

CHAPTER 1

Introduction to Computers and Java

starting out with >>>

JAVA™

From Control Structures through Objects

6TH EDITION



TONY GADDIS

Chapter Topics

Chapter 1 discusses the following main topics:

- Introduction
- Why Program?
- Computer Systems: Hardware and Software
- Programming Languages
- What Is a Program Made Of?
- The Programming Process
- Object-Oriented Programming

Java History

- 1991 - Green Team started by Sun Microsystems (now owned by Oracle).
- *7 Handheld controller for multiple entertainment systems.
- There was a need for a programming language that would run on various devices.
- Java (first named Oak) was developed for this purpose.

Introduction

- Java enabled web browser (*HotJava*) demonstrated at 1995 Sun World conference.
- Java incorporated into Netscape shortly after.
- Java is “cross platform”, meaning that it can run on various computer operating systems.

Java Applications and Applets

- Java programs can be of two types:
 - Applications
 - Stand-alone programs that run without the aid of a web browser.
 - Relaxed security model since the user runs the program locally.
 - Applets
 - Small applications that require the use of a Java enabled web browser to run.
 - Enhanced security model since the user merely goes to a web page and the applet runs itself.

Why Program?

- Computers are tools that can be programmed to perform many functions, such as:
 - spreadsheets
 - games
 - databases
 - etc.
 - word processing
- Computers are versatile because they can be programmed.
- Computer Programmers implement programs that perform these functions.

Why Program?

Aspects of a computer program that must be designed:

- The logical flow of the instructions
- The mathematical procedures
- The layout of the programming statements
- The appearance of the screens
- The way information is presented to the user
- The program's "user friendliness"
- Manuals, help systems, and/or other forms of written documentation.

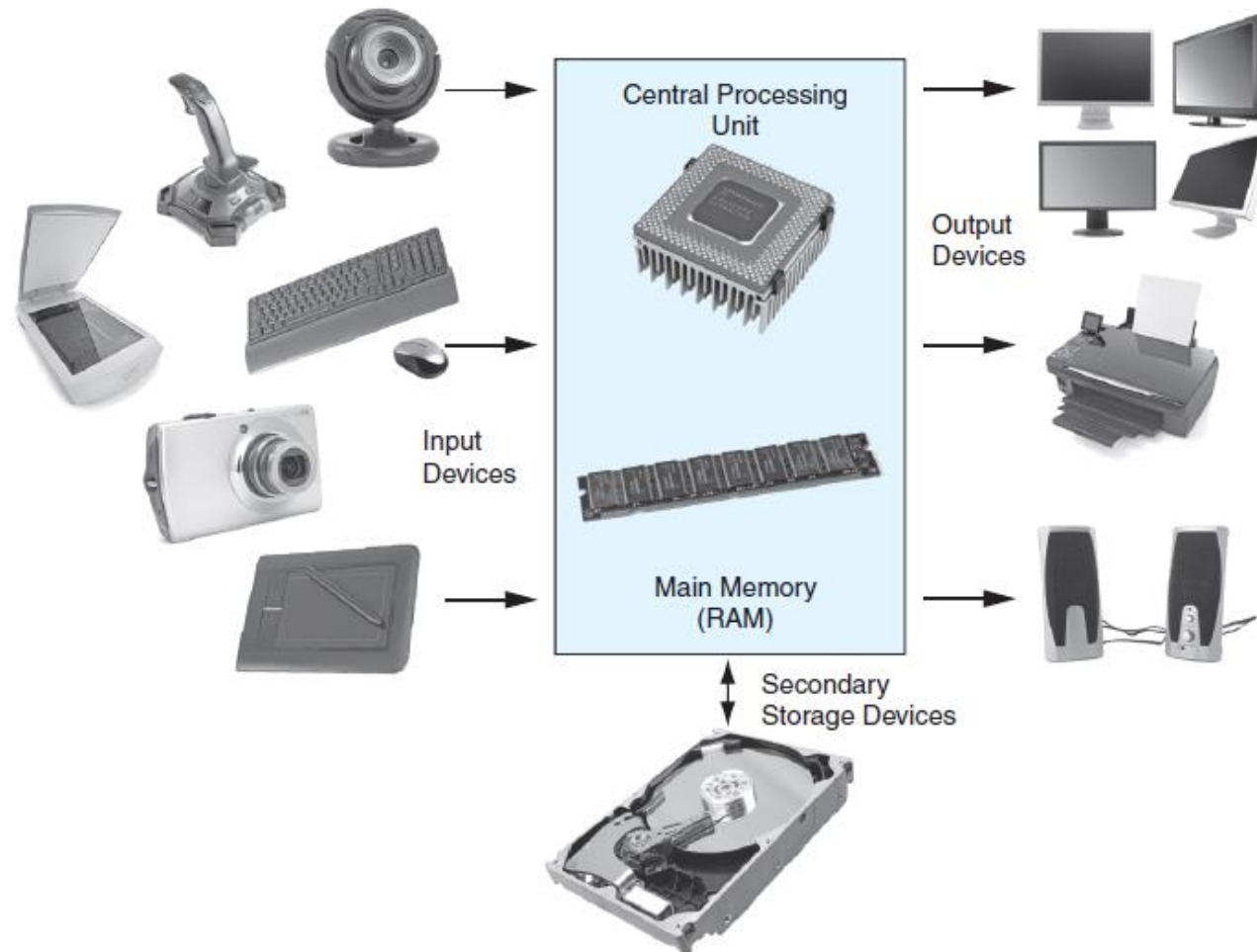
Why Program?

- Programs must be analytically correct as well.
- Programs rarely work the first time they are programmed.
- Programmers must perform the following on a continual basis:
 - analyze,
 - experiment,
 - correct, and
 - redesign.
- Programming languages have strict rules, known as *syntax*, that must be carefully followed.

