaussure = chaussure;
 }
 protected void setSacDeSoin(String sacDeSoin) {
  this.sacDeSoin = sacDeSoin;
 }
}
```

Guerrier.java

Code : Java

```
public class Guerrier extends Personnage {

 public void combattre() {
  if(this.armes.equals("pistolet"))
   System.out.println("Attaque au pistolet !");
  else if(this.armes.equals("fusil de sniper"))
   System.out.println("Attaque au fusil de sniper !");
  else
   System.out.println("Attaque au couteau !");
 }
}
```

Sniper.java

Code : Java

```
public class Sniper extends Personnage{
 public void combattre() {
  if(this.armes.equals("fusil à pompe"))
   System.out.println("Attaque au fusil à pompe !");
  else
   System.out.println("Je me sers de mon fusil à lunette !");
 }
}
```

Civil.java

Code : Java

```
public class Civil extends Personnage{
 public void combattre(){
  if(this.armes.equals("couteau"))
   System.out.println("Attaque au couteau !");
  else
   System.out.println("Je ne combats PAS !");
 }
}
```

Medecin.java

Code : Java

```
public class Medecin extends Personnage{
 public void combattre() {
  if(this.armes.equals("pistolet"))
   System.out.println("Attaque au pistolet !");
  else
   System.out.println("Vive le scalpel !");
 }

 public void soigner(){
  if(this.sacDeSoin.equals("petit sac"))
   System.out.println("Je peux recoudre des blessures.");
  else
   System.out.println("Je soigne les blessures.");
 }
}
```

Chirurgien.java

Code : Java

```
public class Chirurgien extends Personnage{
 public void soigner(){
  if(this.sacDeSoin.equals("gros sac"))
   System.out.println("Je fais des merveilles.");
  else
   System.out.println("Je fais des opérations.");
 }
}
```

Voici un programme de test :

Code : Java

```
public static void main(String[] args) {
  Personnage[] tPers = {new Guerrier(), new Chirurgien(), new
```

```
        Civil(), new Sniper(), new Medecin()};
          String[] tArmes = {"pistolet", "pistolet", "couteau", "fusil à
        pompe", "couteau"};
          for(int i = 0; i < tPers.length; i++){
           System.out.println("\nInstance de " +
        tPers[i].getClass().getName());
           System.out.println("*****************************************");
           tPers[i].combattre();
           tPers[i].setArmes(tArmes[i]);
           tPers[i].combattre();
           tPers[i].seDeplacer();
           tPers[i].soigner();
          }
         }
```

Et le résultat de ce test :

```
Instance de Guerrier
*****************************************
Attaque au couteau !
Attaque au pistolet !
Je me déplace à pied.
Je ne soigne pas.

Instance de Chirurgien
*****************************************
Je ne combats PAS !
Je ne combats PAS !
Je me déplace à pied.
Je fais des opérations.

Instance de Civil
*****************************************
Je ne combats PAS !
Attaque au couteau !
Je me déplace à pied.
Je ne soigne pas.

Instance de Sniper
*****************************************
Je me sers de mon fusil à lunette !
Attaque au fusil à pompe !
Je me déplace à pied.
Je ne soigne pas.

Instance de Medecin
*****************************************
Vive le scalpel !
Vive le scalpel !
Je me déplace à pied.
Je soigne les blessures.
```

Vous constatez avec émerveillement que votre code fonctionne très bien. Les actions par défaut sont respectées, les affectations d'actions aussi. Tout est parfait !
Vraiment ? Vous êtes sûr de ça ?

Je ne vois pas ce qui cloche !

Pourtant, je vois du code dupliqué dans certaines classes !
En plus, nous n'arrêtons pas de modifier nos classes sans arrêt...
Lors de Z-Army1, celles-ci étaient pourtant très bien ! Qu'est-ce qui ne va pas ? Je ne comprends pas !

Là-dessus, votre patron rentre dans votre bureau pour vous dire que : "***les actions de vos personnages devront pouvoir êtres utilisables à la volée et, en fait, les personnages pouvaient très bien apprendre au fil du jeu...***"

Et là, inutile de demander un congé à votre patron pour cause de migraine !
Les changements s'accumulent, votre code devient de moins en moins lisible et réutilisable, bref, l'enfer sur terre.

Faisons un point sur la situation :

- du code dupliqué s'insinue dans votre code ;
- à chaque modification de comportement, vous êtes obligés de modifier le code source de la (ou des) classe(s) concernée(s) ;
- votre code perd en *réutilisabilité* et, du coup, votre code n'est pas extensible du tout !

Extensible ? Tu entends quoi par là ?

Par là j'entends que vos objets, sortis de leurs contexte, ne pourront plus être réutilisés. Ils auront été modelés pour l'application que vous êtes en train de programmer. Ceci est dû au fait que nous avons utilisé l'héritage à outrance...

Problème : le fait est que dans notre programme, les personnages sont liés entre eux mais ceux-ci ont des comportements tellement différents que nous ne savons pas quoi en faire ! Nous avons essayé de placer ces derniers à différents endroits dans notre hiérarchie mais le problème persiste, au final...

Voyons comment résoudre ce problème.
Retour en haut

Une solution simple et robuste : le pattern strategy

Après toutes ces émotions, vous allez enfin avoir une solution à ce problème de modification de code source !
Si vous vous souvenez de ce que j'ai dit dans l'introduction, un des fondements de la programmation orientée objet est : **l'encapsulation !**

Le pattern strategy est basé sur ce principe simple.

L'encapsulation est un mécanisme visant à rassembler des données et / ou des méthodes au sein d'une structure spécifique.
Ces méthodes et / ou ces données sont ainsi réutilisables partout ailleurs dans le programme.

Je me doute que cette phrase semble pompeuse... Mais remplacez "***une structure***" par "***un objet***".
Vous devez mieux comprendre le sens de cette phrase, non ?

Bon, vous avez compris que le pattern strategy consiste à créer des objets avec des données et / ou méthodes.

Oui, on comprend bien, mais lesquelles ?

Tout simplement ce qui change dans votre programme !

Le principe de base de ce pattern est le suivant :

isolez ce qui varie dans votre programme et encapsulez-le !

Désolé, mais on ne comprend toujours pas...

Pas de panique, nous allons y aller doucement.

Déjà, quels sont les éléments qui ne cessent de varier dans notre programme ?

- La méthode `combattre()` ;
- la méthode `seDeplacer()` ;
- la méthode `soigner()`.

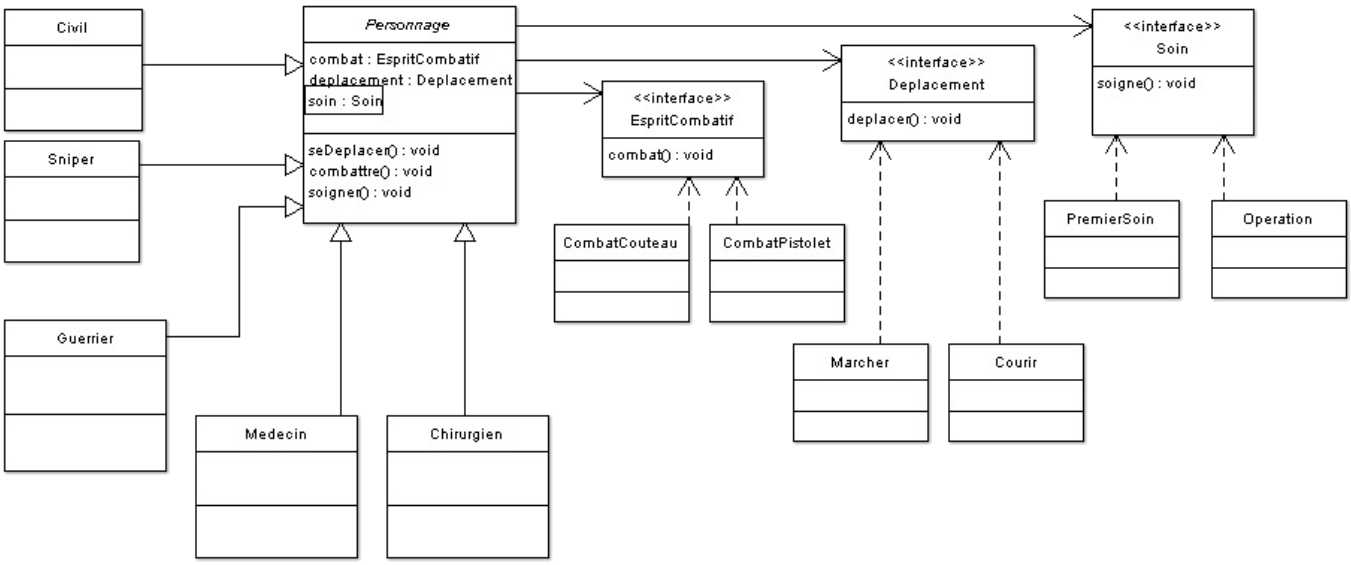
Nous avons tenté, en vain, de déployer ces comportements dans notre hiérarchie de classe, mais sans grand succès dû aux problèmes cités plus haut...

Ce qui serait vraiment grandiose, ce serait d'avoir la possibilité de ne modifier que les comportements et non les objets qui ont ces comportements !

Là, je vous arrête un moment. Vous venez de fournir la solution de vive voix. Vous avez dit : "***Ce qui serait vraiment grandiose, ce serait d'avoir la possibilité de ne modifier que les comportements et non les objets qui ont ces comportements***".

Lorsque je vous ai présenté les diagrammes UML, je vous ai fourni une astuce pour bien différencier les liens entre les objets.
Dans notre cas, nos classes héritant de `Personnage` héritent aussi des ses comportements et, par conséquent, on peut dire que nos classes filles sont des `Personnage`.
Concernant les comportements de la classe mère, ils semblent ne pas être au bon endroit dans la hiérarchie. Vous ne savez plus quoi en faire et vous vous demandez s'ils ont vraiment leur place dans cette classe ?

Il vous suffit de sortir ces comportements de la classe mère, de créer une classe abstraite ou une interface symbolisant ce comportement, et de dire à votre classe `Personnage` d'**avoir ces comportements**.
Voici mon nouveau diagramme de classes :



Ouh là ! Qu'est-ce que c'est que toutes ces interfaces ?

Je me doutais un peu que vous fronceriez les sourcils...
N'oubliez pas que votre code doit être souple et robuste et, même si ce chapitre vous montre les limites de l'héritage, n'oubliez pas que le polymorphisme est inhérent à l'héritage (et aux implémentations d'interfaces).
Il faut que vous vous rendiez compte qu'utiliser une interface de cette manière revient à créer un super-type de variable et, du coup, nous pourrons utiliser les classes héritant de ces interfaces de façon polymorphe, sans se soucier de savoir de quelle classe nos objets sont issus !
Dans notre cas, nous allons avoir des objets de type `EspritCombatif`, `Soin` et `Deplacement` dans notre classe `Personnage` !

Nous pouvons résumer la situation comme ceci : dans nos hiérarchies de classes, il est parfois préférable de privilégier la composition (= "a un") à l'héritage (= "est un") *[cf. chapitre sur UML]*.
Les comportements susceptibles d'être trop difficiles à généraliser dans la classe mère pourront ainsi être isolés en créant un nouveau type d'objet (encapsulation) correspondant à chaque comportement !

Avant de nous lancer dans le codage de nos nouvelles classes, vous devez vous rendre compte que leur nombre a considérablement augmenté depuis le début de ce chapitre.
Afin de pouvoir y voir plus clair et ainsi gagner en clarté, nous allons gérer nos différentes classes avec différents `packages`.

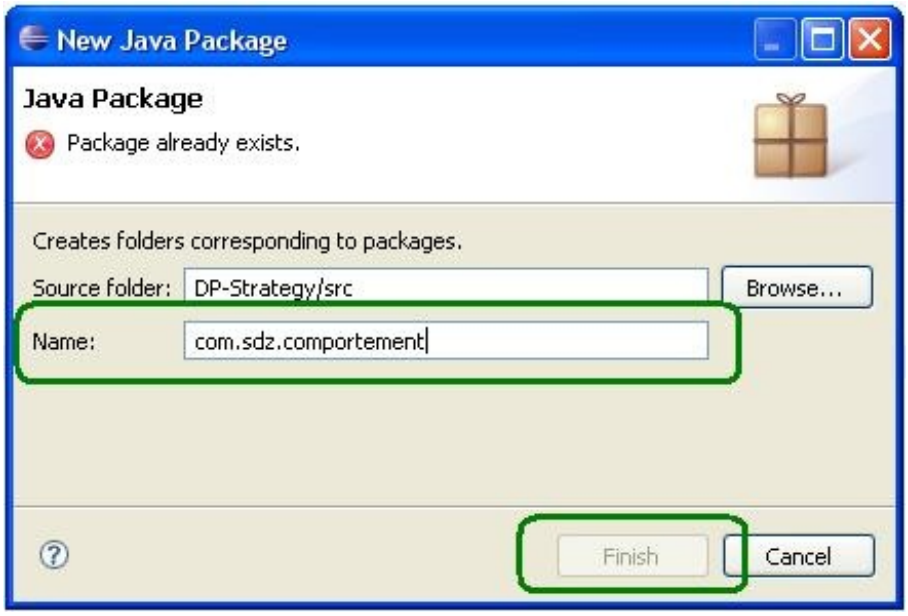
Comme je vous l'avais déjà dit dans un précédent chapitre, un package est un dossier comprenant plusieurs classes ou dossiers en son sein. Les classes sont regroupées par utilité ou par thème.

L'un des avantages de faire ceci est que nous allons gagner en lisibilité dans notre package par défaut mais aussi que les classes mises dans un package sont plus facilement transportables d'une application à l'autre. Pour cela, il vous suffira d'inclure le dossier de votre package dans un projet et d'importer les classes qui vous intéressent !

Comment crée-t-on un nouveau package ?

Ah, ce n'est pas difficile du tout, il vous suffit de cliquer sur cette icône :

Une boîte va s'ouvrir vous demandant le nom de votre package :



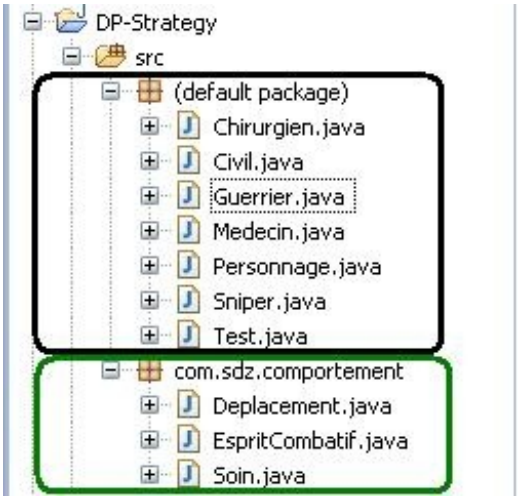
Attention : il existe aussi une convention de nommage pour les packages !

- Ceux-ci doivent être écrits entièrement en minuscules.
- Les caractères doivent être de a à z, de 0 à 9 et un point (.).
- Sun indique que tout package doit commencer par : com, edu, gov, mil, net, org ou les deux lettres identifiants un pays (ISO Standard 3166, 1981) donc, fr => France, eng => England...

Nous sommes sur le Site du Zér0, j'ai donc pris le nom à l'envers : sdz.com => com.sdz.
Par exemple, mes packages ont tendance à s'appeler **com.cysboy.<nom>**.

Bon, fin de l'aparte.

Cliquez sur "**Finish**" pour créer le package. Ensuite, il ne vous reste plus qu'à créer les interfaces de notre diagramme de classe dans ce package (clic droit dessus, "**new / interface**").
Et voilà ! Votre package est prêt à l'emploi :



Ce sera donc dans ce package que nous allons développer nos comportements !

Vous pouvez très bien déclarer un package par comportement... Comme si vous rangiez des dossiers dans un classeur !

Doucement ! Tu ne pourrais pas plutôt nous expliquer un peu plus tout ce mic-mac avec tes interfaces...

J'allais justement le faire.
Comme nous l'avons remarqué tout au long de ce chapitre, les comportements de nos personnages sont trop épars pour être définis dans notre super-classe **Personnage**. Vous l'avez dit vous-mêmes, il faudrait que l'on ne puisse modifier que les comportements et non les classes héritant de notre super-classe !
Les interfaces nous servent à créer un super-type d'objet ; ainsi, nous utiliserons des objets de type :

- **EspritCombatif** qui ont une méthode **combat()** ;
- **Soin** qui ont une méthode **soigne()** ;
- **Deplacement** qui ont une méthode **deplace()**.

Dans notre classe **Personnage**, nous avons ajouté une instance de chaque type de comportement, vous avez dû le remarquer : il y a ces attributs dans notre schéma !
Nous allons développer un comportement par défaut pour chaque type de comportement et nous allons affecter cet objet dans notre super-classe. Les classes filles, elles, auront des instances différentes, correspondant à leur besoin.

Du coup, que fait-on des méthodes de la super-classe **Personnage** ?

Nous les gardons, mais, au lieu d'avoir une redéfinition de ces dernières, la super-classe va invoquer la méthode de comportement de chaque objet. Ainsi, nous n'avons plus à redéfinir ou à modifier nos classes ! La seule chose qu'il vous reste à faire, c'est d'affecter une instance de comportement à vos objets.

Vous comprendrez mieux avec un exemple. Voici quelques implémentations de comportements.

*Implémentations de l'interface **EspritCombatif***

Code : Java

```
package com.sdz.comportement;

public class Pacifiste implements EspritCombatif {
 public void combat() {
  System.out.println("Je ne combats pas ! ");
 }
}
```

Code : Java

```
package com.sdz.comportement;

public class CombatPistolet implements EspritCombatif{
 public void combat() {
  System.out.println("Je combats au pitolet !");
 }
}
```

Code : Java

```
package com.sdz.comportement;

public class CombatCouteau implements EspritCombatif {
 public void combat() {
  System.out.println("Je me bats au couteau !");
 }
```

```
}
```

Implémentations de l'interface ***Deplacement***

Code : Java

```
package com.sdz.comportement;

public class Marcher implements Deplacement {
 public void deplacer() {
  System.out.println("Je me déplace en marchant.");
 }
}
```

Code : Java

```
package com.sdz.comportement;

public class Courir implements Deplacement {
 public void deplacer() {
  System.out.println("Je me déplace en courant.");
 }
}
```

Implémentations de l'interface ***Soin***

Code : Java

```
package com.sdz.comportement;

public class PremierSoin implements Soin {
 public void soigne() {
  System.out.println("Je donne les premiers soins.");
 }
}
```

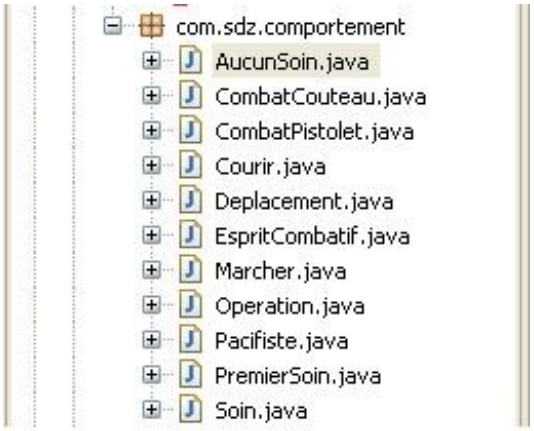
Code : Java

```
package com.sdz.comportement;

public class Operation implements Soin {
 public void soigne() {
  System.out.println("Je pratique des opérations !");
 }
}
```

Code : Java

```java
package com.sdz.comportement;

public class AucunSoin implements Soin {
 public void soigne() {
  System.out.println("Je ne donne AUCUN soin !");
 }
}
```

Voici ce que vous devriez avoir dans votre nouveau package :



Les classes mises dans un package et destinées à être utilisées à l'extérieur du dit package **DOIVENT être déclarées `public` !!**
Sinon, les classes ne seront visible qu'à l'intérieur du package et vous ne pourrez pas les utiliser !

Maintenant que nous avons défini des objets de comportements, nous allons pouvoir remanier notre classe **`Personnage`**.
Nous allons ajouter les variables d'instances, des mutateurs et des constructeurs afin de pouvoir initialiser nos objets.

Voici la nouvelle version de notre super-classe :

Code : Java

```java
import com.sdz.comportement.*;

public abstract class Personnage {

 //Nos instances de comportements
 protected EspritCombatif espritCombatif = new Pacifiste();
 protected Soin soin = new AucunSoin();
 protected Deplacement deplacement = new Marcher();

 /**
* Constructeur par défaut
*/
 public Personnage(){}

 /**
* Constructeur avec paramètres
* @param espritCombatif
* @param soin
* @param deplacement
*/
 public Personnage(EspritCombatif espritCombatif, Soin soin,
   Deplacement deplacement) {
  this.espritCombatif = espritCombatif;
  this.soin = soin;
  this.deplacement = deplacement;
```

```
    }
    /**
   * Méthode de déplacement de personnage
   */
    public void seDeplacer(){
     //On utilise les objets de déplacement de façon polymorphe
     deplacement.deplacer();
    }
    /**
   * Méthode que les combattants utilisent
   */
    public void combattre(){
     //On utilise les objets de déplacement de façon polymorphe
     espritCombatif.combat();
    }
    /**
   * Méthode de soin
   */
    public void soigner(){
     //On utilise les objets de déplacement de façon polymorphe
     soin.soigne();
    }

    //*************************************************************
    // ACCESSEURS
    //*************************************************************

    /**
   * Redéfinit le comportement au combat
   * @param espritCombatif
   */
    protected void setEspritCombatif(EspritCombatif espritCombatif) {
     this.espritCombatif = espritCombatif;
    }
    /**
   * Redéfinit le comportement de Soin
   * @param soin
   */
    protected void setSoin(Soin soin) {
     this.soin = soin;
    }
    /**
   * Redéfinit le comportement de déplacement
   * @param deplacement
   */
    protected void setDeplacement(Deplacement deplacement) {
     this.deplacement = deplacement;
    }
   }
```

Il y a eu du changement depuis le début... Mais maintenant, nous n'utilisons plus des méthodes définies dans notre hiérarchie de classe, mais des implémentations concrètes d'interfaces !
Les méthodes que nos objets appellent utilisent chacune d'elle un objet de comportement. Nous pouvons donc définir des guerriers, des civils, des médecins... tous personnalisables puisqu'il suffit de changer leurs instances de comportements pour que les comportements de ceux-ci changent instantanément. La preuve en image.

Je ne vais pas vous donner les codes de toutes les classes... En voici seulement quelques-unes.

Guerrier.java

Code : Java

```
import com.sdz.comportement.*;

public class Guerrier extends Personnage {
```

```
 /**
* Constructeur par défaut
*/
 public Guerrier(){
  this.espritCombatif = new CombatPistolet();
 }
 /**
* Constructeur personnalisé
* @param espritCombatif
* @param soin
* @param deplacement
*/
 public Guerrier(EspritCombatif espritCombatif, Soin soin,
   Deplacement deplacement) {
  //Appel au constructeur de la super classe
  super(espritCombatif, soin, deplacement);
 }
}
```

Civil.java

Code : Java

```
import com.sdz.comportement.*;

public class Civil extends Personnage{

 public Civil() {}

 public Civil(EspritCombatif espritCombatif, Soin soin,
   Deplacement deplacement) {
  super(espritCombatif, soin, deplacement);
 }

}
```

Medecin.java

Code : Java

```
import com.sdz.comportement.*;

public class Medecin extends Personnage{

 public Medecin() {
  this.soin = new PremierSoin();
 }

 public Medecin(EspritCombatif espritCombatif, Soin soin,
   Deplacement deplacement) {
  super(espritCombatif, soin, deplacement);
 }