Computer Systems: Hardware

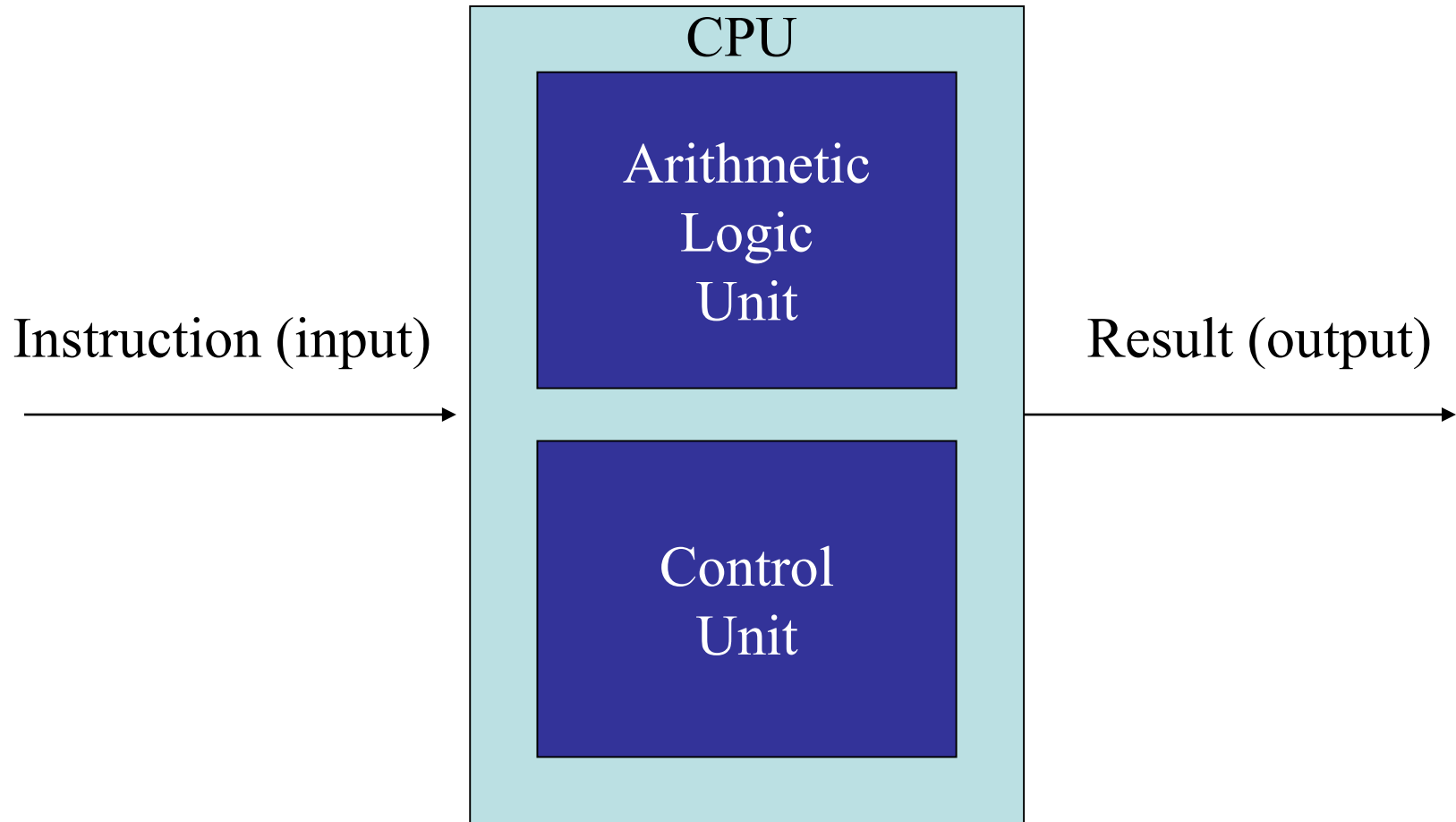
- Computer hardware components are the physical pieces of the computer.
- The major hardware components of a computer are:
 - The central processing unit (CPU)
 - Main memory
 - Secondary storage devices
 - Input and Output devices

Computer Systems: Hardware



Computer Systems: Hardware

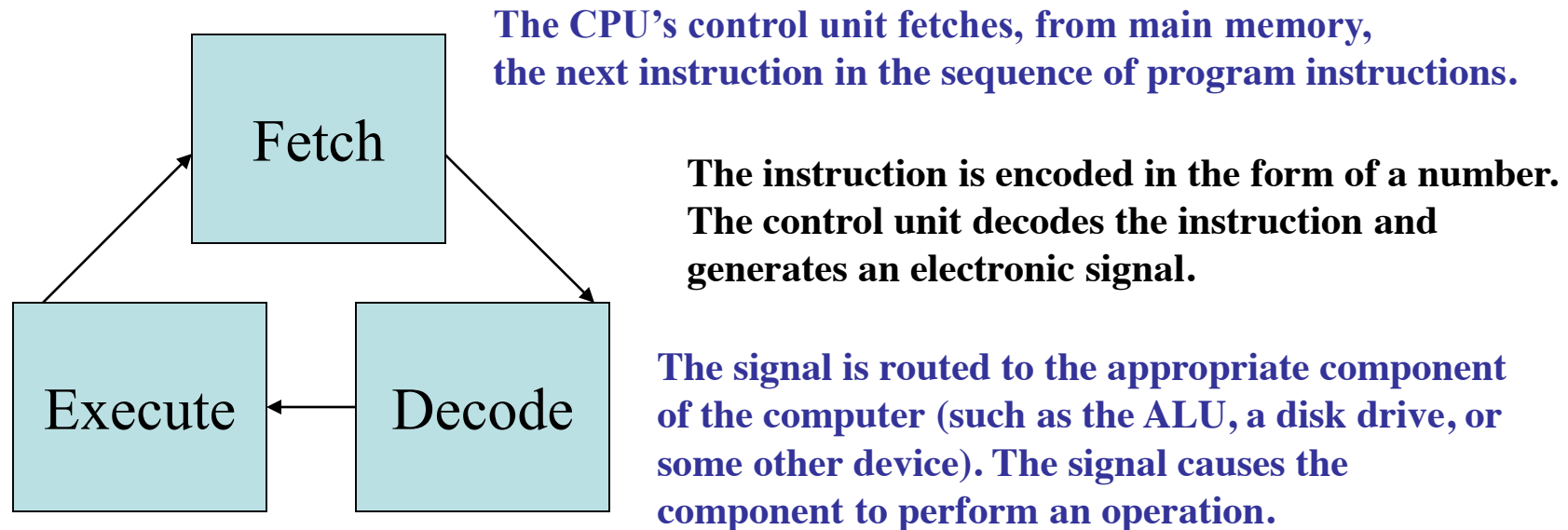
Central Processing Unit



Computer Systems: Hardware

Central Processing Unit

- The CPU performs the fetch, decode, execute cycle in order to process program information.



Computer Systems: Hardware

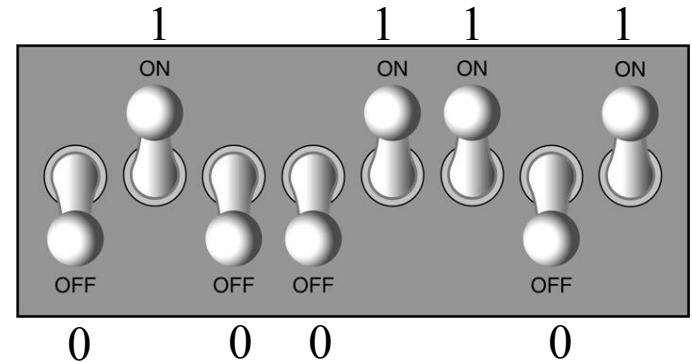
Main Memory

- Commonly known as *random-access memory* (*RAM*)
- RAM contains:
 - currently running programs
 - data used by those programs.
- RAM is divided into units called *bytes*.
- A byte consists of eight *bits* that may be either on or off.

Computer Systems: Hardware

Main Memory

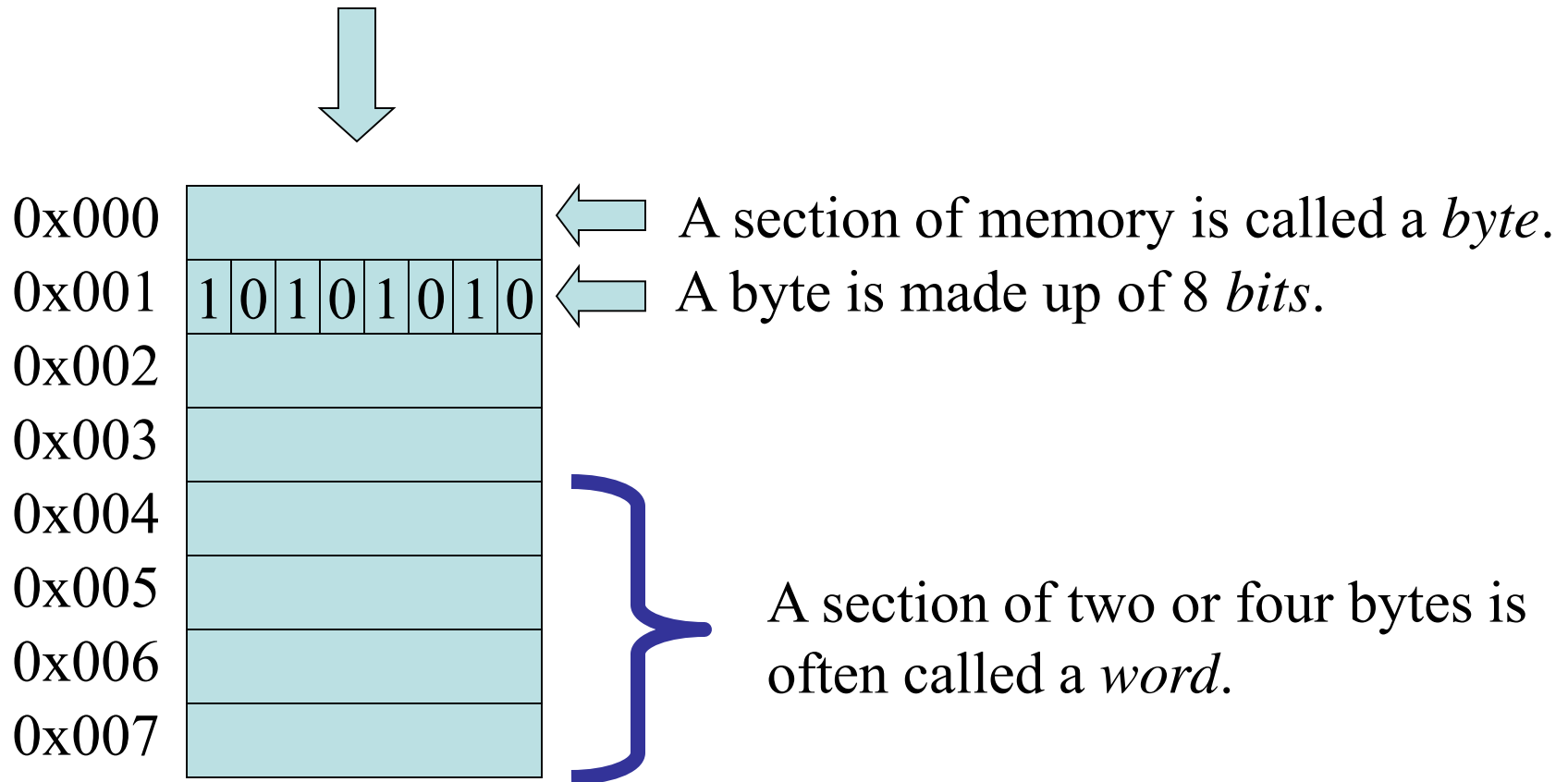
- A bit is either on or off:
 - 1 = on
 - 0 = off
- The bits form a pattern that represents a character or a number.
- Each byte in memory is assigned a unique number known as an *address*.
- RAM is *volatile*, which means that when the computer is turned off, the contents of RAM are erased.



Computer Systems: Hardware

Main Memory

Main memory can be visualized as a column or row of cells.



Computer Systems: Hardware

Secondary Storage Devices

- Secondary storage devices are capable of storing information for longer periods of time (*non-volatile*).
- Common Secondary Storage devices:
 - Disk drive
 - Solid state drive
 - External drive
 - USB drive
 - CD drive
 - DVD drive

Computer Systems: Hardware

Input Devices

- Input is any data the computer collects from the outside world.
- That data comes from devices known as *input devices*.
- Common input devices:
 - Keyboard
 - Mouse
 - Scanner
 - Digital camera

Computer Systems: Hardware

Output Devices

- Output is any data the computer sends to the outside world.
- That data is displayed on devices known as *output devices*.
- Common output devices:
 - Monitors
 - Printers
- Some devices such as disk drives perform input and output and are called *I/O devices* (input/output).

Computer Systems: Software

- Software refers to the programs that run on a computer.
- There are two classifications of software:
 - Operating Systems
 - Application Software

Computer Systems: Software

Operating Systems

- An operating system is a set of programs that manages the computer's hardware devices and controls their processes.
- Most all modern operating systems are multitasking.