}
```

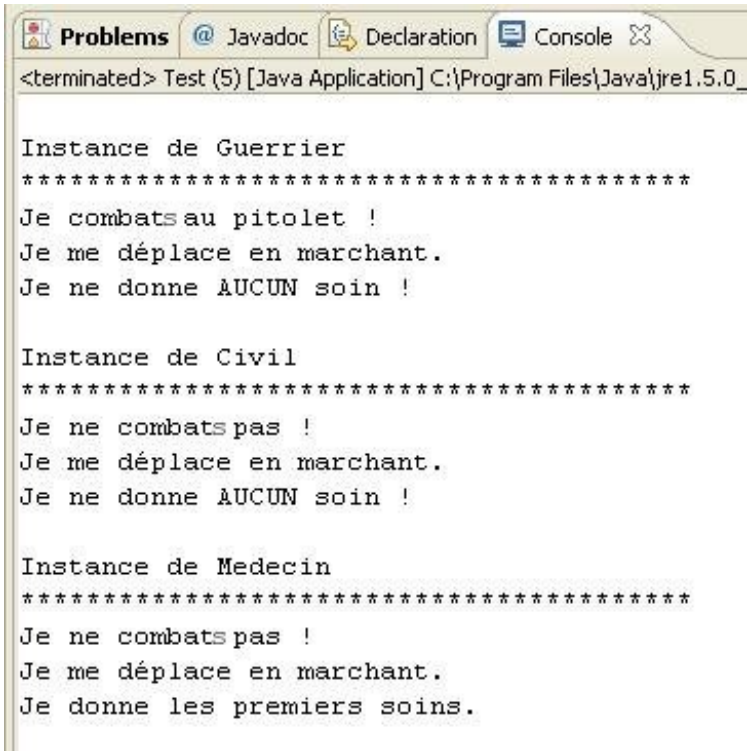
N'oubliez pas d'importer le package contenant nos classes de comportement !

Maintenant, voici un exemple d'utilisation :

Code : Java

```
class Test{
 public static void main(String[] args) {
  Personnage[] tPers = {new Guerrier(), new Civil(), new Medecin()};

  for(int i = 0; i < tPers.length; i++){
   System.out.println("\nInstance de " +
tPers[i].getClass().getName());
   System.out.println("****************************************");
   tPers[i].combattre();
   tPers[i].seDeplacer();
   tPers[i].soigner();
  }
 }
}
```

Le résultat de ce code nous donne :



Vous pouvez voir que nos personnages ont tous un comportement par défaut qui leur conviennent bien !
Nous avons spécifié, dans le cas où c'est nécessaire, le comportement par défaut d'un personnage dans son constructeur par défaut :

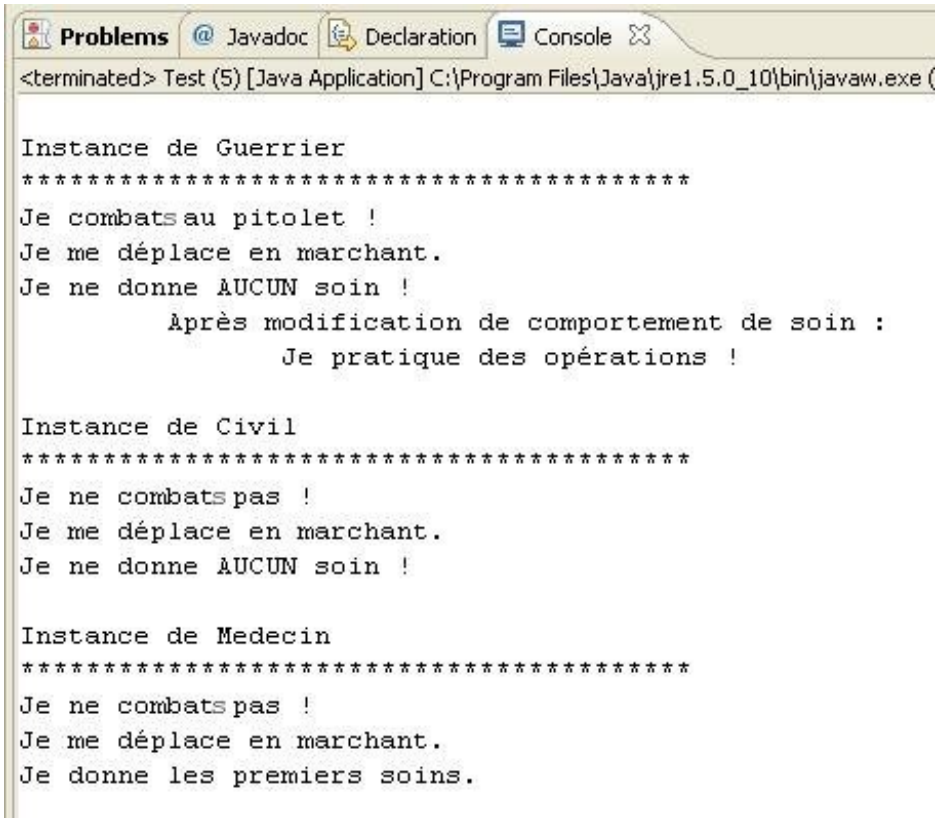
- le guerrier se bat avec un pistolet ;
- le médecin soigne.

Or, voyons comment dire à nos personnages de faire autre chose... Que diriez-vous de faire faire une petite opération chirurgicale à notre objet `Guerrier` ? Pour ce faire, vous pouvez redéfinir son comportement de soin avec son mutateur, présent dans la super-classe :

Code : Java

```
import com.sdz.comportement.*;


class Test{
 public static void main(String[] args) {
  Personnage[] tPers = {new Guerrier(), new Civil(), new Medecin()};

  for(int i = 0; i < tPers.length; i++){
   System.out.println("\nInstance de " +
tPers[i].getClass().getName());
   System.out.println("*****************************************");
   tPers[i].combattre();
   tPers[i].seDeplacer();
   tPers[i].soigner();
   if(tPers[i].getClass().getName().equals("Guerrier")){
    tPers[i].setSoin(new Operation());
    System.out.print(" \t Après modification de comportement de soin
: \n \t\t");
    tPers[i].soigner();
   }
  }
 }
}
```

Ce qui nous donne :



Vous voyez que le comportement de soin de notre objet a changé dynamiquement, sans que nous ayons besoin de changer la moindre ligne de son code source !

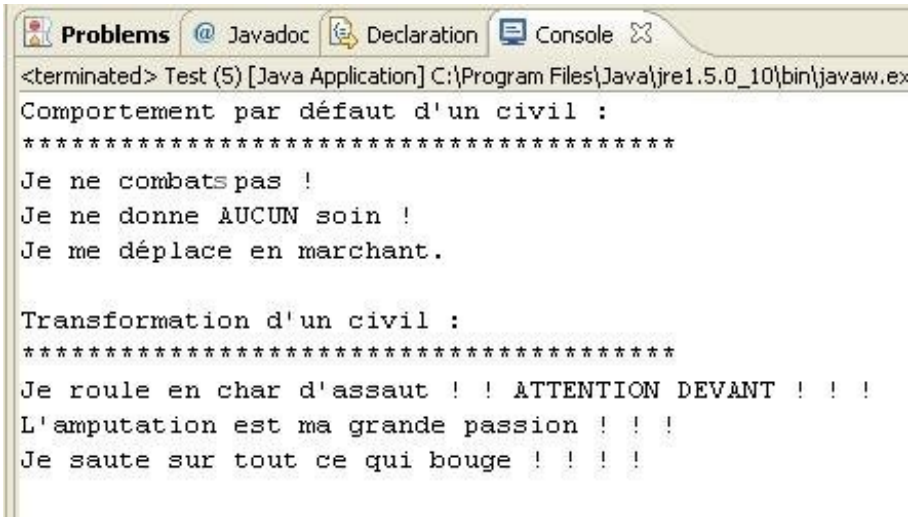
Le plus beau dans le fait de travailler comme ceci, c'est que vous pouvez tout à fait instancier des **`Guerrier`** avec des comportements différents très simplement, mais vous pouvez aussi leur donner des comportements que vous codez à la volée !

Regardez ceci :

Code : Java

```
import com.sdz.comportement.*;


class Test{
 public static void main(String[] args) {
  Personnage civil = new Civil();
  System.out.println("Comportement par défaut d'un civil : ");
  System.out.println("**************************************");
  civil.combattre();
  civil.soigner();
  civil.seDeplacer();

  System.out.println("\nTransformation d'un civil : ");
  System.out.println("**************************************");
  civil.setDeplacement(new Deplacement(){
   public void deplacer() {
    System.out.println("Je saute sur tout ce qui bouge ! ! ! !");
   }
  });
  civil.setSoin(new Soin(){
   public void soigne() {
    System.out.println("L'amputation est ma grande passion ! ! !");
   }
  });
  civil.setEspritCombatif(new EspritCombatif(){
   public void combat() {
    System.out.println("Je roule en char d'assaut ! ! ATTENTION
DEVANT ! ! !");
   }
  });

  civil.combattre();
  civil.soigner();
  civil.seDeplacer();

 }
}
```

Ce qui donne, au final :



Vous avez pu constater que vous n'avez plus de code dupliqué !

Les modifications de comportement deviennent très simples à faire et vous n'avez plus à modifier le code source de votre classe **`Personnage`** en cas de changements...

Je suppose que, maintenant que vous avez vu cet exemple, vous avez deviné où et quand vous avez utilisé le pattern strategy ! Lorsque vous programmiez des implémentations de `ActionListener` pour la gestion de vos événements...

Sauf que, dans ce cas, il y a une nuance. Vous avez utilisé le pattern strategy pour créer des comportements lors d'événements sur votre IHM, mais ces interfaces de gestion d'événements sont utilisées dans un autre pattern : **le pattern observer !**
Nous aborderons ce dernier très bientôt...

Bon, je crois qu'un petit topo s'impose...
Retour en haut

Ce qu'il faut retenir

- Les design pattern sont des modèles de conception permettant de créer des programmes souples et faciles à maintenir.
- Le pattern strategy permet de rendre une hiérarchie hermétique à la modification tout en lui permettant d'avoir des comportements différents.
- La base de ce DP réside dans l'encapsulation.
- Vous devez isoler les parties qui ont tendance à trop changer dans vos codes et les encapsuler.
- L'action ci-dessus permet de ne pas avoir de code dupliqué dans vos applications car les comportements sont encapsulés !
- Ceci peut être résumé ainsi : dans certains cas, vous devrez préférer la composition (= "a un") à l'héritage (= "est un").

Retour en haut

Voici un chapitre qui a dû vous montrer l'héritage sous un autre jour !
Vous pouvez voir qu'il y a des solutions simples à utiliser et qui vous permettent d'avoir un code "***hermétique à la modification***".

Le but final, c'est de n'avoir à modifier que l'endroit dans lequel vous utilisez vos objets et non vos objets eux-mêmes, et ce pattern fait ça très bien !

Nous avons vu comment faire en sorte de modifier les comportements de vos objets de façon dynamique, nous allons maintenant voir comment rajouter des fonctionnalités à vos objets dynamiquement.
En route pour **le pattern decorator !**
Retour en haut

Ajouter des fonctionnalités dynamiquement à vos objets : le pattern decorator

Après votre initiation lors du chapitre précédent, nous allons continuer avec un autre pattern très utilisé.
Celui-ci est utilisé dans une hiérarchie de classes Java !

Ne vous laissez pas abuser, vous verrez que plusieurs DP sont utilisés dans le langage Java. Je vais m'efforcer de vous les expliquer et de faire le rapprochement avec le langage...

Premier point important : rappelez-vous que ce qui fait la force des DP, c'est de pouvoir avoir des classes hermétiques à la modification mais capables de s'adapter automatiquement !

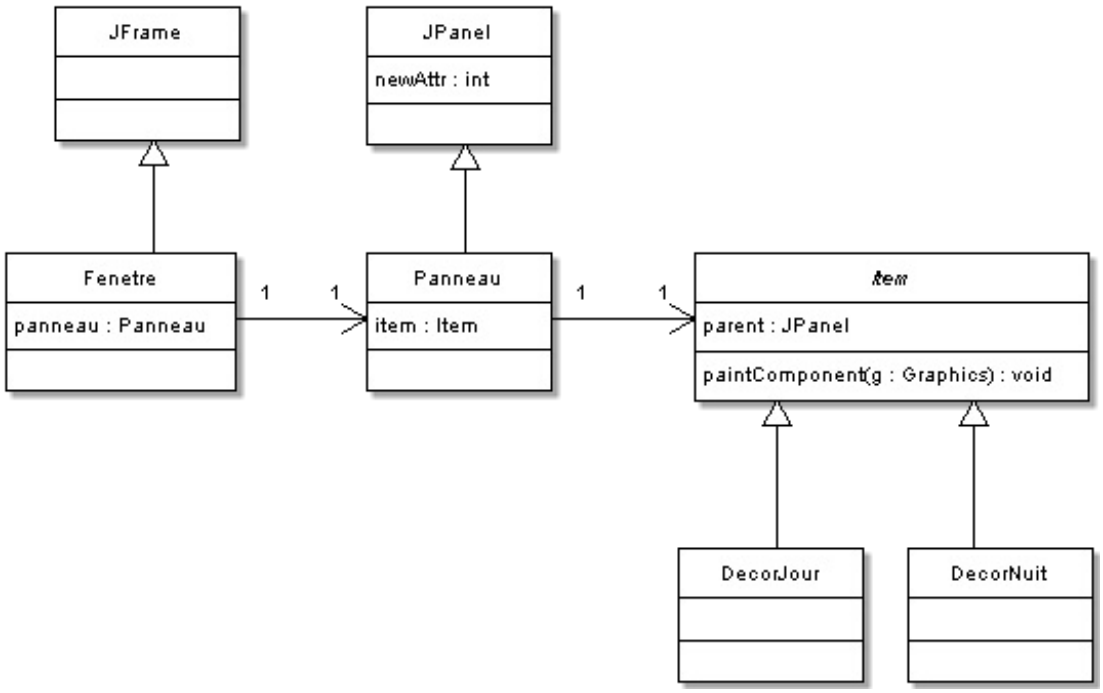
Dans ce chapitre, nous allons voir qu'il est possible de rajouter des fonctionnalités à vos objets de façon dynamique, donc, sans modifier la moindre ligne de code source dans l'objet utilisant la dite fonctionnalité.

L'exemple que j'ai choisi est très simple, vous verrez...

Je sens que vous devez être impatients de commencer...

Posons le problème

Vous êtes toujours un jeune développeur plein d'avenir dans un société de jeux vidéo.
Seulement, cette fois, vous devez créer un programme permettant de créer des décors. Vous avez déjà fait un premier jet de code qui semble très prometteur ! Voici le diagramme de classe de votre hiérarchie :



Cette structure doit vous être familière, maintenant... Nous avons une fenêtre héritée de `JFrame` qui a un objet hérité, lui, de `JPanel`. Ce dernier a un objet permettant de dessiner un décor de fond, qui n'est autre qu'une implémentation de l'interface `Item` (pattern strategy), afin de prévoir les modifications futures !

Voici un exemple de code représentant cette hiérarchie :

Secret (cliquez pour afficher)

Fenetre.java

Code : Java

```
import javax.swing.JFrame;

public class Fenetre extends JFrame {

 public Fenetre(){
  this.setSize(300, 300);
  this.setLocationRelativeTo(null);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setTitle("Decorateur");
  this.setResizable(false);
  this.setContentPane(new Panneau());
 }

 public static void main(String[] args){
  Fenetre fen =  new Fenetre();
  fen.setVisible(true);
 }
}
```

Panneau.java

Code : Java

```
import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JPanel;

import com.sdz.decorator.DecorJour;
import com.sdz.decorator.Item;


public class Panneau extends JPanel {

 private Item decor;

 /**
* @param decor
*/
 public Panneau(Item decor) {
  super();
  this.decor = decor;
 }
 /**
* Par défaut
*/
 public Panneau(){
  this.decor = new DecorJour(this);
 }

 public void paintComponent(Graphics g){
  this.decor.paintComponent(g);
 }
}
```

Item.java

Code : Java

```
package com.sdz.decorator;

import java.awt.Graphics;

import javax.swing.JPanel;

public abstract class Item {

 /**
* conteneur parent
*/
 protected JPanel parent;

 /**
* Constructeur avec paramètres
* @param width
* @param height
*/
 public Item(JPanel pan) {
  this.parent = pan;
 }

 /**
```

```
 * Constructeur par défaut
 */
  public Item() {}

  public void paintComponent(Graphics g){
   this.parent.getGraphicsConfiguration();
  }
}
```

DecorJour.java

Code : Java

```
package com.sdz.decorator;

import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JPanel;

public class DecorJour extends Item {

  /**
 *
 */
  public DecorJour() {
   super();
  }

  /**
 * @param width
 * @param height
 */
  public DecorJour(JPanel pan) {
   super(pan);
  }

  public void paintComponent(Graphics g) {
   super.paintComponent(g);
   //Ciel bleu
   g.setColor(Color.blue);
   g.fillRect(0, 0, this.parent.getWidth(), this.parent.getHeight()
- this.parent.getHeight()/3);
   //Pelouse
   g.setColor(Color.green);
   g.fillRect(0, this.parent.getHeight() -
this.parent.getHeight()/3, this.parent.getWidth(),
this.parent.getHeight());
   //Le soleil
   g.setColor(Color.yellow);
   g.fillOval(this.parent.getWidth() - this.parent.getWidth()/4,
this.parent.getWidth()/12, this.parent.getWidth()/6,
this.parent.getWidth()/6);

  }
}
```

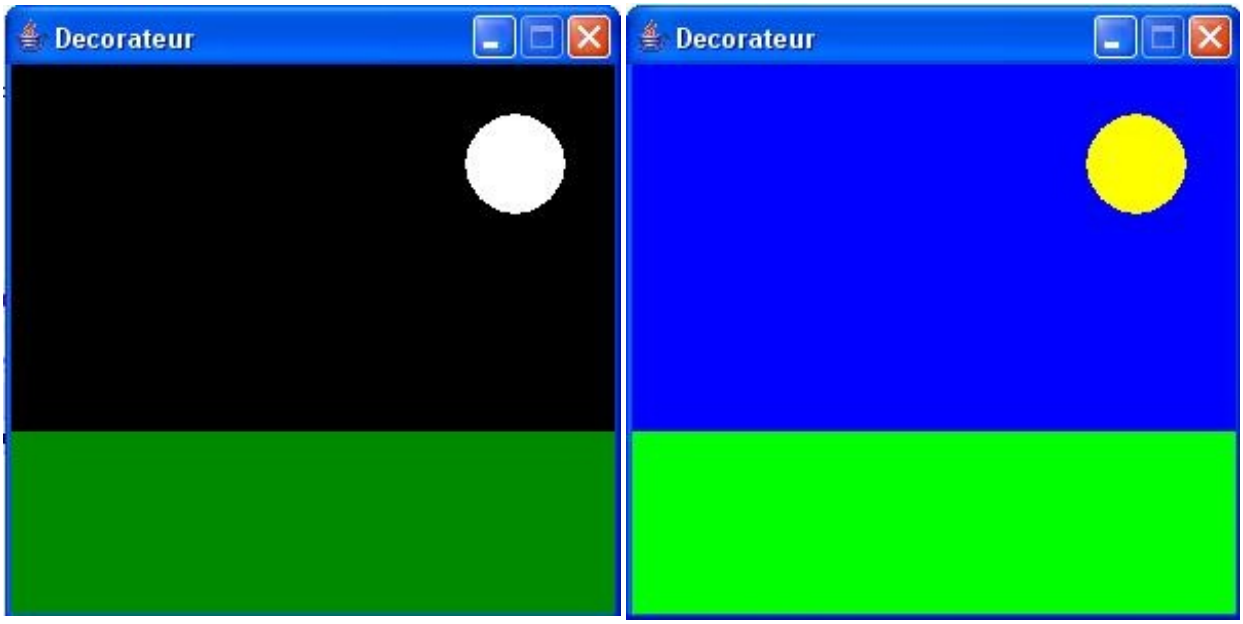
DecorNuit.java

Code : Java

```
package com.sdz.decorator;

import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JPanel;

public class DecorNuit extends Item {
 /**
*
*/
 public DecorNuit() {
  super();
 }

 /**
* @param pan
*/
 public DecorNuit(JPanel pan) {
  super(pan);
 }

 public void paintComponent(Graphics g) {
  super.paintComponent(g);
  //La nuit
  g.setColor(Color.black);
  g.fillRect(0, 0, this.parent.getWidth(), this.parent.getHeight()
- this.parent.getHeight()/3);
  //Pelouse
  g.setColor(new Color(0, 138, 0));
  g.fillRect(0, this.parent.getHeight() -
this.parent.getHeight()/3, this.parent.getWidth(),
this.parent.getHeight());
  //La lune
  g.setColor(Color.white);
  g.fillOval(this.parent.getWidth() - this.parent.getWidth()/4,
this.parent.getWidth()/12, this.parent.getWidth()/6,
this.parent.getWidth()/6);