Computer Systems: Software

Operating Systems

- A multitasking operating system is capable of running multiple programs at once.
 - Unix
 - Linux
 - Mac OS
 - Windows
- The technique is called time sharing.
- A multitasking system divides the allocation of hardware resources and the attention of the CPU among all the executing programs.

Computer Systems: Software

Application Software

- *Application software* refers to programs that make the computer useful to the user.
- Application software provides a more specialized type of environment for the user to work in.
- Common application software:
 - Spreadsheets
 - Word processors
 - Accounting software
 - Tax software
 - Games

Programming Languages

- A program is a set of instructions a computer follows in order to perform a task.
- A programming language is a special language used to write computer programs.
- A computer program is a set of instructions that enable the computer to solve a problem or perform a task.
- Collectively, these instructions form an *algorithm*

Programming Languages

- An algorithm is a set of well defined steps to completing a task.
- The steps in an algorithm are performed sequentially.
- A computer needs the algorithm to be written in *machine language*.
- Machine language is written using *binary numbers*.
- The binary numbering system (base 2) only has two digits (0 and 1).

Programming Languages

- The binary numbers are encoded as a machine language.
- Each CPU has its own machine language.
 - Motorola 68000 series processors
 - Intel x86 series processors
 - ARM processors, etc.
- Example of a machine language instruction:
10110100000000101

Programming Languages

- In the distant past, programmers wrote programs in machine language.
- Programmers developed higher level programming languages to make things easier.
- The first of these was *assembler*.
- Assembler made things easier but was also processor dependent.

Programming Languages

- High level programming languages followed that were not processor dependent.
- Some common programming languages:

Java	C	Visual Basic
BASIC	C++	Python
COBOL	C#	Ruby
Pascal	PHP	JavaScript

Programming Languages

Common Language Elements

- There are some concepts that are common to virtually all programming languages.
- Common concepts:
 - Key words
 - Operators
 - Punctuation
 - Programmer-defined identifiers
 - Strict syntactic rules.

Programming Languages

Sample Program

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        String message = "Hello World";
        System.out.println(message);
    }
}
```

Programming Languages

Sample Program

- Key words in the sample program are:
 - `public`
 - `static`
 - `class`
 - `void`
- Key words are lower case (Java is a case sensitive language).
- Key words cannot be used as a programmer-defined identifier.

Programming Languages

- Semi-colons are used to end Java statements; however, not all lines of a Java program end a statement.
- Part of learning Java is to learn where to properly use the punctuation.

Programming Languages

Lines vs Statements

- There are differences between lines and statements when discussing source code.

```
System.out.println(  
    message);
```

- This is one Java statement written using two lines. Do you see the difference?
- A statement is a complete Java instruction that causes the computer to perform an action.

Programming Languages

Variables

- Data in a Java program is stored in memory.
- Variable names represent a location in memory.
- Variables in Java are sometimes called fields.
- Variables are created by the programmer who assigns it a programmer-defined identifier.

example: `int hours = 40;`

- In this example, the variable *hours* is created as an integer (more on this later) and assigned the value of 40.

Programming Languages

Variables

- Variables are simply a name given to represent a place in memory.

0x000	
0x001	
0x002	
0x003	
0x004	
0x005	
0x006	
0x007	

Programming Languages

Variables

The Java Virtual Machine (JVM) actually decides where the value will be placed in memory.

0x000	
0x001	
0x002	
0x003	72
0x004	
0x005	
0x006	
0x007	

Assume that the this variable declaration has been made.