 }
}
```

Vous devriez comprendre ce code sans problème si vous avez lu la partie 3 du tutoriel !
La seule chose qui change, c'est l'appel à la méthode `getGraphicsConfiguration()` qui donne l'autorisation de peindre dans notre composant depuis l'extérieur.

Voici le rendu des deux classes dérivant de la classe **`Item`** :



La différence entre les deux affichages est minime... Les couleurs changent, c'est tout...
Maintenant, afin de ne pas trop compliquer les choses, nous n'allons travailler que sur l'ajout de fonctionnalités sur la classe **`DecorJour`**. Ce que nous voulons faire, c'est trouver un moyen simple et efficace de pouvoir ajouter des éléments à notre décor :

- un arbre ;
- un nuage ;
- ...

Vu que vous connaissez le pattern strategy, vous pouvez trouver une méthode simple : créer une collection pouvant contenir plusieurs objets de type **`Item`** et balayer celle-ci lors de l'appel à la méthode `painComponent(Graphics g)` de votre objet **`DecorJour`**. Cette façon de faire est bonne mais il y a une autre façon de faire...
Et si je vous dit en prime qu'il y a un moyen d'arriver à faire ceci sans utiliser de boucle et sans modifier le code source de notre classe **`DecorJour`** ?

On serait curieux de savoir comment tu vas t'y prendre !

Tout simplement en utilisant le pattern decorator. Vous allez voir que celui-ci peut s'avérer très utile dans certaines situations !
Retour en haut

Le pattern decorator

Vous avez vu, lors du chapitre précédent, que la composition (= "a un") est souvent préférable à l'héritage (= "est un") : vous aviez défini de nouveaux comportements pour vos objets en créant un super-type d'objet par comportement.
Ce pattern aussi utilise la composition comme principe de base ! En fait, au final, vous allez voir que nos objets seront composés d'autres objets. La différence résidera dans le fait que nos nouvelles fonctionnalités ne seront pas obtenues uniquement en créant de nouveaux objets , **mais en associant ceux-ci avec des objets existants**.
Ce sera cette association qui créera de nouvelles fonctionnalités !

Tout ça a l'air bien beau, mais on ne comprend pas grand-chose...

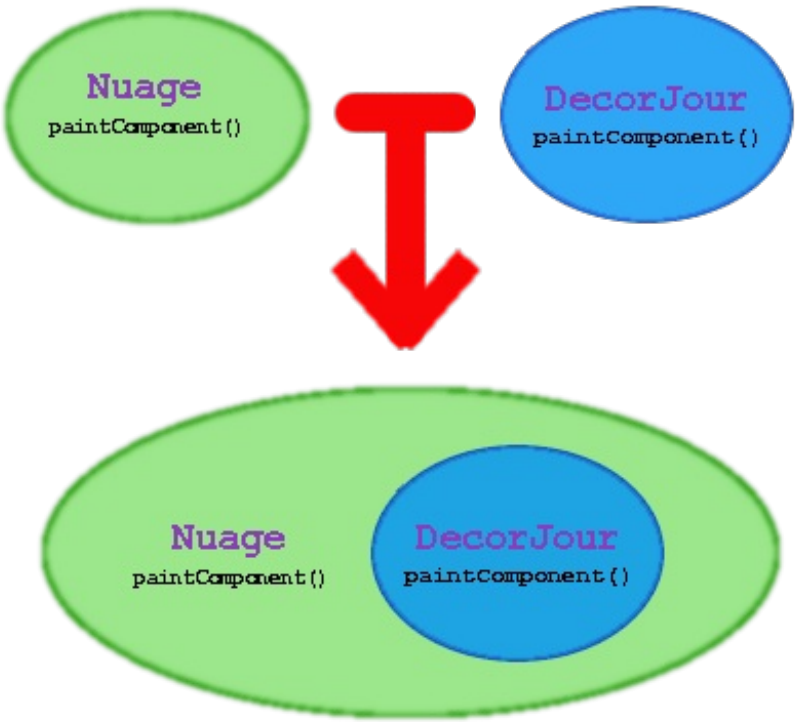
Vous allez voir que tout va devenir limpide.
Nous allons procéder de la façon suivante :

- nous allons créer un objet `DecorJour` ;
- nous allons lui ajouter un nuage ;
- nous allons aussi lui ajouter un arbre ;
- nous appellerons la méthode qui dessine dans notre composant, et celle-ci dessinera le tout !

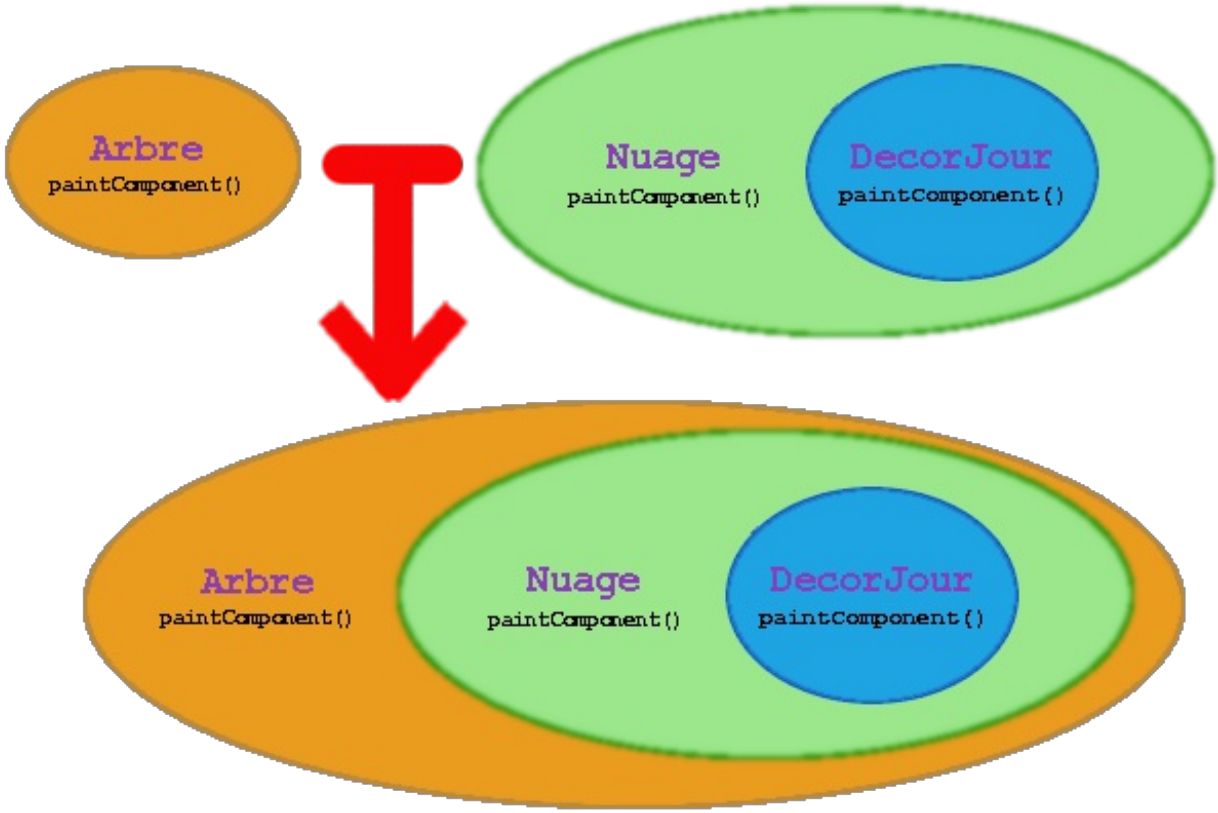
Tout ceci démarre avec un concept fondamental : **l'objet de base et les objets qui le décorent DOIVENT avoir le même type !**
Ceci pour une bonne raison : polymorphisme, polymorphisme et **polymorphisme !**

Ouh là, on ne te suit pas du tout !

Vous allez comprendre. En fait, les objets qui vont décorer notre décor vont avoir la même méthode `paintComponent(Graphics g)` que notre objet principal et nous allons faire fondre cet objet dans les autres !
Ceci signifie que nos objets qui vont servir de décorateur vont avoir une instance de type `Item` en leur sein ! Ceux-ci vont englober les instances les unes après les autres et du coup, nous pourrons appeler la méthode `paintComponent(Graphics g)` de manière récursive !
Vous pouvez voir les décorateurs comme des poupées russes : vous pouvez mettre une poupée dans une autre !
Ce qui signifie que si nous décorons notre décor avec un objet nuage, la situation pourrait être symbolisée comme suit :



Vous pouvez voir que nos deux objets ont une méthode `paintComponent(Graphics g)` et que l'instance de notre décor se trouve maintenant dans notre objet **Nuage**. À ce stade, si nous voulons rajouter un élément de décoration, il nous suffit d'appliquer le même principe. Voici un schéma symbolisant l'ajout d'un arbre décorant le nouvel objet :



L'arbre contient l'instance de la classe **Nuage** qui, elle, contient l'instance de **DecorJour**...

En fait, on va passer notre instance d'objet en objet !

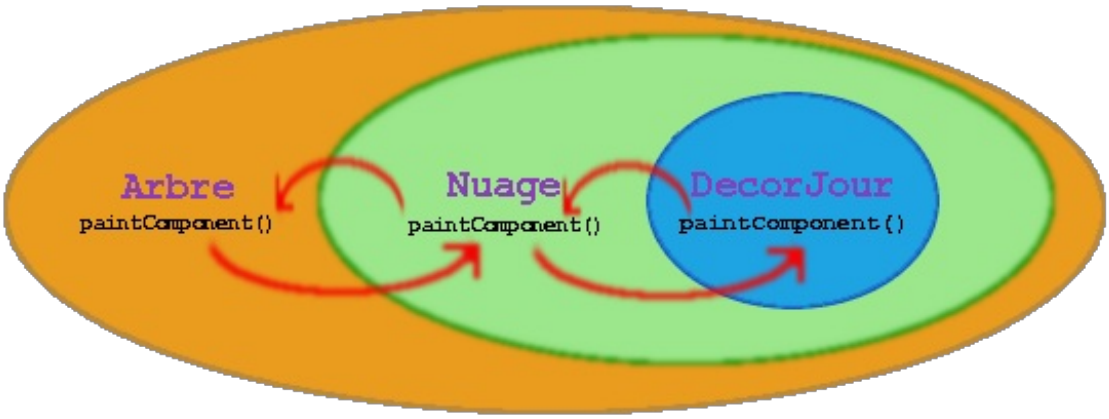
À peu de chose près, c'est ça ! Sauf que seuls les éléments décorant prendront une instance en paramètre...

Et comment tu vas faire pour ajouter les fonctionnalités des objets décorant ?

Tout simplement en appelant la méthode `paintComponent(Graphics g)` de l'instance se trouvant dans l'objet avant de faire les traitements de la même méthode de l'objet courant !
Souvenez-vous lorsque j'expliquais comment fonctionne la pile d'invocation des méthodes dans un thread.

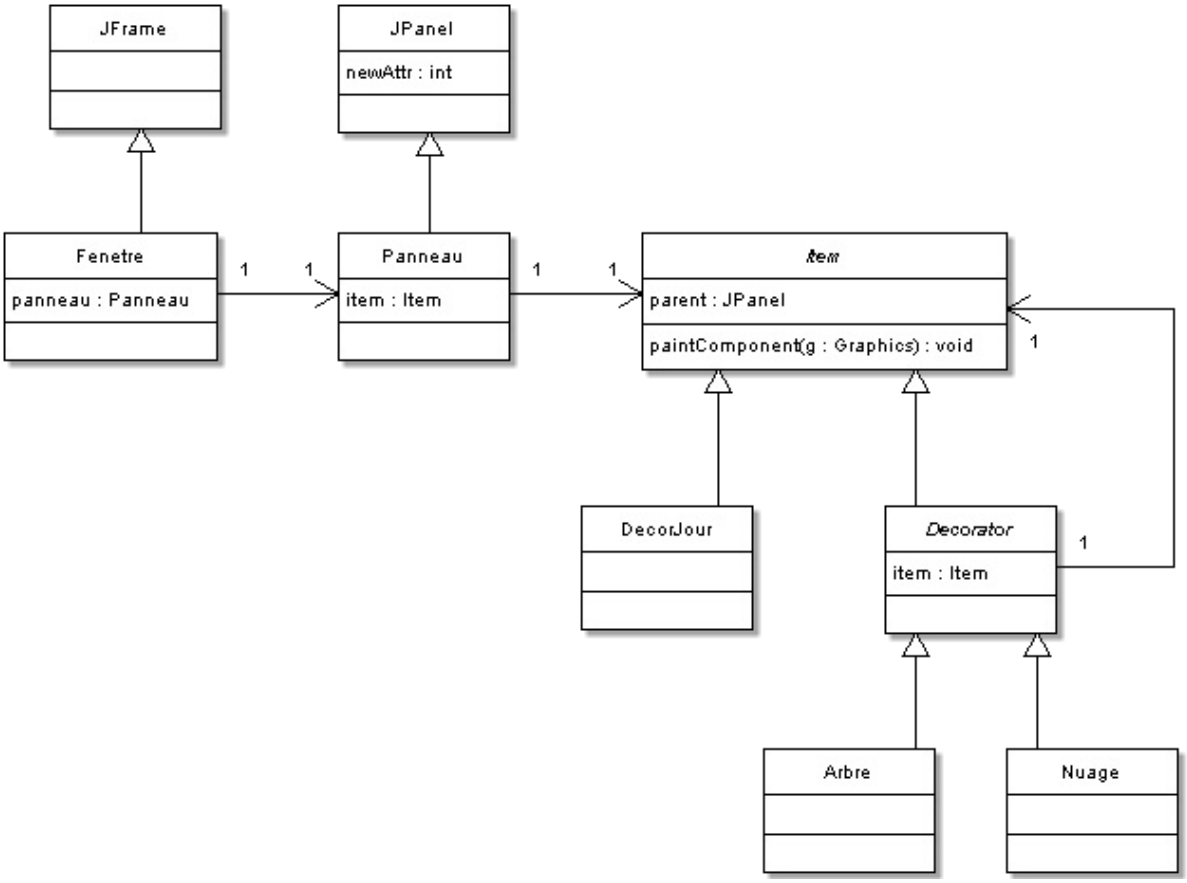
- La méthode de l'objet le plus global, **Arbre**, est appelé en premier.
- Celle-ci appelle la méthode de l'objet de type **Item**, ici, un objet **Nuage**, se trouvant en son sein.
- La méthode du dit objet est à son tour invoquée, mais invoque aussi la méthode de l'objet **Item** qu'il englobe ; nous arrivons à notre objet **DecorJour**.
- Celui-ci va tracer la pelouse, le soleil et le ciel.
- La méthode terminée, les instructions de l'objet **Nuage** sont exécutées, celui-ci trace un nuage.
- Et pour finir, les instructions de l'objet **Arbre** sont exécutées, un arbre apparaît !

Voici un schéma résumant la situation :



Je pense que vous devez y voir un peu plus clair... Mais un exemple concret est toujours plus parlant.

Voici le diagramme de classe de notre programme :



Heu...
Pourquoi tu as mis une classe abstraite entre les objets qui vont nous servir de décorateur et la super-classe **Item** ?

Tout simplement parce nous voulons que seuls les objets décorant aient une instance d'**Item** en leur sein ! Si nous n'avions pas la relation entre **Decorator** et **Item**, nous aurions eu une liaison de notre super-classe vers elle-même... Et du coup, nos objets de décor (**DecorJour** et **DecorNuit**) auraient eu une instance d'eux-mêmes, ce qui aurait signifié qu'un objet **DecorJour** peut en décorer un autre ! Et nous ne voulons pas ça !
Voilà pourquoi il y a une classe abstraite.

Tout comme pour le pattern strategy, l'utilisation d'une classe abstraite permet de définir un super-type d'objet. Nous aurions très bien pu utiliser une interface...

Voici le code source des classes rajoutées dans notre hiérarchie, mais avant de regarder, essayez de créer ces implémentations tous seuls :

Secret (cliquez pour afficher)

Decorator.java

Code : Java

```
package com.sdz.decorator;

import java.awt.Graphics;

import javax.swing.JPanel;

public abstract class Decorator extends Item {
 protected Item item;

 /**
* @param pan
*/
 public Decorator(JPanel pan, Item item) {
  super(pan);
  this.item = item;
 }

 public void paintComponent(Graphics g) {
  //Nous appelons la méthode de la super-classe
  super.paintComponent(g);
  //Enfin nous appelons la méthode de notre instance !
  this.item.paintComponent(g);
 }
}
```

Nuage.java

Code : Java

```
package com.sdz.decorator;

import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JPanel;

public class Nuage extends Decorator {

 /**
* @param pan
* @param item
*/
 public Nuage(JPanel pan, Item item) {
  super(pan, item);
 }

 public void paintComponent(Graphics g) {
  //On invoque la méthode de la classe Decorator
  super.paintComponent(g);

  //On trace notre nuage de taille 30 de haut
  //à un endroit au hasard dans le ciel
```

```
    //Ordonnée à ne pas dépasser
    int y = this.parent.getHeight() - this.parent.getHeight()/3;
    int x = this.parent.getWidth();

    int ordY = y, ordX = x;

    //tant que les coordonnées ne sont pas bonnes
    do{
     ordY = (int)(Math.random() * 100);
     ordX = (int)(Math.random() * 100);
    }while((ordY > y && ordY < 55) && (ordX > x && ordX < x+80));

    //Couleur des nuages : blanc
    g.setColor(Color.white);
    //On dessine des ronds blancs de différentes dimensions
    //presque collés
    g.fillOval(ordX+30, ordY, 20, 20);
    g.fillOval(ordX+42, ordY-8, 28, 28);
    g.fillOval(ordX+60, ordY-14, 34, 34);
    g.fillOval(ordX+80, ordY-8, 28, 28);
    g.fillOval(ordX+100, ordY, 20, 20);
   }
  }
```

Arbre.java

Code : Java

```
package com.sdz.decorator;

import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JPanel;

public class Arbre extends Decorator {

 /**
 * @param pan
 * @param item
 */
 public Arbre(JPanel pan, Item item) {
  super(pan, item);
 }

 public void paintComponent(Graphics g) {
  //On invoque la méthode de la classe Decorator
  super.paintComponent(g);

  //On trace notre nuage de taille 30 de haut
  //à un endroit au hasard dans le ciel

  //Ordonnée à ne pas dépasser
  int y = this.parent.getHeight() - this.parent.getHeight()/3;
  int x = this.parent.getWidth();

  int ordY = y, ordX = x;

  //tant que les coordonnées ne sont pas bonnes
  do{
   do{
    ordY = (int)(Math.random() * 1000);
    ordX = (int)(Math.random() * 100);
   }while(ordY < (this.parent.getHeight() -
```

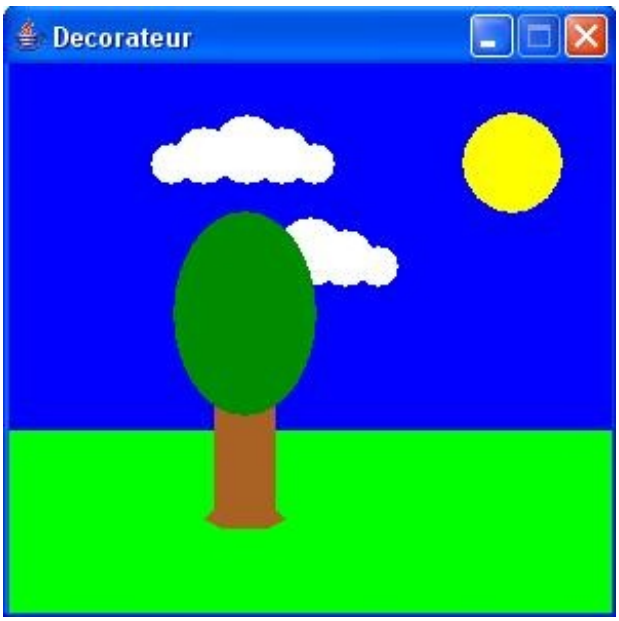
```
        this.parent.getHeight()/3));
          }while(ordY > this.parent.getHeight());

          //System.out.println("ordY : " + ordY);
          //Couleur du tronc : marron
          g.setColor(new Color(169, 97, 36));
          //On crée les coordonnées de nos points
          int[] tabX = {ordX, ordX+10, ordX+30, ordX+40, ordX+35, ordX+35,
ordX+5, ordX+5, ordX};
          int[] tabY = {ordY, ordY+5,  ordY+5,  ordY ,   ordY-5,  ordY-60,
ordY-60, ordY-5, ordY};
          g.fillPolygon(tabX, tabY, 9);

          g.setColor(new Color(0, 140, 0));
          g.fillOval(ordX-15, ordY-150, 70, 100);
         }
        }
```

Panneau.java

Code : Java

```
import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JPanel;

import com.sdz.decorator.*;


public class Panneau extends JPanel {

 private Item decor;

 /**
* @param decor
*/
 public Panneau(Item decor) {
  super();
  this.decor = decor;
 }
 /**
* Par défaut
*/
 public Panneau(){
  this.decor = new DecorJour(this);
  this.decor = new Nuage(this, this.decor);
  this.decor = new Nuage(this, this.decor);
  this.decor = new Arbre(this, this.decor);
 }

 public void paintComponent(Graphics g){
  this.decor.paintComponent(g);
 }
}
```

Vous pouvez voir ce que ce code donne avec le screenshot ci-dessous :



Dans les méthodes de nos objets servant à décorer, vous avez pu voir que je génère ces derniers de façon aléatoire dans une zone spécifique : ce screenshot ne sera peut-être pas ce que vous aurez au final... Relancez le programme pour avoir plusieurs aperçus !

Vous avez remarqué que le seul morceau de code qui a changé se trouvait dans le constructeur de notre objet **Panneau** ! Voici le morceau de code en question :

Code : Java

```
//On crée notre décor de fond
this.decor = new DecorJour(this);
//On ajoute un nuage
this.decor = new Nuage(this, this.decor);
//Un deuxième nuage
this.decor = new Nuage(this, this.decor);
//Et un arbre
this.decor = new Arbre(this, this.decor);
```

Comme je vous l'ai expliqué plus haut, lorsque notre objet va invoquer la méthode `paintComponent(Graphics g)` de son décor, l'invocation va se faire comme mentionné précédemment.
Ici, nous voulons que les éléments se décorent dans un ordre distinct :

- le fond se peint en premier ;
- ensuite, les décorateurs (nuages, arbre).

Vous devez comprendre pourquoi : si nous avions peint les décorateurs en premier et le fond en dernier, vous n'auriez vu que le fond puisqu'il recouvre toute la surface de notre conteneur ! Les décorateurs auraient été effacés, tout simplement.
C'est pour cette raison que, dans la méthode `paintComponent(Graphics g)` de la super-classe **Decorator**, nous avons mis l'invocation de cette même méthode de l'objet **Item** en son sein avant de faire toute chose ! Ainsi, les décorateurs sont peints après le fond et donc recouvrent celui-ci par une nuage, un arbre...

Vous devez savoir que le pattern decorator fonctionne aussi dans l'autre sens, c'est-à-dire que vous pouvez exécuter le code du premier objet en tout premier lieu et invoquer la même méthode de l'objet **Item** ensuite.

Dans notre cas, ça n'a que peu d'intérêt pour la raison que nous venons de voir...

Voilà, je vous félicite d'avoir appris à utiliser votre deuxième pattern de conception !
L'avantage de tels modèles de conception doit vous paraître de plus en plus évidente, maintenant.

C'est sûr que là, on comprend mieux. Il était difficile de croire qu'on puisse ajouter des fonctionnalités à des objets sans modifier le code source de ceux-ci, mais c'est vrai !

En plus, nous avons utilisé un principe de conception, très important, sans que je vous le dise : **il est préférable, dans les limites du possible, de restreindre les possibilités de changement d'un objet.**
Pour faire simple, il faut éviter qu'un objet soit habilité à faire des actions différentes. Par exemple, vous avez vu que, dans le package `java.io` , les classes sont regroupées par fonctionnalité :

- les classes qui lisent les flux ;
- les classes qui écrivent sur les flux.

Ceci car une classe qui aurait pour rôle de faire les deux actions pourrait être amenée à changer si la façon de lire change OU si la façon d'écrire change, OU les deux !
Vous avez pu constater, lors du chapitre précédent, que les changements éventuels peuvent être des ennemis redoutables : inutile, donc, de leur faciliter la tâche !

Au fait, tu nous as dit que ce pattern était utilisé dans des classes Java...

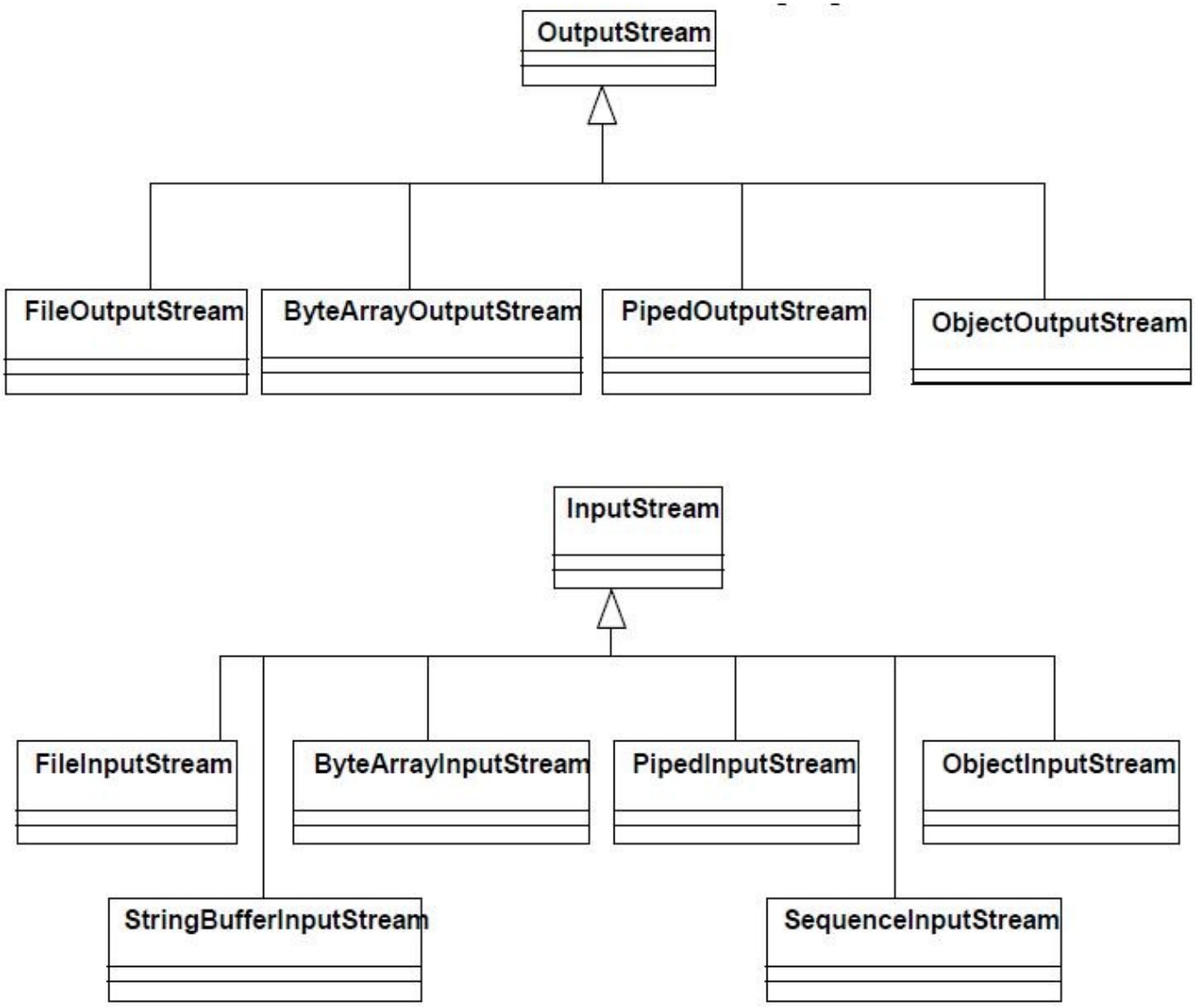
Tout à fait ! Le moment est venu de vous révéler ce grand secret.

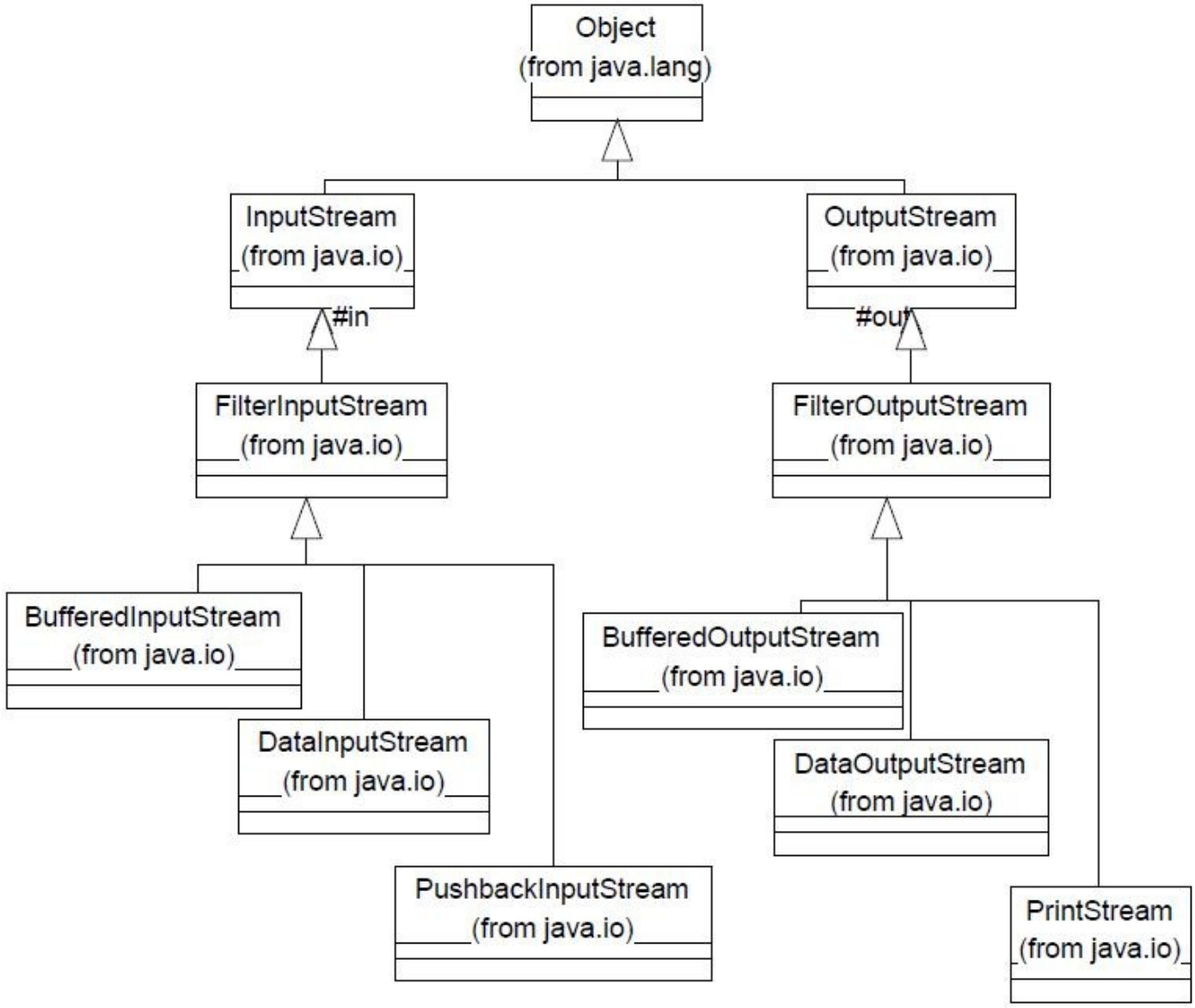
Retour en haut

Les mystères de java.io

Lorsque nous avons vu les flux d'entrée / sortie en Java, vous avez pu constater qu'il y a un nombre colossal de classes dans le package `java.io` .

Voici un schéma donnant un aperçu des ces classes :





Elles ne sont pas toutes présentes, mais ça vous donne un rendu...

Cette quantité de classes s'explique car celles-ci utilisent le pattern decorator !
Les classes présentes dans le deuxième schéma correspondent aux décorateurs des classes se trouvant dans le premier schéma !
Par exemple, vous pouvez avoir ce genre de code :

Code : Java

```
FileInputStream fis = new FileInputStream("toto.txt");
//Ou encore
BufferedInputStream bis = new BufferedInputStream(new
FileInputStream("toto.txt"));
//Ou alors
DataInputStream dis = new DataInputSteam(new BufferedInputStream(new
FileInputStream("toto.txt")));
```

Vous voyez que nous retrouvons la logique du décorateur dans cette façon de faire !
En fait, chaque décorateur ajoute une fonctionnalité à l'objet de base, ce qui rend celui-ci plus performant ou plus intuitif à utiliser...

Vous pouvez même créer vos propres décorateurs pour ce package !

Maintenant que vous connaissez et savez utiliser le pattern decorator, vous devriez avoir une meilleurs approche de ce package.

Bon : ceci dit, il est temps de faire un tour sur le topo...
Retour en haut

Ce qu'il faut retenir

- Le pattern decorator permet d'ajouter des fonctionnalités de façon dynamique à un objet.
- Afin d'utiliser le polymorphisme, les décorateurs et les objets destinés à être décorés doivent dériver d'une même super-classe.
- Les objets décorateurs ont une instance du type de leur super-classe en leur sein.
- Grâce à cette instance, ils peuvent invoquer la méthode commune et ainsi rajouter des traitements à celle-ci.
- La façon d'utiliser un décorateur peut se faire dans deux sens :
  - soit en exécutant les traitements de l'objet du super-type en premier. Les méthodes seront exécutées du premier objet instancié vers le dernier,
  - soit en exécutant le code du décorateur en premier : l'exécution se fait en sens contraire, du dernier objet instancié vers le premier.

Retour en haut

Encore un chapitre riche en nouveautés !
J'espère qu'il vous a plu...

Vous commencez à apercevoir la toute puissance des patterns de conception.
Nous allons donc continuer avec un pattern non moins pratique : **le pattern observer !**

Retour en haut

Soyez à l'écoute de vos objets : le pattern observer

Dans ce chapitre, nous verrons comment faire dialoguer vos objets entre eux.
Vous allez voir que ceci est assez facile, au final, mais l'approche n'est pas évidente au premier abord !

Ce pattern est celui utilisé pour gérer les événements sur vos IHM. C'est par le biais de ce dernier que vos composants peuvent faire des actions lorsque vous cliquez dessus, que vous le survolez...

Je vois que vous êtes impatients de voir ce dernier ! Donc, let's go les zéros !

Posons le problème

Sachant que vous êtes un développeur Java chevronné, un de vos amis proches vous demande si vous pourriez l'aider à faire une horloge digitale en Java.
Celui-ci a la gentillesse de vous fournir les classes à utiliser afin de permettre de faire son horloge.

Votre ami a l'air de s'y connaître car ce qu'il vous a fourni est bien structuré.

Package com.sdz.vue classe Fenetre.java

Code : Java

```
package com.sdz.vue;
import java.awt.BorderLayout;
import java.awt.Font;

import javax.swing.JFrame;
import javax.swing.JLabel;

import com.sdz.model.Horloge;

public class Fenetre extends JFrame{

 private JLabel label = new JLabel();
 private Horloge horloge;

 public Fenetre(){
  /* On initialise notre JFrame */
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setLocationRelativeTo(null);
  this.setResizable(false);
  this.setSize(200, 80);
  /* On initialise l'horloge */
  this.horloge = new Horloge();
  /* On initialise notre JLabel */
  Font police = new Font("DS-digital", Font.TYPE1_FONT, 30);
  this.label.setFont(police);
  this.label.setHorizontalAlignment(JLabel.CENTER);
  /* On ajoute le JLabel à notre JFrame */
  this.getContentPane().add(this.label, BorderLayout.CENTER);
 }


 /* Méthode main pour lancer le programme */
 public static void main(String[] args){
  Fenetre fen = new Fenetre();
  fen.setVisible(true);
 }

}
```

Package com.sdz.model classe Horloge.java

Code : Java

```
package com.sdz.model;

import java.util.Calendar;

public class Horloge extends Thread{
        //Objet calendrier pour récupérer l'heure courante.
 private Calendar cal;
 private String hour = "";
```

```
        public Horloge(){
         Thread t = new Thread(this);
         t.start();
        }

        public void run() {
         while(true){

                                          //On récupère l'instance d'un
    calendrier à chaque tour
                                  //celui-ci va nous permettre de
    récupérer l'heure actuelle
                    this.cal = Calendar.getInstance();
                                          this.hour =  /* Les heures */
             this.cal.get(Calendar.HOUR_OF_DAY) + " : "
             +
             (   /* Les minutes */
              this.cal.get(Calendar.MINUTE) < 10
              ? "0" + this.cal.get(Calendar.MINUTE)
              : this.cal.get(Calendar.MINUTE)
             )
             + " : "
             +
             (   /* Les secondes */
              (this.cal.get(Calendar.SECOND)< 10)
              ? "0"+this.cal.get(Calendar.SECOND)
              : this.cal.get(Calendar.SECOND)
             );
                                 try {
            Thread.sleep(1000);
           } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
           }
          }
         }
        }
```

Si vous ne possédez pas la même police d'écriture que j'ai utilisée, prenez-en une autre : **Arial** ou **Courrier** par exemple...

Le problème auquel il est confronté est simple : **impossible de pouvoir faire communiquer l'horloge avec sa fenêtre...**

Je ne vois pas où est le problème ! Il n'a qu'à faire passer son instance de **JLabel** dans son objet **Horloge** et le tour est joué !

En fait, votre ami, dans son infinie sagesse, souhaite ne pas faire dépendre son horloge de son interface graphique et, je le cite, juste au cas où il devrait passer d'une IHM **swing** à une IHM **AWT**.

C'est vrai que si on passe notre objet d'affichage dans notre horloge, si nous devons changer de type d'IHM, toutes les classes devront être modifiées ! Pas génial...
En fait, lorsque vous faites ceci, on dit que vous **couplez des objets : vous rendez un ou plusieurs objets dépendants d'un ou plusieurs autres objets !**

Entendez par là que vous ne pourrez plus utiliser l'(les) objet(s) couplé(s) indépendamment de l'objet (des objets) auquel il(s) est(sont) attaché(s) !

Le couplage entre objets est l'un des problèmes principaux concernant la ré-utilisabilité d'objets...
Dans notre cas, si vous voulez utiliser votre objet **Horloge** ailleurs, vous serez confrontés à pas mal de problèmes puisque cet objet ne s'affichera que dans un **JLabel** !

Bon, on a compris : le pattern observer va tous nous sauver !

Tout à fait !
Celui-ci va permettre de faire communiquer des objets entre eux sans qu'ils se connaissent réellement !
Vous devez être curieux de voir comment faire... je vous propose donc, sans plus tarder, de vous le présenter.
Retour en haut

Des objets qui parlent et qui écoutent : le pattern observer

Nous allons faire un point sur ce que vous savez de ce pattern pour le moment :

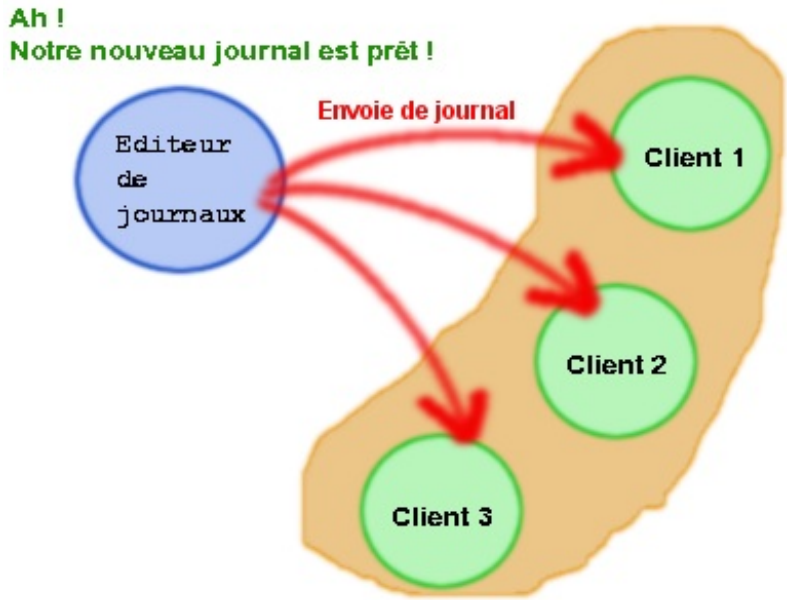
- il permet de faire communiquer des objets entre eux ;
- c'est un bon moyen d'éviter le couplage entre les objets.

Ce sont tout de même deux points cruciaux, mais il y a encore un point qui va vous plaire et que vous ne savez pas encore : **tout se fera automatiquement !**

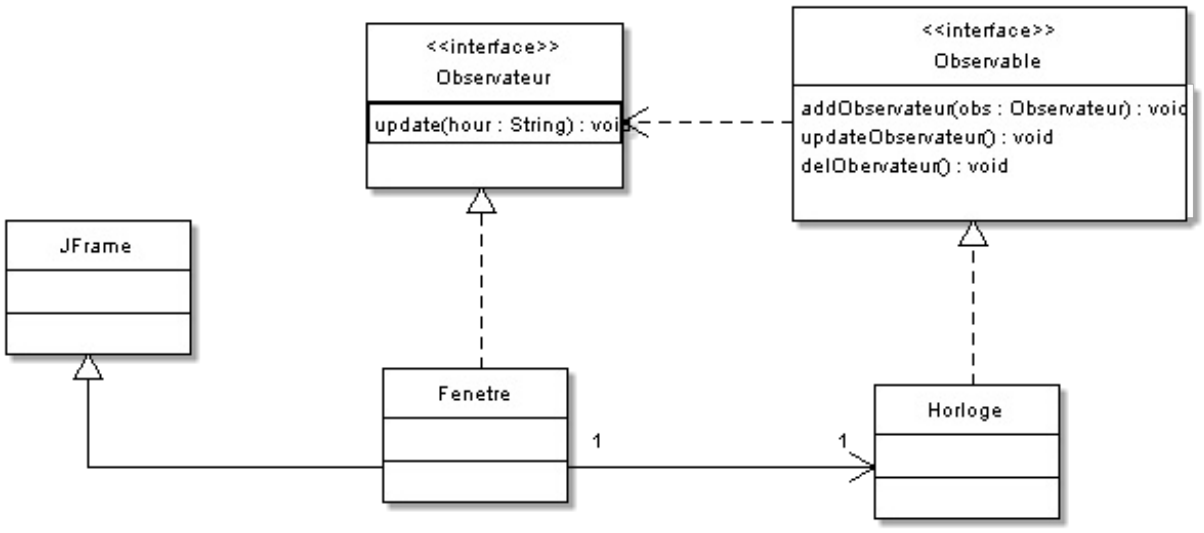
Comment les choses vont se passer ? Faisons un point de ce que nous voulons pour notre horloge digitale : celle-ci doit pouvoir avertir notre objet servant à afficher l'heure lorsque celui-ci doit mettre à jour son affichage.
Ici, vu que les horloges du monde entier se mettent à jour toutes les secondes, la nôtre fera de même...

La chose merveilleuse avec ce pattern, c'est que notre horloge ne se contentera pas de mettre un objet au courant que sa valeur a changé, elle pourra même mettre plusieurs observateurs au courant ! !

En fait, pour schématiser au maximum, voyez la relation entre les objets implémentant le pattern observer comme un éditeur de journal et ses clients :



Avec ce schéma, vous pouvez en conclure que notre objet défini comme observable pourra avoir plusieurs objets qui l'observent. **On dit que cet objet a *une relation de un à plusieurs* vers l'objet `Observateur`.**
Avant de vous expliquer un peu plus le principe de ce pattern, voici le diagramme de classe de notre application :



Avec ce diagramme, vous pouvez vous apercevoir que ce ne sont pas nos instances d'**Horloge** ou de **JLabel** que nous allons passer, mais des implémentations d'interfaces !

En effet, vous savez maintenant que les classes implémentant une interface peuvent être appelées par le type de l'interface. Dans notre cas, notre classe **Fenetre** va implémenter l'interface **Observateur**, celle-ci pourra donc être considérée comme un type **Observateur** !
Vous constaterez aussi que dans la deuxième interface, celle dédiée à l'objet **Horloge**, nous avons trois méthodes :

- une qui permet d'ajouter des observateurs, nous allons donc gérer une collection d'observateurs ;
- une qui permet de retirer les observateurs ;
- et enfin une qui permet de mettre à jour les observateurs !

Grâce à ceci, nos objets ne sont plus liés par leur type respectif, mais par leurs interfaces !
L'interface qui va apporter les méthodes de mise à jour, d'ajout observateur... va travailler avec des objets de type **Observateur** !

Ainsi le couplage ne se fait plus directement, mais il s'opère par le biais de ces interfaces.

Tu nous as dit qu'il fallait éviter le couplage !

Oui, mais dans certains cas il est nécessaire... Ici, il faut que nos deux interfaces soient couplées pour que le système fonctionne. De même que, lors du chapitre précédent, nos classes étaient très fortement couplées puisqu'elles devaient travailler ensemble, nous devions donc faire en sorte de ne pas les séparer.

Voici comment l'application va fonctionner.

- Nous allons avoir une instance de la classe **Horloge** dans notre classe **Fenetre**.
- Cette dernière va implémenter l'interface **Observateur**.
- Notre objet **Horloge**, implémentant l'interface **Observable**, va attendre d'avoir des objets à prévenir de ses changements.
- Nous allons informer notre horloge que notre fenêtre va l'observer.
- À partir de là, notre objet **Horloge** va faire le reste : à chaque changement, nous allons appeler la méthode qui met à jour tous les observateurs !

Voici le code source de ces interfaces (j'ai créé un package com.sdz.observer).

Observateur.java

Code : Java

```
package com.sdz.observer;

public interface Observateur {
 public void update(String hour);
}
```

Observer.java

Code : Java

```
package com.sdz.observer;

public interface Observable {
 public void addObservateur(Observateur obs);
 public void updateObservateur();
```

```
    public void delObservateur();
}
```

Voici maintenant le code de nos deux classes, travaillant main dans la main mais ne se rencontrant JAMAIS.

Horloge.java

Code : Java

```
package com.sdz.model;

import java.util.ArrayList;
import java.util.Calendar;

import com.sdz.observer.Observable;
import com.sdz.observer.Observateur;

public class Horloge extends Thread implements Observable{

 //On récupère l'instance d'un calendrier
 //celui-ci va nous permettre de récupérer l'heure actuelle
 private Calendar cal;
 private String hour = "";
 //Notre collection d'observateurs !
 private ArrayList<Observateur> listObservateur = new
ArrayList<Observateur>();

 public Horloge(){
  Thread t = new Thread(this);
  t.start();
 }

 public void run() {
  while(true){
   this.cal = Calendar.getInstance();
   this.hour =  /* Les heures */
     this.cal.get(Calendar.HOUR_OF_DAY) + " : "
     +
     (   /* Les minutes */
      this.cal.get(Calendar.MINUTE) < 10
      ? "0" + this.cal.get(Calendar.MINUTE)
      : this.cal.get(Calendar.MINUTE)
     )
     + " : "
     +
     (   /* Les secondes */
      (this.cal.get(Calendar.SECOND)< 10)
      ? "0"+this.cal.get(Calendar.SECOND)
      : this.cal.get(Calendar.SECOND)
     );

   //On avertit les observateurs que l'heure a été mise à jour !
   this.updateObservateur();

   try {
    Thread.sleep(1000);
   } catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
   }
  }
 }

 /**
* Ajoute un observateur à la liste
```

```
    */
     public void addObservateur(Observateur obs) {
      this.listObservateur.add(obs);
     }
     /**
    * Retire tous les observateurs de la liste
    */
     public void delObservateur() {
      this.listObservateur = new ArrayList<Observateur>();
     }
     /**
    * Avertit les observateurs que l'observable a changé
    * et invoque la méthode update de chaque observateur !
    */
     public void updateObservateur() {
      for(Observateur obs : this.listObservateur )
       obs.update(this.hour);
     }
    }
```

Fenetre.java

Code : Java

```
package com.sdz.vue;
import java.awt.BorderLayout;
import java.awt.Font;

import javax.swing.JFrame;
import javax.swing.JLabel;

import com.sdz.model.Horloge;
import com.sdz.observer.Observateur;

public class Fenetre extends JFrame {

 private JLabel label = new JLabel();
 private Horloge horloge;

 public Fenetre(){
  /* On initialise notre JFrame */
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setLocationRelativeTo(null);
  this.setResizable(false);
  this.setSize(200, 80);

  /* On initialise l'horloge */
  this.horloge = new Horloge();
  //On place un écouteur sur notre horloge
  this.horloge.addObservateur(new Observateur(){
   public void update(String hour) {
    label.setText(hour);
   }
  });

  /* On initialise notre JLabel */
  Font police = new Font("DS-digital", Font.TYPE1_FONT, 30);
  this.label.setFont(police);
  this.label.setHorizontalAlignment(JLabel.CENTER);
  /* On ajoute le JLabel à notre JFrame */
  this.getContentPane().add(this.label, BorderLayout.CENTER);
 }
```