```
int length = 72;
```

The variable length is a symbolic name for the memory location 0x003.

The Compiler and the Java Virtual Machine

- A programmer writes Java programming statements for a program.
- These statements are known as *source code*.
- A *text editor* is used to edit and save a Java *source code file*.
- Source code files have a *.java* file extension.
- A *compiler* is a program that translates source code into an executable form.

The Compiler and the Java Virtual Machine

- A compiler is run using a source code file as input.
- Syntax errors that may be in the program will be discovered during compilation.
- *Syntax errors* are mistakes that the programmer has made that violate the rules of the programming language.
- The compiler creates another file that holds the translated instructions.

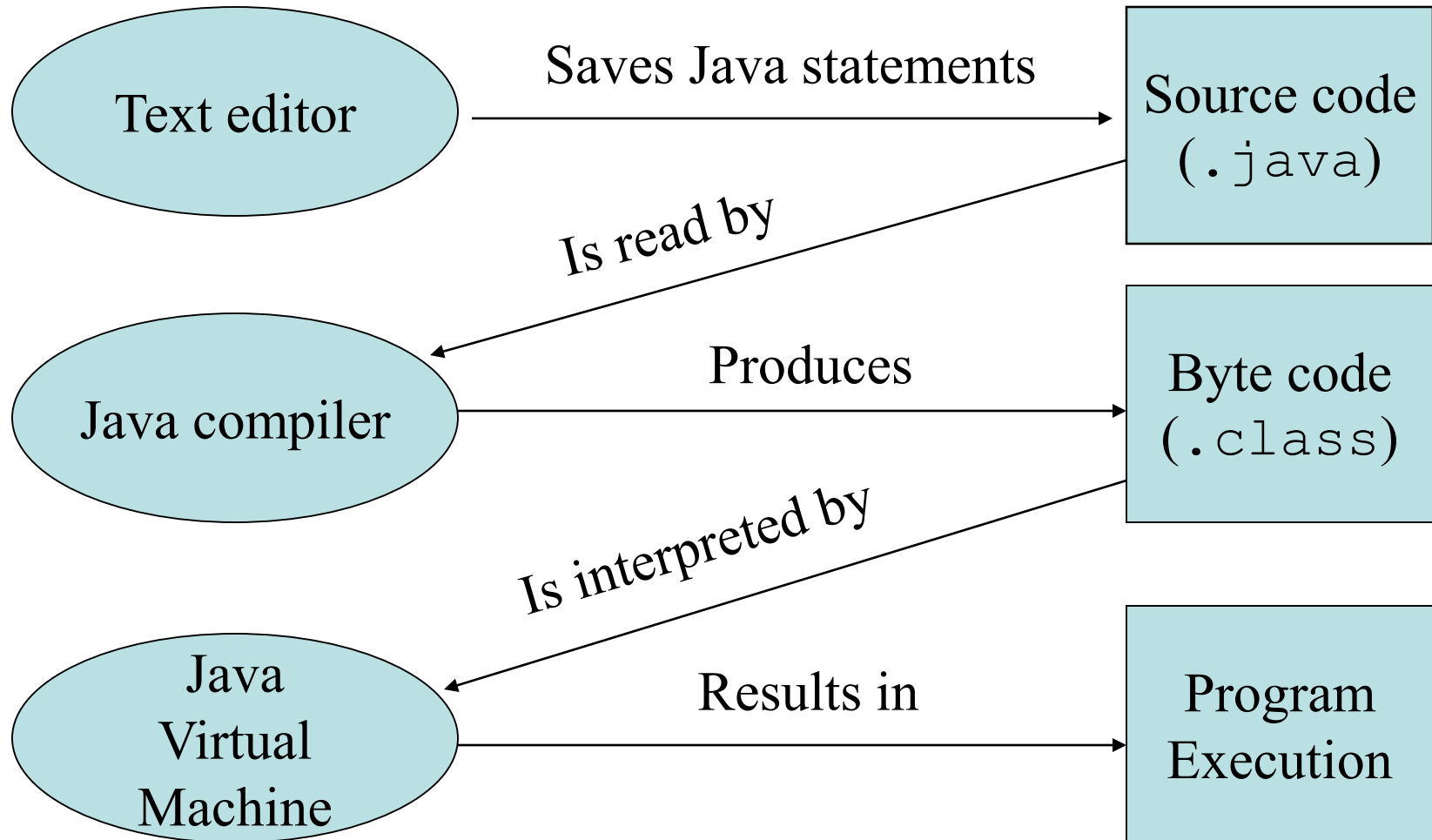
The Compiler and the Java Virtual Machine

- Most compilers translate source code into *executable* files containing *machine code*.
- The Java compiler translates a Java source file into a file that contains *byte code* instructions.
- Byte code instructions are the machine language of the *Java Virtual Machine (JVM)* and cannot be directly executed directly by the CPU.

The Compiler and the Java Virtual Machine

- Byte code files end with the *.class* file extension.
- The JVM is a program that *emulates* a micro-processor.
- The JVM executes instructions as they are read.
- JVM is often called an *interpreter*.
- Java is often referred to as an *interpreted language*.

Program Development Process



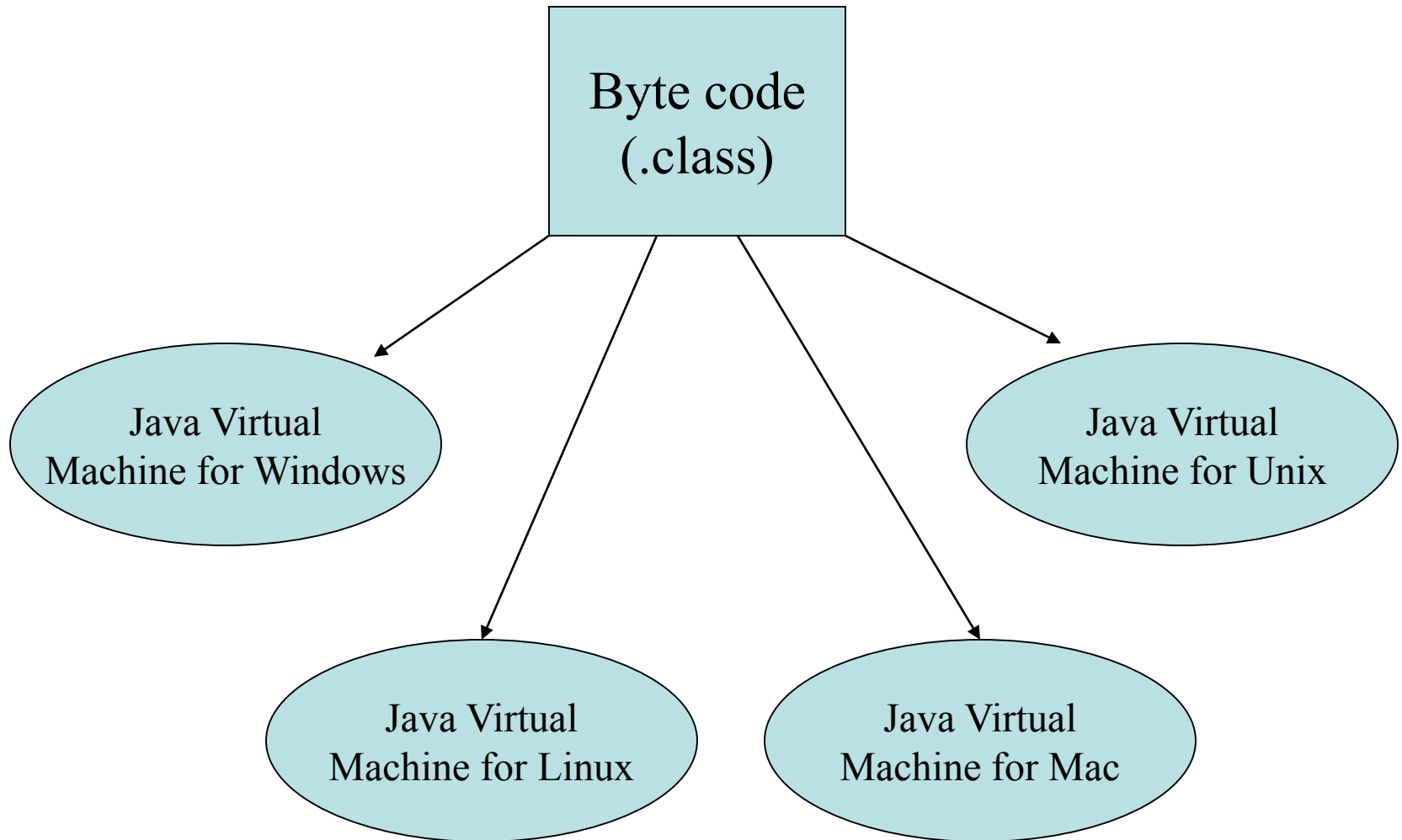
Portability

- *Portable* means that a program may be written on one type of computer and then run on a wide variety of computers, with little or no modification.
- Java byte code runs on the JVM and not on any particular CPU; therefore, compiled Java programs are highly portable.
- JVMs exist on many platforms:
 - Windows
 - Mac
 - Linux
 - Unix
 - BSD
 - Etc.

Portability

- With most programming languages, portability is achieved by compiling a program for each CPU it will run on.
- Java provides an JVM for each platform so that programmers do not have to recompile for different platforms.

Portability



Java Versions

- The software you use to write Java programs is called the Java Development Kit, or JDK.
- There are different editions of the JDK:
 - Java SE - Java2 *Standard Edition*.
 - Java EE - Java2 *Enterprise Edition*.
 - Java ME - Java2 *Micro Edition*.
- Available for download at <http://java.oracle.com>

Compiling a Java Program

- The Java compiler is a *command line* utility.
- The command to compile a program is:
java filename.java
- javac is the Java compiler.
- The `.java` file extension must be used.

Example: To compile a java source code file named Payroll.java you would use the command:

javac Payroll.java

The Programming Process

1. Clearly define what the program is to do.
2. Visualize the program running on the computer.
3. Use design tools to create a model of the program.
4. Check the model for logical errors.

The Programming Process

5. Enter the code and compile it.
6. Correct any errors found during compilation.
Repeat Steps 5 and 6 as many times as necessary.
7. Run the program with test data for input.
8. Correct any runtime errors found while running the program.
Repeat Steps 5 through 8 as many times as necessary.
9. Validate the results of the program.

Software Engineering

- Encompasses the whole process of crafting computer software.
- Software engineers perform several tasks in the development of complex software projects.
 - designing,
 - writing,
 - testing,
 - debugging,
 - documenting,
 - modifying, and
 - maintaining.

Software Engineering

- Software engineers develop:
 - program specifications,
 - diagrams of screen output,
 - diagrams representing the program components and the flow of data,
 - pseudocode,
 - examples of expected input and desired output.

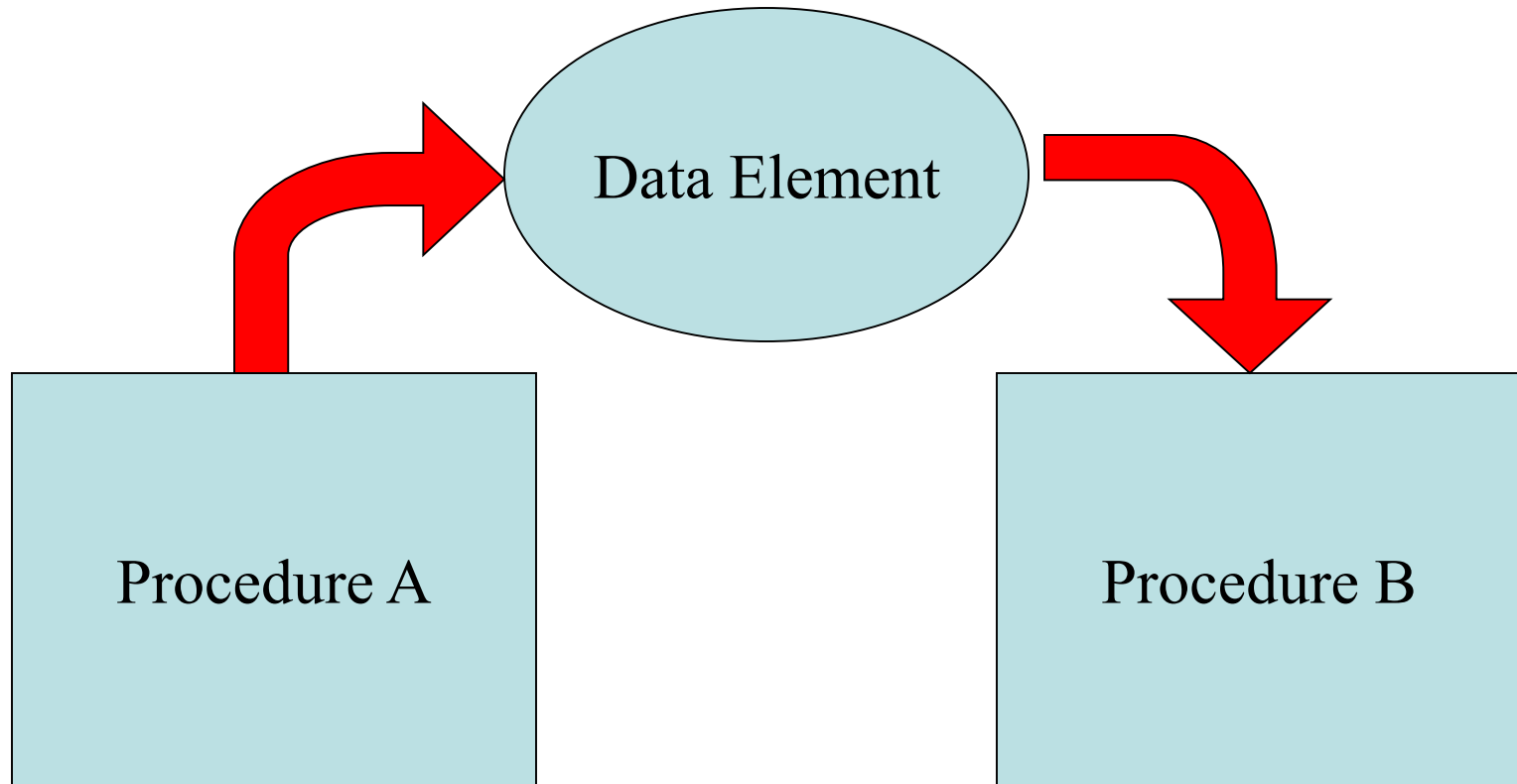
Software Engineering

- Software engineers also use special software designed for testing programs.
- Most commercial software applications are large and complex.
- Usually a team of programmers, not a single individual, develops them.
- Program requirements are thoroughly analyzed and divided into subtasks that are handled by
 - individual teams
 - individuals within a team.

Procedural Programming

- Older programming languages were procedural.
- A *procedure* is a set of programming language statements that, together, perform a specific task.
- Procedures typically operate on data items that are separate from the procedures.
- In a procedural program, the data items are commonly passed from one procedure to another.

Procedural Programming



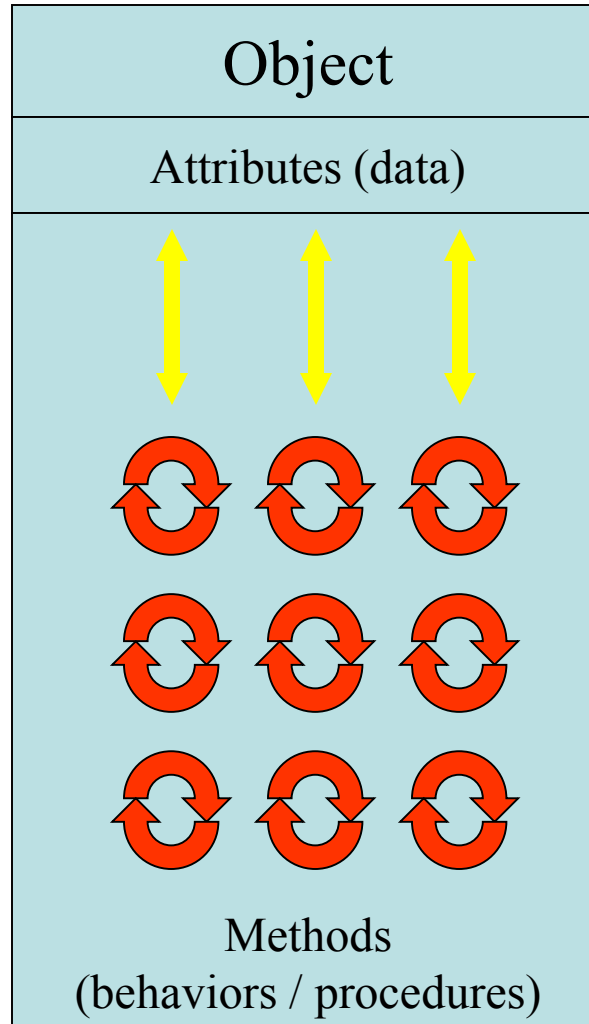
Procedural Programming

- In procedural programming, procedures are developed to operate on the program's data.
- Data in the program tends to be global to the entire program.
- Data formats might change and thus, the procedures that operate on that data must change.

Object-Oriented Programming

- Object-oriented programming is centered on creating objects rather than procedures.
- Objects are a melding of data and procedures that manipulate that data.
- Data in an object are known as *attributes*.
- Procedures in an object are known as *methods*.

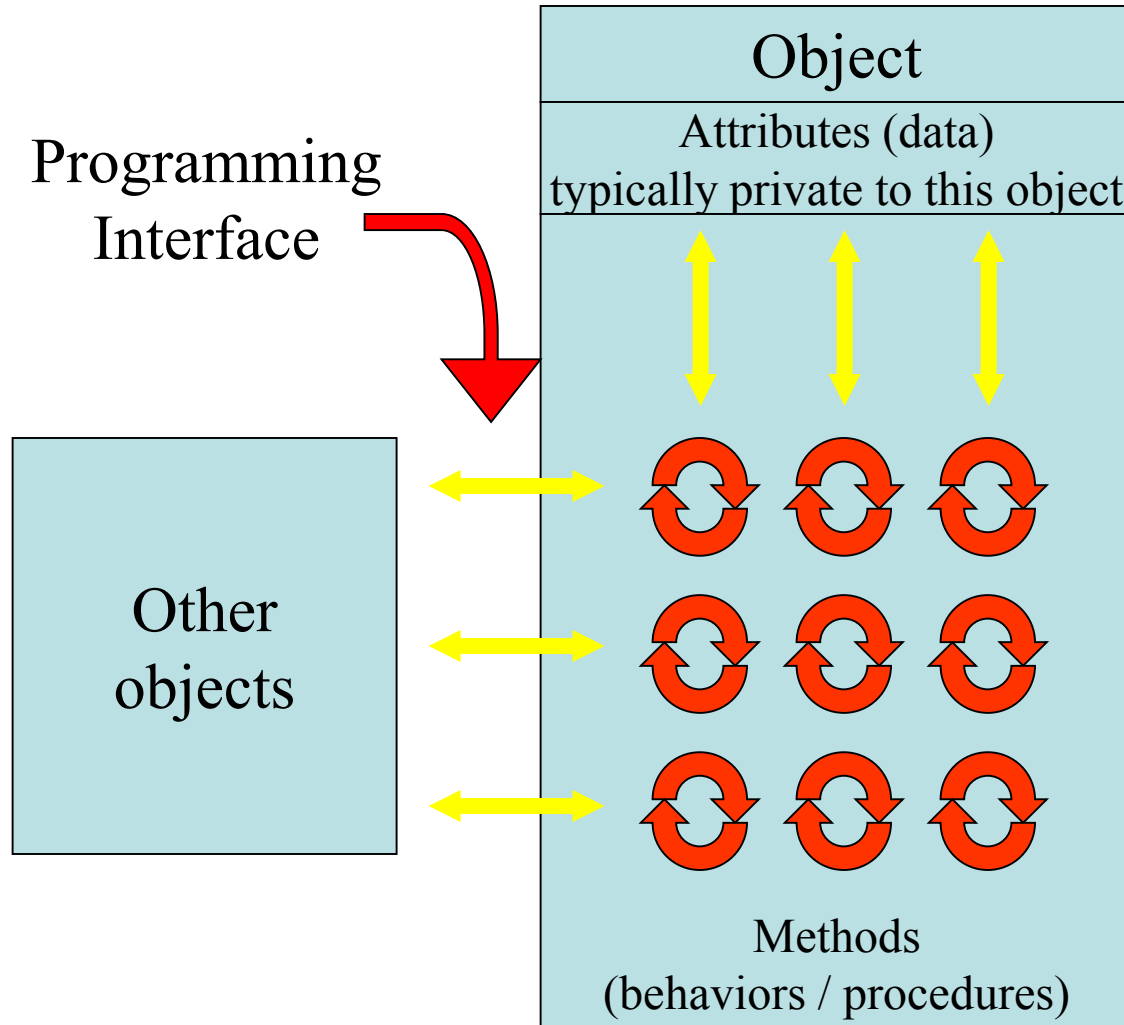
Object-Oriented Programming



Object-Oriented Programming

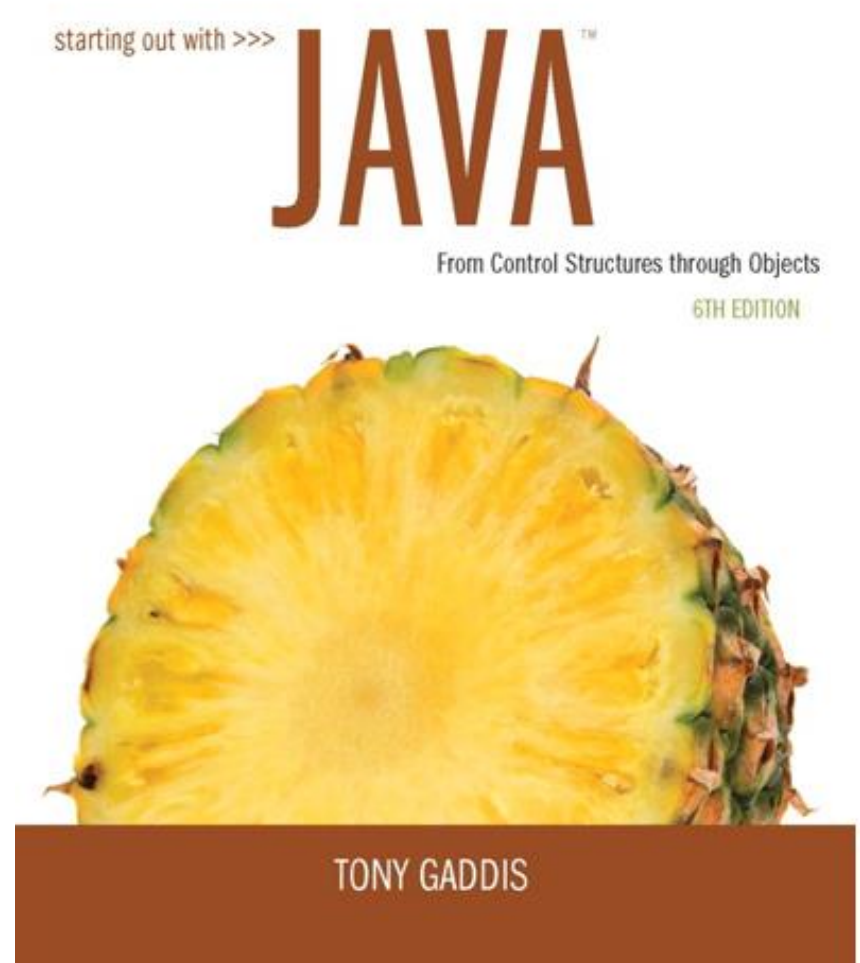
- Object-oriented programming combines data and behavior via *encapsulation*.
- *Data hiding* is the ability of an object to hide data from other objects in the program.
- Only an objects methods should be able to directly manipulate its attributes.
- Other objects are allowed manipulate an object's attributes via the object's methods.
- This indirect access is known as a *programming interface*.

Object-Oriented Programming



CHAPTER 2

Java Fundamentals



Chapter Topics

Chapter 2 discusses the following main topics:

- The Parts of a Java Program
- The `print` and `println` Methods, and the Java API
- Variables and Literals
- Primitive Data Types
- Arithmetic Operators
- Combined Assignment Operators

Chapter Topics (2)

- Creating named constants with `final`
- The `String` class
- Scope
- Comments
- Programming style
- Using the `Scanner` class for input
- Dialog boxes

Parts of a Java Program

- A Java source code file contains one or more Java classes.
- If more than one class is in a source code file, only one of them may be public.
- The public class and the filename of the source code file must match.
ex: A class named *Simple* must be in a file named *Simple.java*
- Each Java class can be separated into parts.

Parts of a Java Program

- See example: [Simple.java](#)
- To compile the example:
 - **javac Simple.java**
 - Notice the `.java` file extension is needed.
 - This will result in a file named *Simple.class* being created.
- To run the example:
 - **java Simple**
 - Notice there is no file extension here.
 - The *java* command assumes the extension is `.class`.

Analyzing The Example