```
    /* Méthode main pour lancer le programme */
    public static void main(String[] args){
     Fenetre fen = new Fenetre();
     fen.setVisible(true);
    }
}
```

Exécutez ce code et vous verrez que tout fonctionne à merveille !
Vous venez de faire communiquer deux objets entre eux avec un couplage proche de zéro ! Félicitations !

Vous pouvez donc voir de vos yeux que lorsque l'heure change, la méthode `updateObservateur()` est invoquée. Celle-ci parcourt sa collection d'objets **`Observateur`** et appelle la méthode `update(String hour)` de celui-ci. La méthode étant redéfinie dans notre classe **`Fenetre`** afin de mettre à jour le **`JLabel`**, l'heure s'affiche !

La seule chose qui me dérange dans mon exemple tel qu'il est fait, c'est que vous ne voyez pas que l'objet observé met à jour tous les objets qui l'écoutent...
Afin de remédier à cela, nous allons quelque peu modifier notre objet **`Fenetre`** afin que celle-ci n'initialise pas tout de suite l'objet **`Horloge`**, et donc ne lui dise pas tout de suite que celui-ci est observé...

Pour ce faire, nous allons initialiser notre **`Horloge`** avec un mutateur dans la classe **`Fenetre`** et créer une autre classe, **`ZFenetre`**, qui elle, sera codée avec **`AWT`**.

Pour faire très simple, pour passer de **`swing`** à **`AWT`**, il vous suffit d'utiliser les mêmes objets en enlevant le "**J**" du début. Donc, **`JButton`** devient **`Button`**, **`JFrame`** devient **`Frame`**...

Voici les codes sources de ces deux classes.

ZFenetre.java

Code : Java

```
package com.sdz.vue;

import java.awt.BorderLayout;
import java.awt.Font;

import java.awt.*;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.JLabel;

import com.sdz.model.Horloge;
import com.sdz.observer.Observateur;

public class ZFenetre extends Frame{
 private Label label = new Label();
 private Horloge horloge;

 public ZFenetre(){
  this.setLocationRelativeTo(null);
  this.setResizable(false);
  this.setSize(200, 80);

  /* On initialise notre JLabel */
  Font police = new Font("DS-digital", Font.TYPE1_FONT, 30);
  this.label.setFont(police);
  /* On ajoute le JLabel à notre JFrame */
  this.add(this.label, BorderLayout.CENTER);
 }
```

```
 public void setHorloge(Horloge horloge) {
  this.horloge = horloge;
  //On place un écouteur sur notre horloge
  this.horloge.addObservateur(new Observateur(){
   public void update(String hour) {
    label.setText(hour);
   }
  });
 }

}
```

Fenetre.java

Code : Java

```
package com.sdz.vue;
import java.awt.BorderLayout;
import java.awt.Font;

import javax.swing.JFrame;
import javax.swing.JLabel;

import com.sdz.model.Horloge;
import com.sdz.observer.Observateur;

public class Fenetre extends JFrame {

 private JLabel label = new JLabel();
 private Horloge horloge;

 public Fenetre(){
  /* On initialise notre JFrame */
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setLocationRelativeTo(null);
  this.setResizable(false);
  this.setSize(200, 80);

  /* On initialise notre JLabel */
  Font police = new Font("DS-digital", Font.TYPE1_FONT, 30);
  this.label.setFont(police);
  this.label.setHorizontalAlignment(JLabel.CENTER);
  /* On ajoute le JLabel à notre JFrame */
  this.getContentPane().add(this.label, BorderLayout.CENTER);
 }

 public void setHorloge(Horloge horloge) {
  this.horloge = horloge;
  //On place un écouteur sur notre horloge
  this.horloge.addObservateur(new Observateur(){
   public void update(String hour) {
    label.setText(hour);
   }
  });
 }

 /* Méthode main pour lancer le programme */
 public static void main(String[] args){
  //Notre horloge UNIQUE
  Horloge horloge = new Horloge();

  //Notre fenêtre héritant de JFrame
  Fenetre fen = new Fenetre();
  fen.setVisible(true);
  fen.setHorloge(horloge);
```

```
        ZFenetre zFen = new ZFenetre();
        zFen.setVisible(true);
        zFen.setHorloge(horloge);
      }
    }
```

Les deux codes diffèrent légèrement, vous avez dû remarquer que certaines méthodes de notre première fenêtre avaient disparu dans la deuxième...

Ce qui nous donne :

À gauche, la fenêtre faite avec **AWT** et à droite, celle faite avec **swing** !

Voilà, vous venez d'apprendre à utiliser votre troisième pattern de conception.
Je ne vous ai pas caché que celui-ci sert dans la gestion des événements d'interfaces graphiques. Vous avez remarqué que la syntaxe est identique.
Par contre, je vous ai caché quelque chose : il existe des classes Java qui permettent d'utiliser le pattern observer sans que vous ayez à coder les interfaces !
Retour en haut

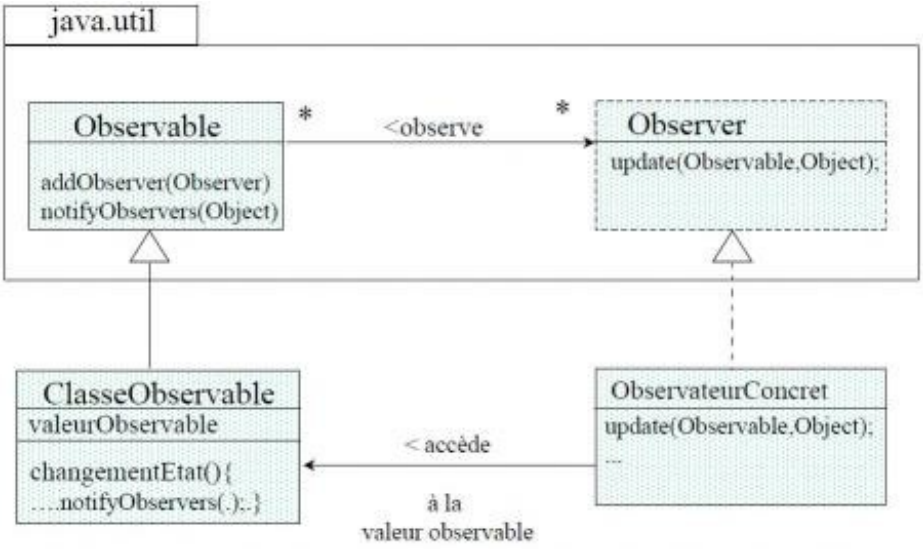
le pattern observer : le retour

En fait, il existe une classe abstraite, **`Observable`** et une interface **`Observer`** dans les classes Java !

Celles-ci fonctionnent de manière quasiment identique à notre façon de faire. Seuls quelques légers détails diffèrent.
Personnellement, je préfère de loin utiliser un pattern observer "*maison*".

Pourquoi ça ?

Tout simplement parce que l'objet que vous souhaitez observer DOIT hériter de la classe **`Observable`**. **Ceci a donc pour conséquence que votre classe ne pourra plus hériter d'une autre classe vu que Java ne gère pas l'héritage multiple !**



Pour les zéros qui ne l'auraient pas compris, ce qui entoure les deux classes sur notre schéma est en fait la symbolique UML d'un package ! Dans ce cas, le package `java.util` .

Vous pouvez voir que celui-ci fonctionne quasiment de la même manière que celui que nous avons développé. Il y a un toutefois un changement dans la méthode `update(Observable obs, Object obj)` : la signature (les paramètres) de la méthode a changé !
Celle-ci prend deux paramètres :

- un objet **`Observable`** ;
- un **`Object`** : donnée supplémentaire que vous souhaitez faire passer !

Vous connaissez le principe de fonctionnement de ce pattern. Il vous est donc facile de comprendre ce schéma. Toutefois, je vous invite à faire vos propres recherches sur l'implémentation de ce pattern dans Java. Vous verrez qu'il y a des subtilités (rien de méchant...).

Je ne vais pas détailler cette implémentation ici, ce n'est pas le but de ce tutoriel.
Je voulais seulement vous montrer un pattern de conception et vous en expliquer le fonctionnement. Vous avez compris, vous savez l'utiliser et le refaire, par conséquent une petite recherche sur les classes **`Observer`** et **`Observable`** ne devrait pas vous poser beaucoup de problèmes !
Personnellement, j'ai été obligé de refaire l'implémentation de ce pattern à chaque fois car mes objets héritaient déjà d'une autre classe...
Retour en haut

Ce qu'il faut retenir

- Le pattern observer permet d'avoir des objets faiblement couplés.
- Ce pattern est basé sur une relation : de un à plusieurs.
- Au lieu de coupler nos objets directement, on préfère coupler les interfaces qu'ils implémentent.
- Grâce à ce pattern, nos objets restent indépendants les uns des autres !

Retour en haut

Chapitre très intéressant, n'est-ce pas ?
Vous pouvez maintenant faire communiquer vos objets entre eux sans problème et façon très optimisée.
Prenez le temps de bien assimiler ce pattern car il pourra sans doute vous sortir de situations complexes...

De plus, ce pattern est la base d'un pattern très connu : **le pattern MVC (modèle, vue, contrôleur)**.

D'ailleurs, il est temps de voir celui-ci !

Retour en haut

Un pattern puissant : le pattern MVC

Voici un chapitre qui risque d'être dur à digérer...
En effet, si le DP MVC est un outil puissant, il n'en est pas moins l'un des patterns les plus compliqués à assimiler et à utiliser.

Ce dernier n'est pas qu'un pattern : il s'agit d'un pattern composé, ce qui signifie qu'il est constitué d'au moins deux patterns (mais ça peut être plus).

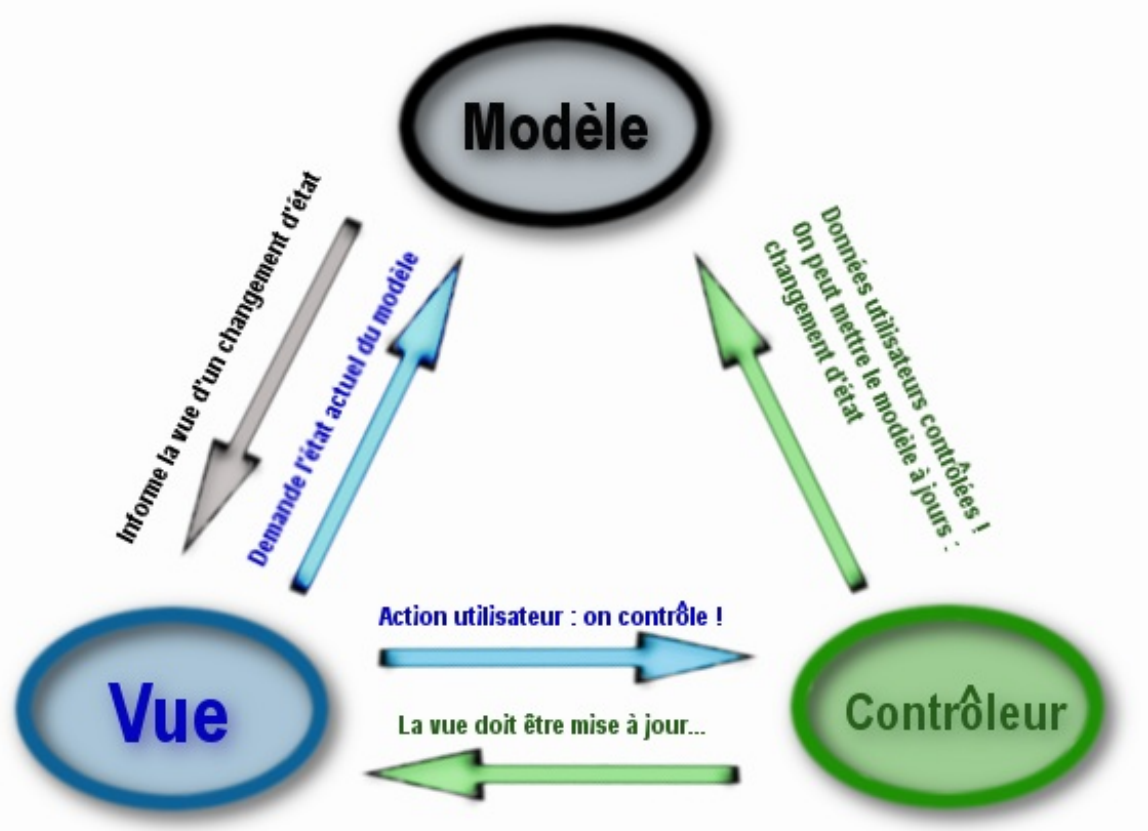
Nous allons voir ceci tout de suite, inutile de tergiverser plus longtemps !

Premiers pas

Dans ce chapitre, nous n'allons pas faire comme d'habitude :

- créer une situation ;
- voir ce que nous pouvons faire ;
- découvrir le pattern.

Je pense que, vu qu'il s'agit d'un pattern compliqué, nous allons directement rentrer dans le vif du sujet... Voici un schéma que ceux qui ont déjà fait des recherches sur ce pattern ont dû déjà voir et revoir :



Avant d'expliquer ce schéma, nous devons faire un point sur ce que sont réellement ces trois entités !

La vue

Ce qu'on nomme la vue est en fait une IHM. Celle-ci est en fait ce qu'a l'utilisateur sous les yeux. La vue peut donc être :

- une application graphique `swing`, `awt`, `swt` pour Java (`Form` pour C#...) ;
- une page web, vous connaissez...
- un terminal Linux ou une console Windows...
- ...

Le modèle

Le modèle peut être divers et varié. il s'agit en général d'un ou plusieurs objets Java (ou bean, nous y reviendrons). Ces objets s'apparentent généralement à ce qu'on appelle souvent "***la couche métier***" de l'application.
Ce sont des objets qui font des traitements absolument transparents pour l'utilisateur. Par exemple, on peut citer des objets dont le rôle est de gérer une ou plusieurs tables d'une base de données.
En trois mots, il s'agit du **cœur du programme !**

Dans le chapitre précédent, nous avons confectionné une petite horloge. Dans cette application, notre fenêtre `swing` correspond à la vue et l'objet `Horloge` correspond au modèle.

Le contrôleur

Cet objet - car il s'agit aussi d'un objet - permet de faire le lien entre la vue et le modèle lorsqu'une interaction utilisateur est survenue sur la vue ! C'est cet objet qui aura pour rôle de contrôler les données et, le cas échéant, ira dire au modèle qu'il peut utiliser les données envoyées par l'utilisateur en toute tranquillité !

Maintenant que toute la lumière est faite sur les trois composants de ce pattern, nous allons expliquer un peu plus la façon de travailler de ce dernier.

Afin d'avoir un exemple concret sur lequel travailler, nous allons reprendre notre calculatrice (cf TP N° 1).

Pour commencer, vous pouvez constater que les trois composants sont tous en interaction les uns avec les autres.
Si nous partons du postulat que votre application, implémentant le pattern MVC, est chargée, voici ce qu'il peut se passer :

- l'utilisateur fait une action sur votre calculatrice (clic sur un bouton) ;
- l'action est captée par le contrôleur qui va vérifier la cohérence des données et, le cas échéant, transformer celles-ci afin d'être comprises par le modèle. Le contrôleur peut aussi demander à la vue de changer...
- le modèle reçoit les données et change d'état (une variable qui change, par exemple...) ;
- le modèle notifie à la vue (ou aux vues) qu'elle (qu'elles) doit (doivent) se mettre à jour ;
- l'affichage dans la vue est modifié en conséquence en allant chercher l'état du modèle.

On y voit déjà plus clair, mais tu nous as dit que MVC est un pattern composé... Il est donc construit avec plusieurs patterns ?

Tout à fait : en fait, avec les chapitres que vous avez vus, vous pouvez isoler deux patterns dans cette architecture.

Je crois voir du pattern observer...

Bien ! Je vois que vous avez bien suivi le chapitre précédent...

Il y a en effet le pattern observer au niveau du modèle. **Comme ceci, lorsque celui-ci va changer d'état, tous les objets qui l'observeront seront mis au courant automatiquement et cela, avec un couplage faible !**

Le deuxième est plus difficile à voir mais il s'agit du pattern strategy !
Ce pattern est situé au niveau du contrôleur. **On dit aussi que le contrôleur est la strategy de la vue !**
En fait, le contrôleur va transférer les données de l'utilisateur au modèle et il a tout à fait le droit de modifier le contenu. Donc, s'il y a des contrôles à faire, la vue aura une implémentation du pattern strategy afin de **centraliser les contrôles et découpler encore plus le modèle de la vue !**

On ne comprend pas trop pourquoi utiliser le pattern strategy alors que le pattern observer aurait suffit !

Souvenez-vous de la raison d'être du pattern strategy : **encapsuler les morceaux de code qui changent !**
Avec l'utilisation de ce pattern, vous prévenez les risques potentiels de changements dans votre logique de contrôle ! Il vous suffira d'utiliser une autre implémentation de votre contrôleur afin d'avoir des contrôles différents.

Ceci dit, vous devez tout de même savoir que le modèle et le contrôleur sont intimement liés : **un objet contrôleur pour notre calculatrice ne servira que pour notre calculatrice ! Nous pouvons donc autoriser un couplage fort entre ces deux objets !**

Vous comprenez le pourquoi du comment maintenant...
Je pense qu'il est temps de se mettre à coder !
Retour en haut

Le modèle

Le modèle sera l'objet qui sera chargé de stocker les données nécessaires à un calcul (nombre et opérateur) et d'avoir le résultat !

Afin de prévoir un changement éventuel de modèle, nous allons créer celui-ci à partir d'un super-type de modèle : comme ça, si un changement s'opère, nous pourrons utiliser les différentes classes filles de façon polymorphe.

Avant de foncer tête baissée, nous allons devoir réfléchir à ce que notre modèle doit savoir faire...
Afin de faire des calculs simples, celui-ci devra pouvoir :

- récupérer et stocker au moins un nombre ;
- stocker l'opérateur de calcul ;
- renvoyer le résultat ;
- calculer le résultat ;
- tout remettre à zéro.

Très bien : voici donc la liste des méthodes que nous trouverons dans notre classe abstraite.
Comme vous le savez, nous allons utiliser le pattern observer afin de faire communiquer notre modèle avec d'autres objets... Il nous faudra donc une implémentation de ce pattern ; le voici, dans un package `com.sdz.observer` .

Observable.java

Code : Java

```
package com.sdz.observer;

public interface Observable {
 public void addObserver(Observer obs);
 public void removeObserver();
 public void notifyObserver(String str);
}
```

Observer.java

Code : Java

```
package com.sdz.observer;

public interface Observer {
 public void update(String str);
}
```

Notre classe abstraite devra donc implémenter ce pattern afin de centraliser les implémentations.
Vu que notre super-type implémente le pattern observer, les classes héritant de cette dernière hériteront aussi des méthodes de ce pattern !

Voici donc le code de notre classe abstraite que nous placerons dans le package `com.sdz.model` .

AbstractModel.java

Code : Java

```
package com.sdz.model;

import java.util.ArrayList;

import com.sdz.observer.Observable;
import com.sdz.observer.Observer;
```

```
public abstract class AbstractModel  implements Observable{

 protected double result = 0;
 protected String operateur = "", operande = "";
 private ArrayList<Observer> listObserver = new
ArrayList<Observer>();
 /**
* Efface
*/
 public abstract void reset();
 /**
* Effectue le calcul
*/
 public abstract void calcul();
 /**
* Affichage forcé du résultat
*/
 public abstract void getResultat();
 /**
* Définit l'opérateur de l'opération
* @param operateur
*/
 public abstract void setOperateur(String operateur);
 /**
* Définit le nombre à utiliser pour l'opération
* @param nbre
*/
 public abstract void setNombre(String nbre) ;


 //*************************************************
 // IMPLÉMENTATION PATTERN OBSERVER
 //*************************************************
 public void addObserver(Observer obs) {
  this.listObserver.add(obs);
 }

 public void notifyObserver(String str) {
  if(str.matches("^0[0-9]+"))
   str = str.substring(1, str.length());

  for(Observer obs : listObserver)
   obs.update(str);
 }

 public void removeObserver() {
  listObserver = new ArrayList<Observer>();
 }


}
```

Ce code est clair et simple à comprendre ; maintenant, nous allons créer une classe concrète héritant de **AbstractModel**.

Voici la classe concrète que j'ai créée.

Calculator.java

Code : Java

```
package com.sdz.model;

import com.sdz.observer.Observable;

public class Calculator extends AbstractModel{
```

```
    //*************************************************
    // MÉTHODES DE CLASSES
    //*************************************************
    /**
   * Définit l'opérateur
   */
    public void setOperateur(String ope){

     //On lance le calcul
     calcul();

     //On stocke l'opérateur
     this.operateur = ope;

     //Si l'opérateur n'est pas =
     if(!ope.equals("=")){
      //On réinitialise l'opérande
      this.operande = "";
     }
    }

    /**
   * Définit le nombre
   * @param result
   */
    public void setNombre(String result){
     //On concatène le nombre
     this.operande += result;
     //On met à jour
     notifyObserver(this.operande);
    }

    /**
   * Force le calcul
   */
    public void getResultat() {
     calcul();
    }

    /**
   * Réinitialise tout
   */
    public void reset(){
     this.result = 0;
     this.operande = "0";
     this.operateur = "";
     //Mis à jour !
     notifyObserver(String.valueOf(this.result));
    }

    /**
   * Calcul
   */
    public void calcul(){
     //S'il n'y a pas d'opérateur, le résultat est le nombre saisi
     if(this.operateur.equals("")){
      this.result = Double.parseDouble(this.operande);
     }
     else{
      //Si l'opérande n'est pas vide, on calcule avec l'opérateur de
   calcul
      if(!this.operande.equals("")){

       if(this.operateur.equals("+")){
        this.result += Double.parseDouble(this.operande);
       }
       if(this.operateur.equals("-")){
        this.result -= Double.parseDouble(this.operande);
       }
```

```
            if(this.operateur.equals("*")){
             this.result *= Double.parseDouble(this.operande);
            }

            if(this.operateur.equals("/")){
             try{
              this.result /= Double.parseDouble(this.operande);
             }catch(ArithmeticException e){
              this.result = 0;
             }
            }
           }
          }
          this.operande = "";
          //On lance aussi la mise à jour !
          notifyObserver(String.valueOf(this.result));
         }
        }
```

Vous ne devriez avoir aucun souci à comprendre ce code, sinon, retournez faire un tour dans la partie 3 du tutoriel !

Voilà, notre modèle est prêt à l'emploi !
Nous allons donc continuer à créer les composants de ce pattern...

Retour en haut

Le contrôleur

Celui-ci sera chargé de faire le lien entre notre vue et notre modèle.
Pour la même raison que pour le modèle : **polymorphisme !** Nous créerons aussi une classe abstraite afin de définir un super-type de variable pour, au cas où, utiliser des contrôleurs de façon polymorphe...

Que doit faire notre contrôleur, ici ?
C'est lui qui va intercepter les actions utilisateur, qui va modeler les données et les envoyer au modèle !
Il devra donc :

- agir lors d'un clic sur un chiffre ;
- agir aussi lors du clic sur un opérateur ;
- avertir le modèle pour qu'il se réinitialise dans le cas d'un clic sur le bouton 'reset' ;
- contrôler les données !

Voilà donc notre liste de méthodes pour cet objet. Cependant, vu que notre contrôleur doit interagir avec le modèle, celui-ci devra avoir une instance de notre modèle !

Voici donc le code source de notre super-classe de contrôle.

AbstractControler.java

Code : Java

```
package com.sdz.controler;

import java.util.ArrayList;

import com.sdz.model.AbstractModel;

public abstract class AbstractControler {

 protected AbstractModel calc;
 protected String operateur = "", nbre = "";
 protected ArrayList<String> listOperateur =  new
ArrayList<String>();

 public AbstractControler(AbstractModel cal){
  this.calc = cal;
  //On définit la liste des opérateurs afin d'être sûr qu'ils
soient bons
  this.listOperateur.add("+");
  this.listOperateur.add("-");
  this.listOperateur.add("*");
  this.listOperateur.add("/");
  this.listOperateur.add("=");
 }

 /**
* Définit l'opérateur
* @param ope
*/
 public void setOperateur(String ope){
  this.operateur = ope;
  control();
 }

 /**
* Définit le nombre
* @param nombre
*/
 public void setNombre(String nombre){
  this.nbre = nombre;
  control();
 }
```

```
    /**
    * Efface
    */
    public void reset(){
     this.calc.reset();
    }

    /**
    * Méthode de contrôle
    */
    abstract void control();
}
```

Nous avons défini les actions globales à notre objet de contrôle et vous constatez aussi qu'à chaque action dans notre contrôleur, celui-ci invoque la méthode `control()` .
Celle-ci va vérifier les données et informer le modèle en conséquence.

Nous allons voir maintenant ce que doit faire notre instance concrète.
Voici donc, sans plus tarder, notre classe.

CalculetteControler.java

Code : Java

```
package com.sdz.controler;

import com.sdz.model.AbstractModel;

public class CalculetteControler extends AbstractControler {

 public CalculetteControler(AbstractModel cal) {
  super(cal);
 }

 void control() {
  //On notifie le modèle d'une action si le contrôle est bon
  //-------------------------------------------------------

  //Si l'opérateur est dans la liste
  if(this.listOperateur.contains(this.operateur)){
   //Si l'opérateur est =
   //On informe le modèle d'afficher le résultat
   if(this.operateur.equals("="))
    this.calc.getResultat();
   //Sinon, on passe l'opérateur au modèle
   else
    this.calc.setOperateur(this.operateur);
  }

  //Si le nombre est conforme
  if(this.nbre.matches("^[0-9.]+$")){
   this.calc.setNombre(this.nbre);
  }
  this.operateur = "";
  this.nbre = "";
 }
}
```

Vous pouvez voir que celle-ci redéfinit la méthode `control()` et qu'elle permet de dire quelles informations envoyer à notre modèle ! Celui-ci mis à jour, les données à afficher dans la vue seront envoyées via l'implémentation du pattern observer entre notre modèle et notre vue.
D'ailleurs, il ne nous manque plus qu'elle.

Donc, allons-y !
Retour en haut

La vue

Voici le plus facile à développer et celui que vous devez maîtriser le mieux..

Nous allons créer celui-ci avec le package `javax.swing` . Vous devriez maintenant avoir l'habitude de créer des IHM, mais si vous ne voulez pas répéter ce que vous avez déjà fait, vous pouvez récupérer mon code d'IHM de calculatrice dans le TP correspondant.
Il va de soi que, vu que les traitements sont tous dans cette classe, nous allons enlever un grand nombre de classes internes !

Je vous donne donc le code source de notre classe que j'ai mis dans le package `com.sdz.vue` .

Calculette.java

Code : Java

```
package com.sdz.vue;


import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingConstants;

import com.sdz.controler.*;
import com.sdz.observer.Observer;


public class Calculette extends JFrame implements Observer{

        private JPanel container = new JPanel();

        String[] tab_string = {"1", "2", "3", "4", "5", "6", "7",
"8", "9", "0", ".", "=", "C", "+", "-", "*", "/"};
        JButton[] tab_button = new JButton[tab_string.length];

        private JLabel ecran = new JLabel();
        private Dimension dim = new Dimension(50, 40);
        private Dimension dim2 = new Dimension(50, 31);
        private double chiffre1;
        private boolean clicOperateur = false, update = false;
        private String operateur = "";

        //Notre instance de notre objet contrôleur
        private AbstractControler controler;

        public Calculette(AbstractControler controler){

                this.setSize(240, 260);
                this.setTitle("Calculette");
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setLocationRelativeTo(null);
                this.setResizable(false);
                initComposant();

                this.controler = controler;

                this.setContentPane(container);
                this.setVisible(true);
        }
```

```
        private void initComposant(){

                Font police = new Font("Arial", Font.BOLD, 20);
                ecran = new JLabel("0");
                ecran.setFont(police);
                ecran.setHorizontalAlignment(JLabel.RIGHT);
                ecran.setPreferredSize(new Dimension(220, 20));

                JPanel operateur = new JPanel();
                operateur.setPreferredSize(new Dimension(55, 225));
                JPanel chiffre = new JPanel();
                chiffre.setPreferredSize(new Dimension(165, 225));
                JPanel panEcran = new JPanel();
                panEcran.setPreferredSize(new Dimension(220, 30));

                //Nous utiliserons le même listener pour tous les
opérateurs
                OperateurListener opeListener = new
OperateurListener();

                for(int i = 0; i < tab_string.length; i++)
                {

                    tab_button[i] = new JButton(tab_string[i]);
                    tab_button[i].setPreferredSize(dim);

                    switch(i){

                     case 11 :
                      tab_button[i].addActionListener(opeListener);
                      chiffre.add(tab_button[i]);
                      break;
                     case 12 :
                      tab_button[i].setForeground(Color.red);
                         tab_button[i].addActionListener(new
ResetListener());
                         tab_button[i].setPreferredSize(dim2);
                         operateur.add(tab_button[i]);
                      break;
                     case 13 :
                     case 14 :
                     case 15 :
                     case 16 :
                      tab_button[i].setForeground(Color.red);

tab_button[i].addActionListener(opeListener);
                         tab_button[i].setPreferredSize(dim2);
                         operateur.add(tab_button[i]);
                      break;
                     default :
                      chiffre.add(tab_button[i]);
                      tab_button[i].addActionListener(new
ChiffreListener());
                      break;
                    }
                }
                panEcran.add(ecran);

panEcran.setBorder(BorderFactory.createLineBorder(Color.black));

                container.add(panEcran, BorderLayout.NORTH);
                container.add(chiffre, BorderLayout.CENTER);
                container.add(operateur, BorderLayout.EAST);
        }



        //**********************************************
        // LES LISTENERS POUR NOS BOUTONS
```

```
            //*********************************************
            class ChiffreListener implements ActionListener{
                    public void actionPerformed(ActionEvent e) {
                        //On affiche le chiffre en plus dans le label
                     String str = ((JButton)e.getSource()).getText();
                        if(!ecran.getText().equals("0"))
                            str = ecran.getText() + str;


        controler.setNombre(((JButton)e.getSource()).getText());
                    }
            }

            class OperateurListener implements ActionListener{
       public void actionPerformed(ActionEvent e) {
        controler.setOperateur(((JButton)e.getSource()).getText());
       }
            }

            class ResetListener implements ActionListener{
                public void actionPerformed(ActionEvent arg0) {
                 controler.reset();
                }
            }



            //*********************************************
            // IMPLEMENTATION DU PATTERN OBSERVER
            //*********************************************
      public void update(String str) {
       ecran.setText(str);
      }
    }
```

Vous devez être à même de comprendre ce code vu qu'il ressemble beaucoup à notre calculette faite dans le TP de la partie 3...

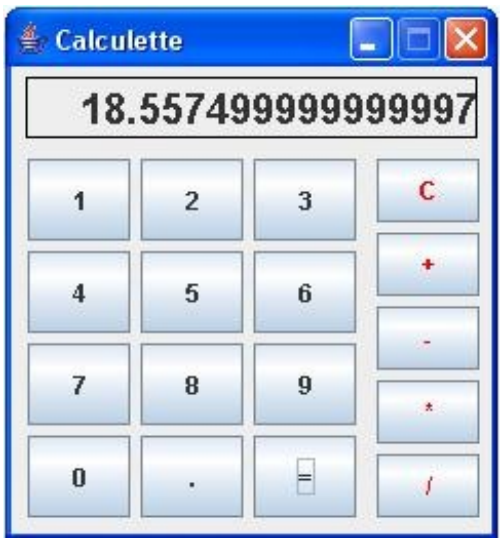
Il y a une nouveauté toutefois, une instance de notre contrôleur ! Celui-ci, je le rappelle, sera la strategy de notre vue...
Notre classe devra donc inclure une instance de notre implémentation du pattern strategy : vous pouvez voir celle-ci juste avant le constructeur de notre classe.

Voilà, toutes nos classes sont opérationnelles.
Il ne nous manque plus qu'une classe de test afin de voir le résultat...
La voici :

Code : Java

```
    import com.sdz.controler.*;
    import com.sdz.model.*;
    import com.sdz.vue.Calculette;

    public class Main {

     public static void main(String[] args) {
      //Instanciation de notre modèle
      AbstractModel calc = new Calculator();
      //création du contrôleur
      AbstractControler controler = new CalculetteControler(calc);
      //Création de notre fenêtre avec le contrôleur en paramètre
      Calculette calculette = new Calculette(controler);
      //Ajout de la fenêtre comme observer de notre modèle
      calc.addObserver(calculette);
     }
    }
```

Testez ce code et le tout fonctionne très bien !
Tous nos objets sont inter-connectés et dialoguent facilement.



Vu que vous connaissez la façon de travailler de ce pattern, nous allons décortiquer ce qu'il se passe.

Lorsque vous cliquez sur un chiffre

- L'action est envoyée au contrôleur.
- Celui-ci vérifie si le chiffre est conforme.
- Il informe le modèle.
- Celui-ci est mis à jour et informe la vue de ses changements.
- Celle-ci rafraîchit son affichage.

Lorsque vous cliquez sur un opérateur

- L'action est toujours envoyée au contrôleur.
- Celui-ci vérifie si l'opérateur envoyé est dans sa liste.
- Le cas échéant, il informe le modèle.
- Celui-ci agit en conséquence et informe la vue de son changement.
- La vue est mise à jour...

La même chose se passe lorsque nous cliquerons sur le bouton "**reset**".

Voilà une bonne chose de faite !

Nous aurions très bien pu faire la même chose sans le contrôleur !

Oui, bien sûr. Même sans modèle !
Rappelez-vous du pourquoi de l'existence des DP : **prévenir des modifications de codes !**

Avec une telle architecture, vous pourrez modifier indépendamment l'un des trois composants de ce pattern.

Si vous ne voulez plus de nombres à virgules à l'affichage mais uniquement des entiers, vous pouvez créer un autre modèle qui ne travaille que sur des entiers !
De même, si vous voulez bloquer certains chiffres ou certains opérateurs, il vous suffit de coder une autre classe héritant de `AbstractControler` qui, elle, ne laissera pas passer tel ou tel choix utilisateur !
Je ne vous donne pas d'exemple pour la vue, nous avons déjà abordé cela dans le chapitre sur le pattern observer.

Oh et puis si, tenez ! C'est encore un des exemples les plus simples qu'on puisse faire...

Au lieu de gérer l'affichage dans notre classe **Calculette**, nous allons créer une classe **Ecran**, toujours dans notre package `com.sdz.vue` , dans laquelle nous allons afficher tout ce que nous renvoie le modèle !

Voici cette classe.

Ecran.java

Code : Java

```
package com.sdz.vue;
import java.awt.*;
import javax.swing.*;
import com.sdz.observer.Observer;

public class Ecran extends JFrame implements Observer{

 private JLabel ecran = new JLabel();

 public Ecran(Component c){
  this.setSize(300, 100);
  this.setLocationRelativeTo(c);
  this.setResizable(false);

  Font police = new Font("DS-digital", Font.BOLD, 50);
  this.ecran.setFont(police);
  this.ecran.setHorizontalAlignment(JLabel.RIGHT);
  this.getContentPane().add(this.ecran, BorderLayout.CENTER);

  this.setVisible(true);
 }

 public void update(String str) {
  ecran.setText(str);
 }

}
```

Et voici notre nouveau code de test :

Code : Java

```
import com.sdz.controler.AbstractControler;
import com.sdz.controler.CalculetteControler;
import com.sdz.model.AbstractModel;
import com.sdz.model.Calculator;
import com.sdz.vue.Calculette;
import com.sdz.vue.Ecran;

public class Main {

 public static void main(String[] args) {

  AbstractModel calc = new Calculator();
  AbstractControler controler = new CalculetteControler(calc);
  Calculette calculette = new Calculette(controler);

  //On crée notre Ecran
  Ecran ecran = new Ecran(calculette);
  //On informe que c'est l'instance d'écran qui écoutera notre
modèle
  calc.addObserver(ecran);
 }
}
```

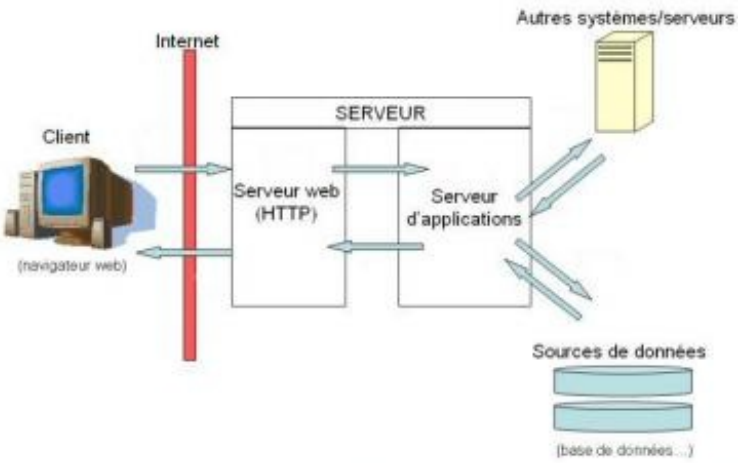
Et le résultat :

Vous pouvez aussi séparer les chiffres des opérateurs... Enfin, faire ce que vous voulez, quoi...

Avant de terminer ce chapitre, vous devez savoir encore quelques choses...
Ce pattern est aussi utilisé lorsque vous développez avec la plateforme JEE et, pour être tout à fait honnête, il en est même le principe de base !
Retour en haut

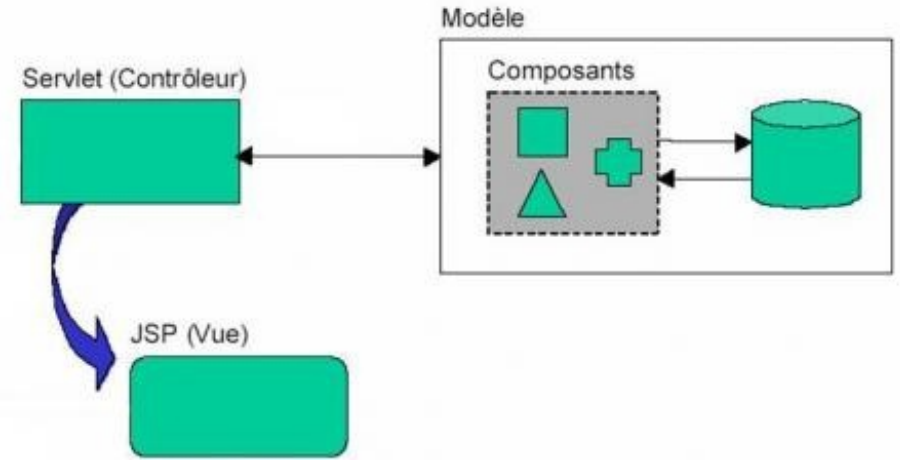
MVC pour le web : le pattern M2VC

Si vous vous rappelez le schéma dont je me suis servi dans le chapitre sur les applets, vous devriez vous souvenir du cycle de vie d'une page web ! Pour mémoire :



Nous reviendrons plus en détail sur le développement web en Java, mais, pour le moment, sachez seulement que vos fichiers seront interprétés et utilisés par **le serveur d'applications**.

Le principe de fonctionnement pour le couple JSP - Servlet est le suivant :



Le schéma est assez explicite.
Vous pouvez voir que les servlets ont le rôle de contrôleurs, les JSP sont les vues et les objets métiers correspondent à votre modèle !
Cependant, un problème pouvait se poser.

En effet, dans certaines boîtes de développement, les tâches sont réparties entre plusieurs développeurs :

- certains seront chargés de développer les objets métiers ;
- d'autres s'occuperont des servlets ;
- et enfin, une troisième équipe sera en charge des vues.

Seulement voilà, les développeurs en charge des vues ne sont peut-être pas des développeurs Java ! Or, avec l'implémentation du pattern MVC vue ci-dessus, il n'était pas rare de trouver des portions de codes Java dans vos pages JSP...

Un peu comme ceci :

Code : Java

```
<html>
  <body>
```

```
        <%
          String[] langage = {"Java", "C++", "C#", "PHP"};
          out.println("<p>Langage de programmtion :");
          out.println("<ul>");
          for(String str : langage)
            out.println("<li>" + str + "</li>");

          out.println("</ul>");
          out.println("</p>");
        %>
      </body>
    </html>
```

Le but de ce chapitre n'étant pas le développement web en Java, je n'expliciterai pas plus ce code ainsi que les suivants. Une partie sera consacrée à cet effet !

Ceci pouvait être dérangeant dans la mesure où un développeur web n'a peut-être pas les connaissances requises pour inclure des portions de code Java dans les pages !
C'est à ce niveau que le pattern M2VC entre en jeu !

Le principe de celui-ci est de pouvoir appliquer le pattern MVC et de séparer les tâches de développement. Ainsi, un développeur web n'a plus qu'à insérer des tags (balises JSP) spécifiés par les développeurs des autres couches applicatives.

En gros, la servlet va communiquer un objet (bean) à la page JSP afin que celle-ci puisse utiliser les informations de ce dernier. Ainsi, le développeur web n'a plus qu'à utiliser une balise propre aux JSP pour afficher le contenu souhaité. Voici un exemple illustrant mes propos :

Code : Java

```
<%jsp:useBean id="compteur" class="Counter" scope="session" />
<html>
  <body>
    <p>
       Affichage de l'attribut "total" de l'objet : <jsp:getProperty
name="compteur" property="total" /> <br />
    </p>
  </body>
</html>
```

Ainsi, les développeurs web n'ont plus à connaître le langage Java, juste à savoir récupérer les informations que leur transmettent les autres couches applicatives !
Le modèle, dans cette architecture, n'avertit plus seulement les vues d'un changement d'état, mais elle leur fournit un ou plusieurs objets pouvant répondre à leurs demandes.

Avant de vous envoyer sur le topo habituel, vous devez savoir encore une petite chose.
Dans le schéma que je vous ai fourni au début de chapitre, le contrôleur passe des informations au modèle. Mais vous pourrez peut-être voir que, parfois, **le contrôleur fait partie des observateurs du modèle...**

Maintenant, nous pouvons voir le topo !

Retour en haut

Ce qu'il faut retenir

- Le pattern MVC est un pattern composé.
- Celui-ci est composé du pattern observer et du pattern strategy.
- L'implémentation du pattern observer permet au modèle de tenir informés ses observateurs.
- L'implémentation du pattern strategy permet à la vue d'avoir des contrôles différents.
- Utiliser ce pattern permet de découpler trois acteurs d'une application, ce qui permet plus de souplesse et de maintenabilité dans la durée de vie de celle-ci.
- Pour la conception d'applications web, on utilise le pattern M2VC.

Toutefois, je mets aussi un bémol à ce pattern. Bien qu'il soit très utile grâce à ses avantages à long terme.
Cependant, vous avez pu constater que **celui-ci complique grandement votre architecture !**
Retour en haut
Un chapitre très riche en informations.

Il faut toutefois que vous sachiez qu'il y a encore un pattern, caché, dans l'implémentation de MVC. En soit, il n'est pas vraiment dans les classes que vous avez programmées, mais plutôt dans le langage Java !

En effet, vous ne le savez sûrement pas, mais vos composants `swing`, comme `AWT`, utilisent un pattern afin de coexister.
Du coup, lorsque vous utilisez une de ces deux bibliothèques, vous utilisez de façon tacite ce pattern.

Je vous propose donc de voir comment fonctionne **le pattern composite**.
Retour en haut

Un véritable assemblage : le pattern composite

Comme je vous l'ai dit à la fin du précédent chapitre, la façon dont Java gère les composants de vos IHM est particulier et c'est le meilleur exemple d'utilisation du pattern composite que je puisse vous donner !

Le but principal de ce pattern est de définir une structure hiérarchique pour un ensemble de données et ceci de façon transparente pour l'utilisateur !
Je me doute bien que ce baratin ne vous parle pas trop... Mais vous allez comprendre !

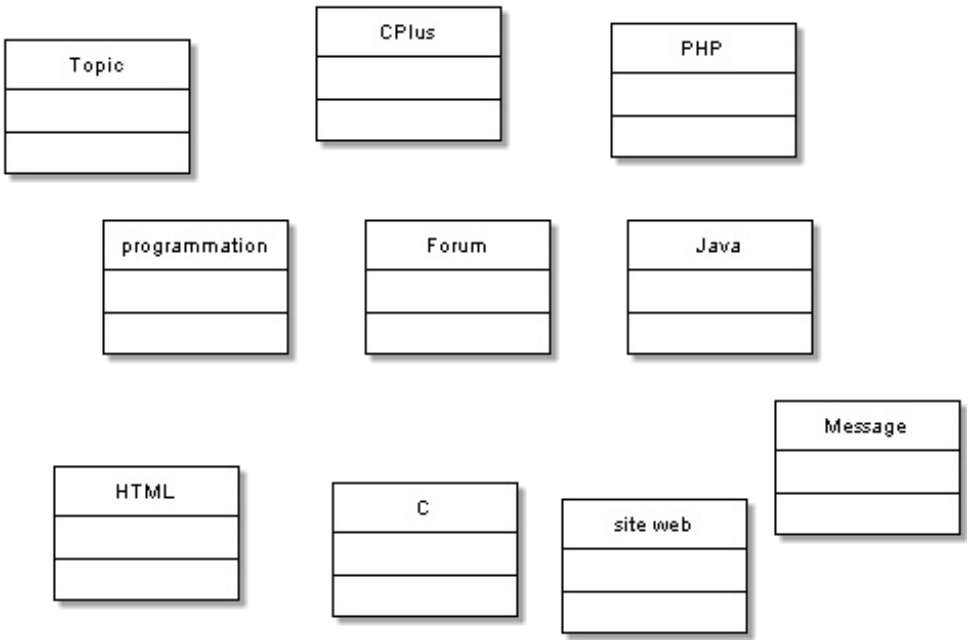
Creusez-vous les méninges

Comme d'habitude, nous allons avoir un petit cas pratique.
Le problème est simple, nous allons voir comment décortiquer la hiérarchie d'un forum avec ce pattern !
Vous avez un forum qui est composé :

- de plusieurs thèmes (au choix) ;
- chaque thème sera composé de plusieurs salons ;
- chaque salon sera composé d'un ou plusieurs topics ;
- et enfin, chaque topic aura un ou plusieurs messages.

Le but du jeu est de réussir à créer un ensemble d'objets interagissant ensemble et que, lorsque vous invoquez la méthode `afficher` de l'objet **`Forum`**, celle-ci affiche tout le contenu de votre forum !
Vous pouvez ajouter des éléments à d'autres avec une méthode `add()` et, si vous voyez d'autres fonctionnalités à implémenter, allez-y.

Je ne vous donne pas de solution, mais voici les classes que je vais utiliser :



La hiérarchie des ces objets est à définir par vous et vous seuls !

Non je rigole, vous pouvez vous y mettre à plusieurs.

Vous pouvez donc faire ce que vous voulez avec ces objets :

- créer un objet mère ;
- faire de l'héritage ;
- tout changer ;
- tout créer.

Seul le nom des classes doit être respecté. De même, il doit y avoir une cohérence entre vos objets... Par là, entendez que les **`Salon`** ne contiendront pas de **`Theme`**... Ceci est un exemple de cohérence mais il y a en a d'autres...

Comment feriez-vous pour gérer ce genre de hiérarchie d'objets ?

Réfléchissez à une hiérarchie de classes qui vous permette de faire tout ça !
Ne vous ruez pas tout de suite vers la solution, essayez par vous-mêmes...

Retour en haut

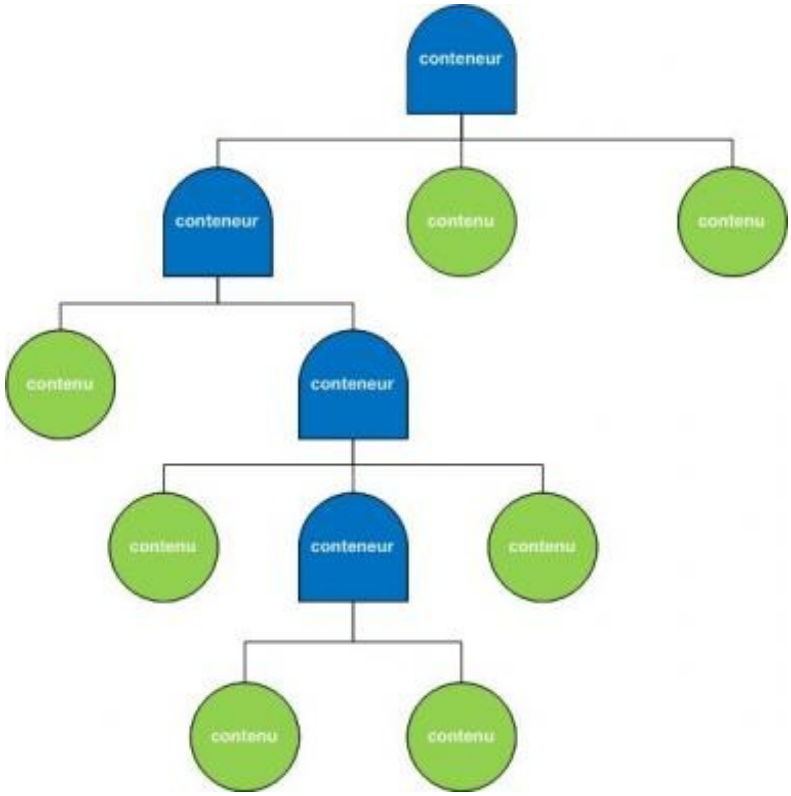
La solution : le pattern composite

Vous pourrez peut-être trouver cette définition du pattern composite :

Citation : Someone

> Le pattern Composite compose des objets en arborescences.
> Ceci permet de représenter une ou plusieurs hiérarchies d'objets sans se soucier de savoir si un objet est composé ou non d'autres objets.

Pour faire court, ce pattern permet de gérer très facilement des données en les structurant en hiérarchie. Voyez ça comme un `JTree` :



En fait, vous pouvez avoir un objet composé d'un ou plusieurs éléments qui peuvent, eux aussi, être composés d'autres éléments, etc.

Afin de simplifier la chose, imaginez une voiture. Cette dernière est composée d'un habitacle, d'un moteur, d'un bas de caisse. Mais ces éléments sont, eux aussi, composés d'autre composants. Ainsi, nous pouvons voir que :

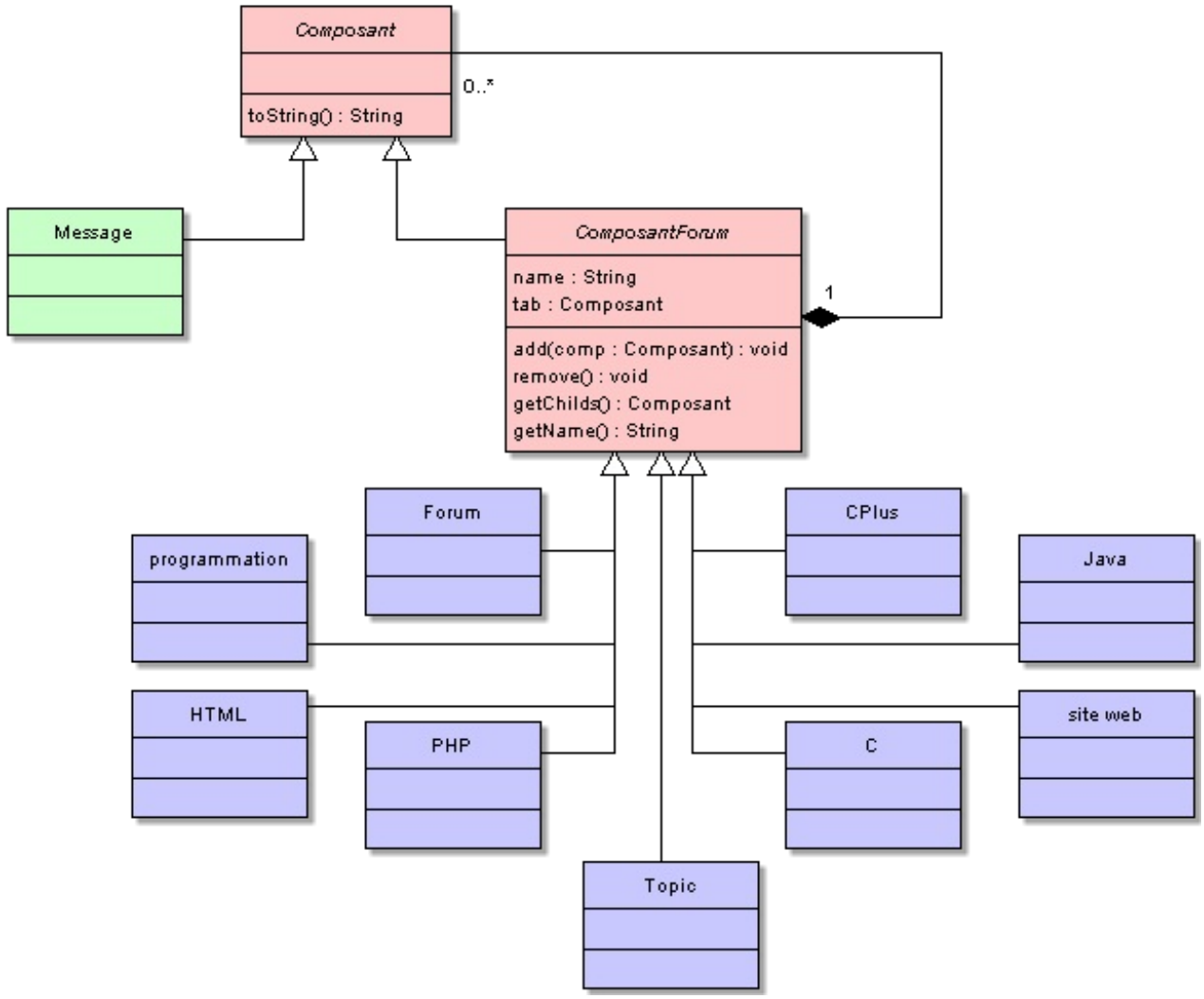
- le moteur est composé d'un radiateur, d'injecteurs, d'un bloc moteur. Mais le bloc moteur est lui aussi composé de telles pièces et de telles autres...
- le radiateur est composé de telles et telles pièces aussi...
- les injecteurs sont des pièces... Ils ne sont pas composés !

On peut aller très loin dans ce type de hiérarchie...

Vous voyez mieux, j'espère !

Si nous en revenons à notre forum, nous pouvez voir que la structure en grappe convient très bien. Il n'y aura pas de branches mixtes, par là entendez : des branches ayant à la fois des messages et des salons (par exemple).

Avant de vous fournir un exemple de code source, voici le diagramme de classe que j'ai fait pour vous :



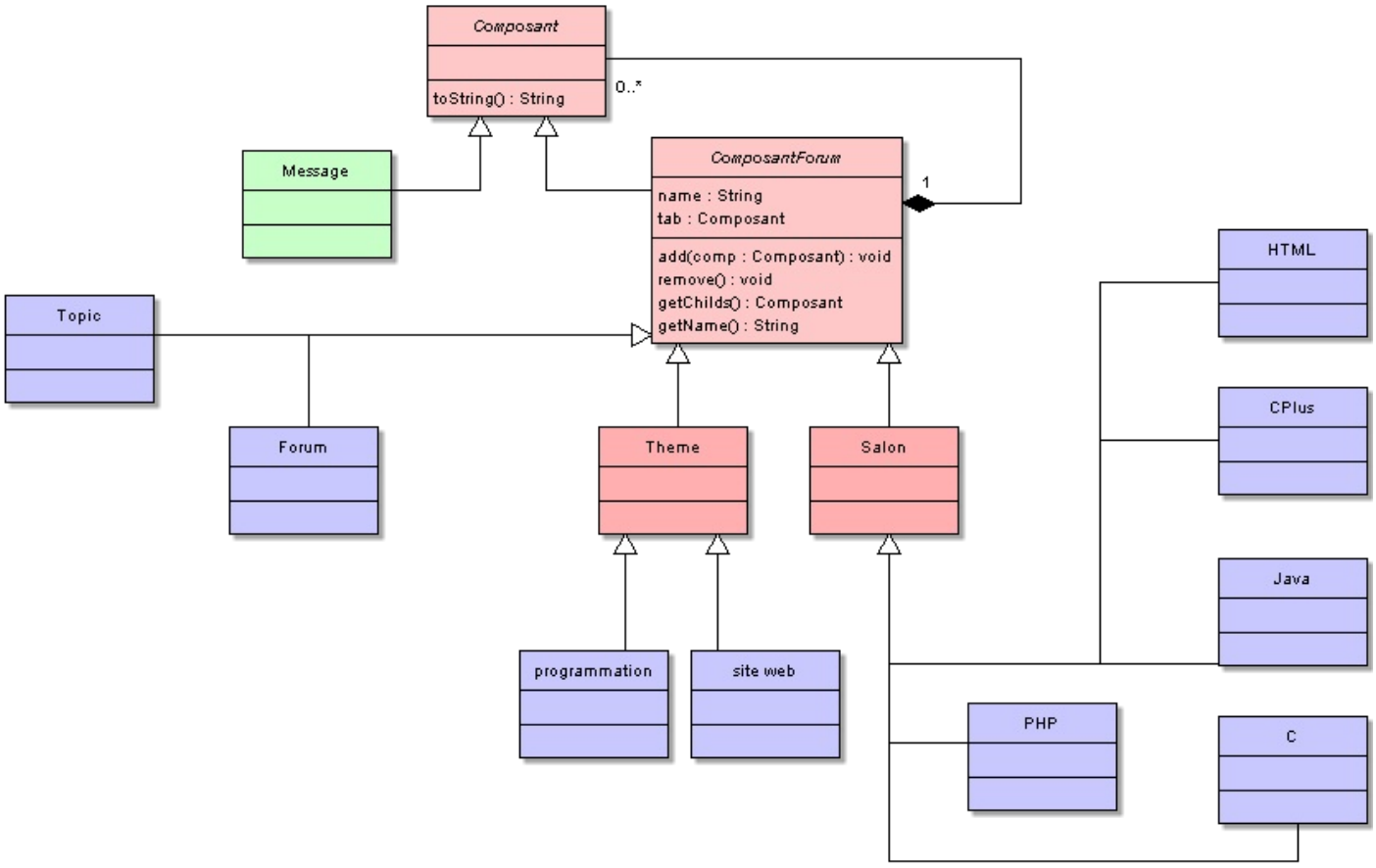
On le trouve bizarre, ton diagramme !

Je sais, les diagrammes de classes peuvent sembler indigestes...
Par contre, vu que vous commencez à savoir les lire, vous devriez voir que :

- vous avez une super-classe dont dérivent toutes les autres et dans laquelle la méthode `toString()` est définie ;
- une classe **`Message`** qui hérite directement de cette super-classe ;
- une autre classe abstraite (eh oui, la première super-classe est aussi abstraite), qui hérite de la super-classe, dans laquelle les méthodes `add()` , `remove()` et `getChilds()` sont définies ;
- et enfin les sous-classes concrètes héritant de la classe mentionnée plus haut !

La sous-classe **`ComposantForum`** a une variable d'instance **`tab`** qui va contenir les composants de chaque objet !
La méthode `toString()` est redéfinie au premier niveau d'héritage.

Voici le diagramme de classe que j'ai réellement utilisé, ceci dû à certaines restrictions sur certains types d'objets :



Pourquoi avoir créé les classes **Theme** et **Salon** ?

Tout simplement car les objets de ces deux types auront des restrictions concernant le genre d'objets dont ils sont constitués... C'est dans ces classes que nous allons pouvoir gérer les composants de nos objets ! Ainsi, nous pourrons seulement ajouter des topics dans un salon, des salons dans un thème, des thèmes dans un forum...
Vous comprenez mieux, j'espère !

Voici les codes source des classes que j'ai créées (j'ai essayé de les regrouper du haut de la hiérarchie vers le bas).

Classe Composant.java

Code : Java

```
package com.sdz.composite;

public abstract class Composant {
	//Cette variable sert seulement pour mon exemple
	protected String tabulation = "";
	public abstract String toString();
}
```

Classe Message.java

Code : Java

```
package com.sdz.composite;

public class Message extends Composant{
```

```
public class Message extends Composant{

        private String message;

        public Message(String message){
                this.message = (message.trim().equals("") || message
== null) ? "" : message;
                this.tabulation = "\t\t";
        }

        public String toString() {
                return this.tabulation + this.message;
        }
}
```

Classe ComposantForum.java

Code : Java

```
package com.sdz.composite;

import java.util.ArrayList;

abstract class ComposantForum extends Composant{
        protected ArrayList tab = new ArrayList();
        protected String name = "";


        public ComposantForum(){}
        public ComposantForum(String nom){this.name = nom;}

        public void add(Composant comp){
                this.tab.add(comp);
        }
        public void remove(int i){
                this.tab.remove(i);
        }
        public ArrayList getChilds(){
                return this.tab;
        }
        public String toString(){
                String str = "";
                str += this.tabulation +"+ " + name + "\n";
                for(Object comp : this.tab){
                        str += this.tabulation +
((Composant)comp).toString() + "\n";
                }
                return  str;
        }
}
```

Classes Forum.java et Topic.java

Code : Java

```
package com.sdz.composite;

import java.util.ArrayList;
public class Forum extends ComposantForum{

        public Forum(){
```

```
                this.name = "Forum";
        }
        public Forum(String nom) {
                super(nom);
        }

        public void add(Composant comp){
                if((comp instanceof Theme))
                        this.tab.add(comp);
        }
}
```

Code : Java

```
package com.sdz.composite;

public class Topic extends ComposantForum {

        public Topic(String string) {
                super(string);
                this.tabulation += "\t\t";
        }

        public void add(Composant comp){
                if((comp instanceof Message))
                        this.tab.add(comp);
        }
}
```

Classe Salon.java et Theme.java

Code : Java

```
package com.sdz.composite;

public class Salon extends ComposantForum{
        public Salon(String nom) {
                super(nom);
                this.tabulation += "\t";
        }

        public void add(Composant comp){
                if((comp instanceof Topic))
                        this.tab.add(comp);
        }

}
```

Code : Java

```
package com.sdz.composite;

public class Theme extends ComposantForum{

        public Theme(String nom) {
```

```
                super(nom);
                this.tabulation += "\t";
        }

        public void add(Composant comp){
                if((comp instanceof Salon))
                        this.tab.add(comp);
        }
}
```

Classes Programmation.java et SiteWeb.java

Code : Java

```
package com.sdz.composite;

public class Programmation extends Theme{

 public Programmation(String nom) {
  super(nom);
 }
}
```

Code : Java

```
package com.sdz.composite;

public class SiteWeb extends Theme {

 public SiteWeb(String nom) {
  super(nom);
 }
}
```

Classes CPlus.java, Java.java, Html.java, PHP.java...

Code : Java

```
package com.sdz.composite;

public class Cplus extends Salon{

        public Cplus(String nom) {
                super(nom);
        }

}
```

Code : Java

```
package com.sdz.composite;

public class Html extends Salon {
```

```
        public Html(String nom) {
                super(nom);
        }
}
```

Code : Java

```
package com.sdz.composite;

public class Java extends Salon {
        public Java(String nom) {
                super(nom);;
        }
}
```

Code : Java

```
package com.sdz.composite;

public class Php extends Salon {

        public Php(String nom) {
                super(nom);
        }
}
```

Code : Java

```
package com.sdz.composite;

public class C extends Salon {

        public C(String nom) {
                super(nom);
        }
}
```

Pfiou !... Il y en a, des classes...
Même si vous aviez bien compris comment les objets allaient se comporter entre eux, vous devez mieux comprendre avec du code sous les yeux !
Maintenant, vous comprenez comment on gère l'interdiction de certains composants dans d'autres...

Dans notre cas, nous n'avons rien fait de spécial !
Mais rien ne vous empêche de faire une levée d'exception si l'utilisateur essaie d'ajouter un type d'objet prohibé !

Vu que nous avons toutes nos classes, il est grand-temps de voir comment celles-ci se comportent...
Voici un code qui permet de tester notre hiérarchie :

Code : Java

```
//J'ai déclaré mes classes dans un autre package, d'où l'import
import com.sdz.composite.*;
public class Test {
```

```
	public static void main(String[] args) {
		//Création de l'objet Forum
		Forum fofo = new Forum();

		//Création d'un thème
		Theme prog = new Programmation("Thème ->
Programmation <-");
		//Ajoute du thème dans le forum
		fofo.add(prog);
		//Création d'un Salon
		Salon java = new Java("Salon : Java");
		//ce salon ne s'ajoutera pas : C'EST INTERDIT
		fofo.add(java);
		//Le salon n'aura pas non plus le thème dans la
collection
		java.add(prog);
		//Par contre le thème aura le salon, et vu que le
forum à le thème...
		prog.add(java);

		//On crée deux topic
		Topic topic = new Topic("Je vous passe le bonjour
!");
		Topic topic2 = new Topic("Problème de pattern !");
		//On les ajoute dans le salon
		java.add(topic);
		java.add(topic2);

		//On ajoute les message dans les topics
		topic.add(new Message("Coucou de cysboy !"));
		topic.add(new Message("Coucou du SDZ ! ! !"));
		topic.add(new Message("Vous suivez toujours..."));

		topic2.add(new Message("Je ne comprends pas bien
le pattern composite !"));
		topic2.add(new Message("Même avec cet exemple tu
ne comprends pas ?"));
		topic2.add(new Message("C'est vrai que là... On
comprend beaucoup mieux"));

		//On crée un deuxième salon
		Salon c = new Cplus("Salon : C++");
		//On l'ajoute dans programmation
		prog.add(c);
		//On crée aussi un topic et on l'ajoute dans le
salon
		Topic topic3 = new Topic("Problème de
pointeur...");
		c.add(topic3);
		//On lui ajoute des messages
		topic3.add(new Message("Je pige rien aux
pointeurs... HELP !"));
		topic3.add(new Message("Va faire un tour sur le
tuto de M@teo..."));
		topic3.add(new Message("Ah oui !.. :p"));

		//Le forum a donc :
		// - un thème
		// - deux salons
		// * Java : 2 topics
		// * C++ : 1 topic
		System.out.println(fofo.toString());

	}
}
```

Voilà ce que me donne ce code :

```
+ Forum
        + Thème -> Programmation <-
                + Salon : Java
                        + Je vous passe le bonjour !
                                Coucou de cysboy !
                                Coucou du SDZ ! ! !
                                Vous suivez toujours...

                        + Problème de pattern !
                                Je ne comprends pas bien le pattern composite !
                                Même avec cet exemple tu ne comprends pas ?
                                C'est vrai que là... On comprend beaucoup mieux


                + Salon : C++
                        + Problème de pointeur...
                                Je pige rien aux pointeurs... HELP !
                                Va faire un tour sur le tuto de M@teo...
                                Ah oui !.. :p
```

Simple, *isn't it* ?
Vous avez une belle hiérarchie d'objets qui s'entremêlent : un objet **`Forum`**, auquel vous ajoutez des thèmes ; à ces thèmes, vous ajoutez des salons qui, eux, ont des topics composés de messages.
Et lorsque vous demandez au forum de s'afficher, tout est automatique ! Même si vous passez des objets non autorisés, cela n'empêche pas le code de fonctionner !

Afin de mieux comprendre ce pattern, voici à quoi il ressemble dans Java.

Retour en haut

Le composite et Java

Je vous l'ai dit au début de ce chapitre et à la fin du précédent : le pattern composite est utilisé pour gérer le contenu des fenêtres en Java.

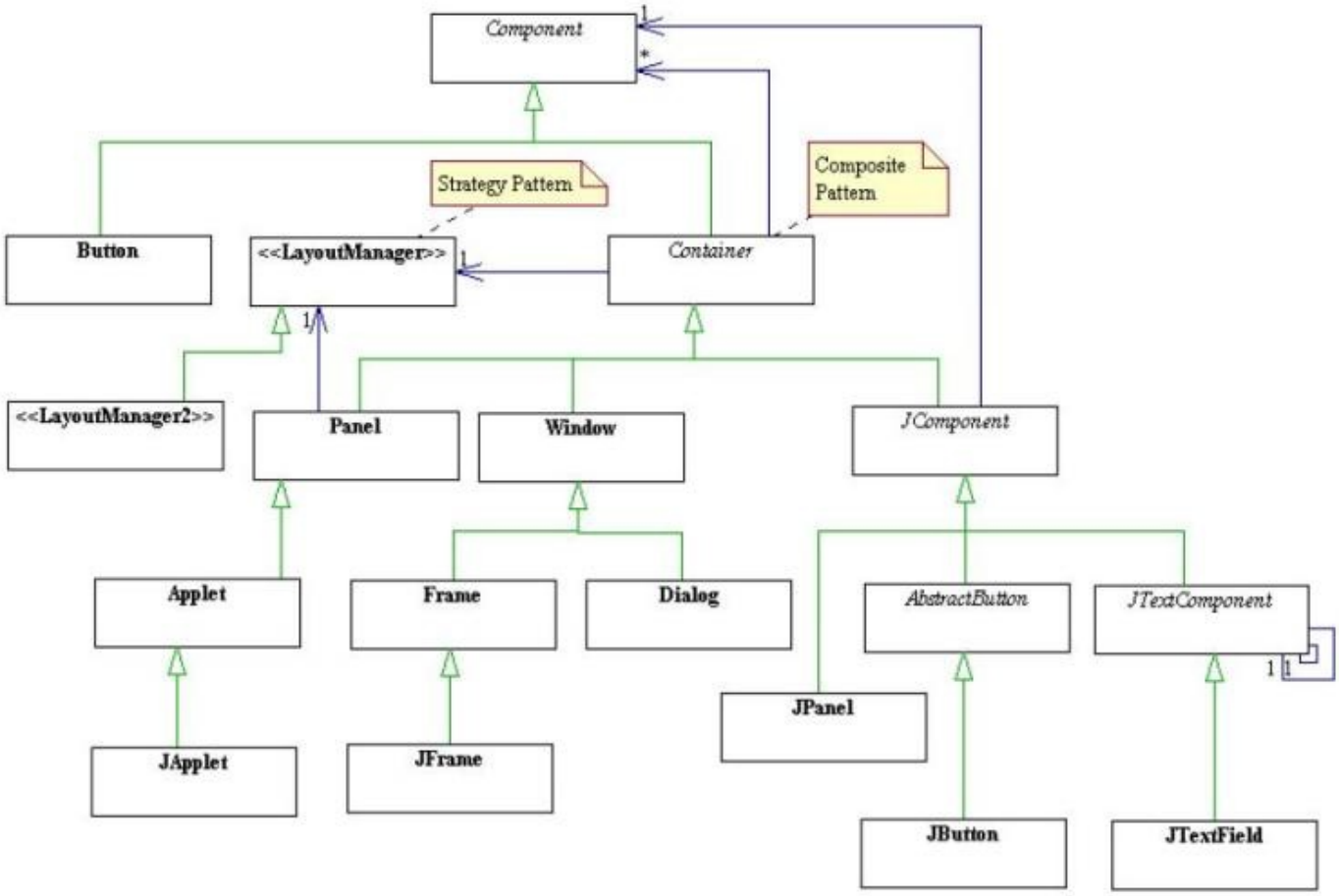
Les objets graphiques que vous utilisez sont assemblés grâce à ce pattern, vous pouvez ainsi mettre des **`JPanel`** dans des **`JPanel`**, qui ont des **`JButton`**, le tout dans un **`JFrame`** !

Vous avez remarqué que ces composants avaient aussi une méthode `add(Component comp)` qui permet d'ajouter un composant !
En fait, vous pouvez voir que s'il y a ce genre de méthode dans le langage, il y a fort à parier que le composite ne soit pas loin...

Tout du moins pour ce qui est des composants graphiques !

Les collections n'implémentent pas vraiment ce pattern, donc pas d'amalgame !

Pour les plus curieux d'entre vous, voici à quoi ressemble le diagramme de classes des composants Java :



Maintenant que le décryptage de diagramme de classes n'a plus de secret pour vous, vous pouvez voir clairement le pattern composite au niveau des objets **`Container`** et **`JComponent`**.

Ceux-ci acceptent donc des objets de type **`Component`** afin de les composer.
De ce fait, on peut donc ajouter un **`JPanel`** dans une **`JFrame`**, etc.

J'espère que c'est assez clair pour vous !

Très clair ! Par contre, on trouve que les patterns composite et decorator se ressemblent beaucoup !

En effet, mais ne vous y trompez pas !

Le pattern decorator implémente une relation de un à un entre ses acteurs, tandis que le pattern composite implémente une

relation de un à plusieurs !
Pour faire simple, pour décorer un objet A, on l'englobe dans un autre objet B.
Pour composer un objet A, on stocke un ou plusieurs objets B dans l'objet A.

De plus, leur rôle est totalement différent :

- **le composite permet de créer des objets composés d'autres objets ;**
- **le decorator permet de rajouter des fonctionnalités à un objet.**

Les diagrammes de classes se ressemblent beaucoup ! Seule la flèche de composition diffère.
Oui, la flèche avec le losange rempli s'appelle une flèche de "***composition***".
Vous voyez que le nom de ce pattern peut être tiré de là... (Bon, là, c'est du folklore informatique...).

Je crois qu'il est grand temps de faire un tour du côté de notre topo.

Retour en haut

Ce qu'il faut retenir

- Le pattern composite permet de faciliter la composition d'objets.
- Ce pattern met en oeuvre une relation d'un à plusieurs envers ses différents acteurs.
- Le composite est utilisé dans le langage Java pour gérer les composants des fenêtres graphiques.
- Ce dernier ressemble au pattern decorator mais la relation de un à plusieurs est absente.
- Le but du composite est de pouvoir construire des objets avec d'autres, tandis que celui du decorator est de rajouter des fonctionnalités à un objet.

Retour en haut

J'espère que vous comprenez mieux comment fonctionne vos IHM, maintenant !

Normalement, vous devriez avoir de bonnes idées pour créer des collections faciles à utiliser...
C'est vrai que ce pattern est souvent utilisé avec des collections : normal, il y a une relation de un à plusieurs !

Retour en haut

J'espère sincèrement que cette partie vous a permis d'y voir plus clair dans les méandres des DP !
Vous conviendrez que ceux-ci sont très utiles et vous permettent de faire des applications plus faciles à maintenir, modifier, faire évoluer...

Retour en haut

Partie 5 : Annexes

Voici une annexe afin de vous fournir un complément sur certains des chapitres vus jusqu'ici !

Retour en haut

Annexe A : liste des mots clés

Voici ici la liste des mots clés du langage Java.
Merci à **Drumer67bts** pour cet annexe ainsi que pour la conclusion !

Les mots clés

Cette annexe est une présentation des mots clés du langage Java et de leur signification. Ces mots clés ne peuvent être utilisés que pour quoi ils ont été créés. Ainsi, vous ne pouvez pas les utiliser comme noms de variables, noms de méthodes ou noms de classes.

À tout moment, cette liste de mots clés peut vous être utile. Si vous ne comprenez pas la signification de l'un d'eux, c'est que vous n'avez sans doute pas lu le chapitre dans lequel il se trouvait.

La signification de la liste de mots clés présentée dans cette annexe sera très courte pour chaque mot car il ne s'agit pas d'un chapitre de cours, mais plutôt de la présentation rapide de ceux-ci listés de façon à pouvoir retrouver rapidement celui que vous recherchez. Ainsi, vous n'aurez pas à rechercher dans tout le tutoriel la signification d'un de ces mots.

Mots clés	Signification
`abstract`	Désigne une classe ou une méthode abstraite
`assert`	Localise les erreurs internes du programme (à utiliser de préférence en phase de test)
`boolean`	Type booléen (valeurs possibles : *true* ou *false*)
`break`	Sort de la boucle en cours ou d'une instruction `switch`
`byte`	Type entier (8 bits) : -128 à 127
`case`	Clause d'une instruction *switch*
`catch`	Clause d'un bloc *try/catch* interceptant un type d'exception
`char`	Type caractère Unicode (16 bits)
`class`	Utilisé pour la définition d'une classe
`continue`	Passe à l'itération suivante d'une boucle sans effectuer les instructions situées après ce mot clé de l'itération en cours
`default`	Cas par défaut d'une instruction *switch*
`do`	Point d'entrée d'une boucle *do/while* (*RÉPÉTER / JUSQU'À* en algorithmique)
`double`	Type nombre flottant en double précision (64 bits)
`else`	Clause d'une instruction *if*
`enum`	Énumère les valeurs possibles pour le type énuméré
`extends`	Indique la classe mère d'une classe

`false`	Valeur booléenne contraire à *true*
`final`	• Constante : type de variable dont le contenu ne peut pas changer • Méthode : elle ne peut pas être surchargée ou redéfinie dans une sous-classe • Classe : elle ne peut pas être dérivée
`finally`	Clause d'un bloc *try/catch* exécutée, qu'il y ait eu déclenchement d'une exception ou non
`float`	Type nombre flottant en simple précision (32 bits)
`for`	Boucle dont le nombre de répétitions est connu (correspond à la boucle *POUR / FIN POUR* en algorithmique)
`if`	Instruction conditionnelle (correspond au *SI / FIN SI* en algorithmique)
`implements`	Indique la ou les classes implémentée(s) par une classe
`import`	Importe un package
`instanceof`	Permet de tester si un objet est une instance d'une classe
`int`	Type entier (32 bits) : -2 147 483 648 à 2 147 483 647
`interface`	Comparable à une classe mais de manière plus abstraite, une interface contient des méthodes devant être redéfinies **obligatoirement** par la classe implémentant l'interface
`long`	Type entier (64 bits) : -9 223 372 036 854 775 808 à 9 223 372 036 854 775 807
`native`	Indique que le code qui suit est écrit dans un autre langage que Java (solution non portable)
`new`	Instancie un nouvel objet ou un tableau
`null`	Référence nulle : un objet ayant pour valeur *null* ne pointe vers aucun objet, il n'a aucune valeur
`package`	Ensemble de classes
`private`	• Classe : uniquement utilisable pour les classes internes • Méthode : on ne peut y accéder qu'à partir de la classe dans laquelle elle est définie • Attribut : on ne peut y accéder qu'à partir de la classe dans lequel il est défini
`protected`	• Classe : uniquement utilisable pour les classes internes • Méthode : on ne peut y accéder qu'à partir de la classe dans laquelle elle est définie, par les classes filles ou le package dans lequel se trouve la classe contenant la méthode • Attribut : on ne peut y accéder qu'à partir de la classe dans lequel il est défini, par les classes filles ou le package dans lequel se trouve la classe contenant l'attribut

public	Donne l'accessibilité par toutes les classes
return	Utilisé dans une méthode pour retourner une valeur ou un objet
short	Type entier (16 bits) : -32768 à 32767
static	• Attribut : instancié une seule fois, l'attribut est commun à l'ensemble des instances d'une classe • Méthode : peut être appelée directement à partir du nom de la classe sans utiliser une instance de la classe
super	Accède à la super-classe ou à son constructeur
switch	Alternative à l'instruction *if* (les différents cas doivent être connus)
synchronized	Assure l'accès à une méthode ou à un bloc de code à un seul thread (les autres threads doivent alors attendre s'ils veulent accéder à cette partie du code)
this	Utilisé dans une classe pour indiquer l'objet courant
throw	Lance une exception
throws	Utilisé pour la déclaration des exceptions que peut lancer une méthode
transient	Déclare une variable non sérialisable
true	Valeur booléenne contraire à *false*
try	Bloc de code susceptible d'intercepter une exception
void	Utilisé pour les méthodes qui ne renvoient rien
volatile	Permet de s'assurer que plusieurs **Thread** concurrent puissent accéder à la dernière valeur de la variable. Donc, une variable déclarée **volatile** est accessible par plusieurs **Thread** simultanément !
while	Boucle dans laquelle il est possible de ne jamais entrer (correspond à *TANT QUE / FIN TANT QUE* en algorithmique)

Retour en haut

Comme vous avez pu le constater, la liste des mots clés est assez importante. Vous êtes sûrement de mon avis : une telle liste était nécessaire pour vous y retrouver.

Retour en haut

Annexe B : Les objets travaillant avec des flux

Voici une liste non exhaustive des objets java travaillant avec les flux I/O.

Les objets traitant des flux d'entrée (in)

Sous-classes de `InputStream`

Sous-classe	Fonction
`FileInputStream`	Permet de créer un flux avec un fichier présent dans le système de fichiers. Cette classe possède un constructeur prenant en paramètre un objet de type `File` ou un `String` , qui représente le chemin vers le fichier.
`ByteArrayInputStream`	Permet de lire des données binaires à partir d'un tableau d'octets.
`PipedInputStream`	Permet de créer une sorte de tube d'entrée (pipe). Dans celui-ci, les informations circuleront sous forme d'octets. Cette classe possède un constructeur ayant pour paramètre un objet de type `PipedOutputStream`. On peut ainsi connecter les deux tubes ; en gros, ce qui est écrit dans une extrémité peut être lu depuis l'autre.
`BufferedInputStream`	Cette classe permet la lecture de données à l'aide d'un tampon, un buffer si vous préférez. À l'instanciation, un tableau d'octets est créé afin de servir de tampon et permet de ne pas surcharger la mémoire. Ce tableau est redimensionné automatiquement à chaque lecture pour contenir les données provenant du flux d'entrée. Ce type d'objet est particulièrement approprié lors de traitement de fichiers volumineux !
`DataInputStream`	Cet objet sert à lire des données représentant des types primitifs de Java (`int`, `boolean`, `double`, `byte`, ...) préalablement écrits par un `DataOutputStream`. Grâce à cet objet, vous pouvez récupérer des éléments sérialisés avec des méthodes comme `readInt()`, `readDouble()`...
`PushbackInputStream`	Lit un flux binaire en entrée et replace le dernier octet lu dans le flux !
`LineNumberInputStream`	Permet d'avoir les numéros de lignes lues dans le flux en plus de lire le flux lui-même.
`SequenceInputStream`	Permet de concaténer deux (ou plus) flux d'entrée, ce qui permet de traiter plusieurs flux d'entrée comme un seul et unique flux !
`ObjectInputStream`	Permet de «désérialiser» un objet, c'est-à-dire de restaurer un objet préalablement sauvegardé à l'aide d'un `ObjectOutputStream`. Cet objet est l'homologue de l'objet `DataInputStream`, à la différence que celui-ci traite des objets.

Sous-classes de `Reader`

Sous-classe	Fonction
`CharArrayReader`	Offre la possibilité de lire un flux de caractères en mémoire.
`StringReader`	Idem que la classe précédente.
`FileReader`	Permet de lire un fichier avec un flux de caractères.
`InputStreamReader`	Convertit un flux binaire en flux de caractères : elle convertit un objet de type `InputStream` en objet de type `Reader`.
`PipedReader`	Idem que leurs cousines héritées de `InputStream`, mais pour des flux de caractères.
`PushbackReader`	Au lieu de remettre un octet dans le flux comme sa cousine (`PushbackInputStream`), celle-ci remet un caractère dans le flux d'entrée.

Retour en haut

Les objets traitant les flux de sortie (out)

Sous-classes de `OutputStream`

Sous-classe	Fonction
`FileOutputStream`	Toutes ces classes ont les mêmes fonctions que les classes héritant de `InputStream`, mais cette fois pour l'écriture.
`ByteArrayOutputStream`	
`PipedOutputStream`	
`BufferedOutputStream`	
`DataOutputStream`	
`PushbackOutputStream`	
`LineNumberOutputStream`	
`SequenceOutputStream`	
`ObjectOutputStream`	

Sous-classes de `Writer`

Sous-classe	Fonction
`CharArrayWriter`	Toutes ces classes ont les mêmes fonctions que les classes héritant de `Reader`, mais cette fois pour l'écriture.
`StringWriter`	
`FileWriter`	
`InputStreamWriter`	
`PipedWriter`	
`PushbackWriter`	

Retour en haut
Retour en haut

Annexe C : Eclipse

Voici un petit pense-bête sur Eclipse !
Vous retrouverez toutes les astuces vues tout au long du tuto ainsi que quelques petits trucs en plus...

Installation

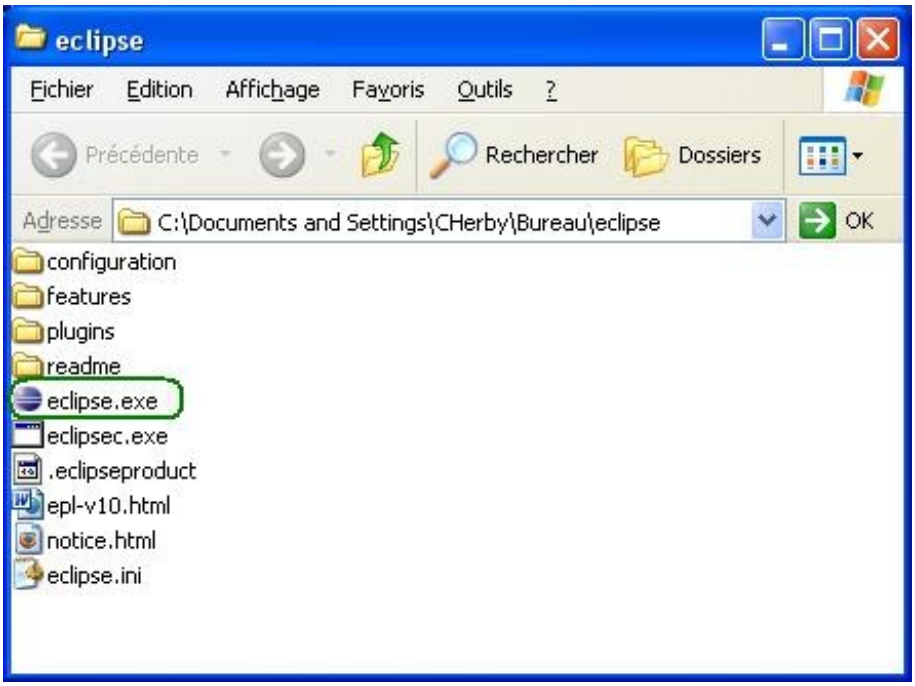
Pour installer Eclipse, vous devez vous souvenir que vous devez AVANT TOUT avoir un environnement Java !
Si vous n'en avez pas, allez faire un tour à cette adresse et téléchargez la dernière version du JRE.
Installez-le, vous êtes maintenant prêts à télécharger et à installer Eclipse...

Installation Windows

Pour le téléchargement, rendez-vous sur ce site, et choisissez le lien "**Eclipse IDE for Java Developers**".

Une fois téléchargé, vous pouvez constater qu'il s'agit d'une archive au format **.zip**. Il vous suffit de l'extraire à l'endroit souhaité (si vous n'avez pas de logiciel d'extraction d'archives, je vous conseille 7Zip).

Vous devriez avoir un dossier contenant ces fichiers :



En vert, vous pouvez voir l'exécutable pour lancer Eclipse !
Et voilà, c'est fini.

Installation Linux

Pour installer Eclipse sous Linux, c'est très simple : vous pouvez aller à la même adresse (site officiel d'Eclipse) et choisissez la version Linux. Il vous suffit ensuite, tout comme pour Windows, d'extraire l'archive au format **.tar.gz** et de double cliquer sur l'icône "**Eclipse**" :

Pour les inconditionnels du terminal, vous pouvez faire une recherche :

Code : Bash

```
apt-cache search eclipse
```

Ensuite, vous repérez la version qu'il vous faut et vous terminez par :

Code : Bash

```
sudo apt-get install <LeNomDeVotreEclipse>
```

Retour en haut

Raccourcis utiles

Voici une liste des raccourcis clavier fort utiles !

- **ALT + SHIFT + N** : Nouveau projet
- **CTRL + S** : enregistrer la ***classe*** * Java où on est positionnés
- **CTRL + SHIFT + S** : tout sauvegarder
- **CTRL + W** : fermer la classe Java où on est positionnés
- **CTRL + SHIFT + W** : fermer toutes les classes Java ouvertes
- **CTRL + C** : copier la sélection
- **CTRL + X** : couper la sélection
- **CTRL + V** : coller la sélection
- **CTRL + A** : tout sélectionner
- **CTRL + F** : chercher / remplacer

Vous avez déjà vu ces raccourcis au tout début du tutoriel !
Mais il y a en a d'autres qui pourront peut-être vous servir :

- **CTRL + ESPACE** : sûrement l'un des raccourcis les plus pratiques et les plus utilisés ! Avec ce dernier, vous n'avez qu'à taper le début d'un nom de méthode et, après avoir fait cette combinaison de touche, Eclipse vous propose une liste de méthodes correspondant au début de votre saisie ;
- **CTRL + SHIFT + O** : *organize Imports*, soit, en français, organisation des imports. Donc, permet de générer les imports dont vous avez besoin pour votre classe ;
- **CTRL + SHIFT + R** : vous permet de rechercher un fichier spécifié dans votre *workspace* ;
- **CTRL + SHIFT + F** : met en forme tout votre code si vous n'avez pas sélectionné de zone spécifique, sinon, met en forme la zone sélectionnée ;
- **CTRL + SHIFT + C** : commente / dé-commente les lignes de code sélectionnées ;
- **CTRL + SHIFT + P** : permet de se déplacer d'une accolade à l'autre (`if`, `while`, `for`...), très pratique pour se retrouver dans un code complexe...
- **ALT + SHIFT + S** : revient à faire "clic-droit / Source" sur votre projet ou votre classe. Permet donc de générer les *getters* et *setters*, des méthodes spécifique (`equals`...) ;
- **SHIFT + ALT + Z** : ouvre le menu "Surround with", ou "entourer par" en français. Ce qui signifie que vous pourrez entourer une portion de code pré-sélectionnée avec ce que vous propose ce raccourci.

Si vous voulez une liste plus complète, vous pouvez aller faire un tour sur ce lien. Il y a une grande quantité de raccourcis !
Retour en haut
Voilà une bonne chose de faite.

J'ose espérer que cette annexe pourra vous servir. En tout cas, elle me sert déjà...
Retour en haut

En espérant que celle-ci vous a été utile.
Ce tutoriel sur Java se termine !
J'ose espérer qu'il vous a été agréable et que avez appris à domestiquer la bête...

Eh ! Tu nous avais promis un tuto sur JDBC ! !

Et je vous réponds qu'un autre tuto portant sur des API Java existe, avec au programme :

- JDBC ;
- programmation réseau ;
- JMF ;
- ...

Et en bonus, vous trouverez même un tuto sur Java EE, qui permet de développer des applications Web. Alors, elle est pas belle

la vie ?

Et oui, le tuto actuel commençait à être un peu gros et rajouter tout ça dedans aurait fini par vous faire peur...

Donc je vous donne rendez-vous dans un autre tuto !
@ bientôt amis ZérOs...
Retour en haut
Partager

Retour en haut

Retour en haut

333 commentaires pour "Apprenez à programmer en Java"
Note moyenne : 3.58 / 4 (982 votes)

Pseudo	Commentaire
Hydro9 Avis : Très bon Ville : Morne-a-l'eau Pays : Guadeloupe Études : Lignes & Formations	# Posté le 19/01/2012 à 02:37:39 Ce tutoriel est très intéressant surtout quand tu veux apprendre le JavaFx Xhtml/css:|||||||| 100% HTML 5/css3 :||| 30 % Javascript:||| 30%
sitws	# Posté le 20/01/2012 à 21:07:57 bonsoir, j'aimerais bien avoir des totoriels de ce genre mais un peu avancé(reseaux, les threads,le web,....) merci d'avance
coopun77	# Posté le 21/01/2012 à 12:32:09 Bonjour, je voulais savoir si il fallait maîtriser toutes les notions aborder pour coder des jeux videos ou à partir de quel moment je peux commencer à m'exercer? Merci beaucoup
satiron Avis : Mitigé Ville : Toulouse Pays : France métropolitaine Études : IUT A Toulouse	# Posté le 22/01/2012 à 17:15:26 Je suis assez mitigé. Les explications sont bonnes, on comprend facilement, de ce cotés là, le cotés pédagogique est là, j'ai même le livre chez moi. J'ai lu ton tuto il y a un certain temps, donc désolé si il y a eu des corrections depuis, ou si je me trompe. Par contre, je suis désolé, mais le code est crade! Je trouve que pour un tuto/livre qui va expliquer ce langage à des centaines de personnes, qui n'ont peut être jamais programmés, leur montrer un code comme ça, c'est pas tip top. Tu devrais regarder du cotés de PMD qui analyse la propreté du code. Quelques exemples quand même: - Ne jamais initialiser des attributs directement, on a un constructeur, il sert à ça!! (private int maVar = 42 par exemple, c'est nul) - Les noms de variables jp1, jp2 et jp3, ça craint - Une règle que l'on m'a toujours appris aussi, un attribut une ligne! Notre esprit vas lire de haut en bas, il risque de ne pas voir une variable si elle est sur la même ligne que l'autre... C'est d'autant plus important quand en plus tu initialise directement ta variable et que ça déborde de la ligne et donne un truc moche - Dans ton code à un moment tu fais un if elseif else avec les formes, je n'ai pas trop regardés (désolé), mais rien qu'en ayant un aperçu du code, je pense que ça doit être possible d'utiliser la poo, avec une classe Carre, une classe Triangle et de la redéfinition de méthode, je peut me tromper vu que je n'ai pas trop regarder, mais mon professeur d'objet m'à toujours dit qu'en POO, les if elseif doivent être rare! - Je ne sais pas si tu l'a fait, je ne me souviens plus, mais je le dis parce que c'est courant : dans un constructeur on ne doit pas appeler de méthode, l'objet n'étant pas totalement construit, il peut y avoir un risque de null pointer exception. Voilà, bon courage! Mais merci quand même pour ce tutoriel qui reste très pédagogue et nous apprends des choses Bonne continuation.

medsaouabi # Posté hier à 15:33:32

Merci beaucoup
C'est parfait

Avis : Très
bon

Voir tous les commentaires