```
// This is a simple Java program.
```

This is a Java comment. It is ignored by the compiler.

```
public class Simple
```

This is the class header for the class Simple

```
{
```

This area is the body of the class Simple. All of the data and methods for this class will be between these curly braces.

```
}
```

Analyzing The Example

```
// This is a simple Java program.
```

```
public class Simple  
{
```

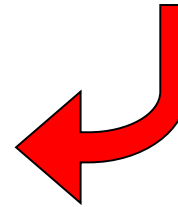
```
    public static void main(String[] args)
```

```
    {
```

```
    }
```

```
}
```

This is the method header for the main method. The main method is where a Java application begins.

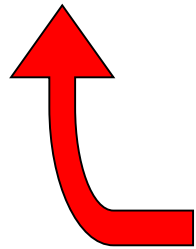


This area is the body of the main method. All of the actions to be completed during the main method will be between these curly braces.

Analyzing The Example

// This is a simple Java program.

```
public class Simple
{
    public static void main(String [] args)
    {
        System.out.println("Programming is great fun!");
    }
}
```



**This is the Java Statement that
is executed when the program runs.**

Parts of a Java Program

- Comments
 - The line is ignored by the compiler.
 - The comment in the example is a single-line comment.
- Class Header
 - The class header tells the compiler things about the class such as what other classes can use it (**public**) and that it is a Java class (**class**), and the name of that class (**Simple**).
- Curly Braces
 - When associated with the class header, they define the scope of the class.
 - When associated with a method, they define the scope of the method.

Parts of a Java Program

- The `main` Method
 - This line must be exactly as shown in the example (except the *args* variable name can be programmer defined).
 - This is the line of code that the *java* command will run first.
 - This method starts the Java program.
 - Every Java application must have a `main` method.
- Java Statements
 - When the program runs, the statements within the `main` method will be executed.
 - Can you see what the line in the example will do?

Java Statements

- If we look back at the previous example, we can see that there is only one line that ends with a semi-colon.

```
System.out.println("Programming is great fun!");
```

- This is because it is the only Java statement in the program.
- The rest of the code is either a comment or other Java framework code.

Java Statements

- Comments are ignored by the Java compiler so they need no semi-colons.
- Other Java code elements that do not need semi colons include:
 - class headers
 - Terminated by the code within its curly braces.
 - method headers
 - Terminated by the code within its curly braces.
 - curly braces
 - Part of framework code that needs no semi-colon termination.

Short Review

- Java is a case-sensitive language.
- All Java programs must be stored in a file with a .java file extension.
- Comments are ignored by the compiler.
- A .java file may contain many classes but may only have one public class.
- If a .java file has a public class, the class must have the same name as the file.

Short Review

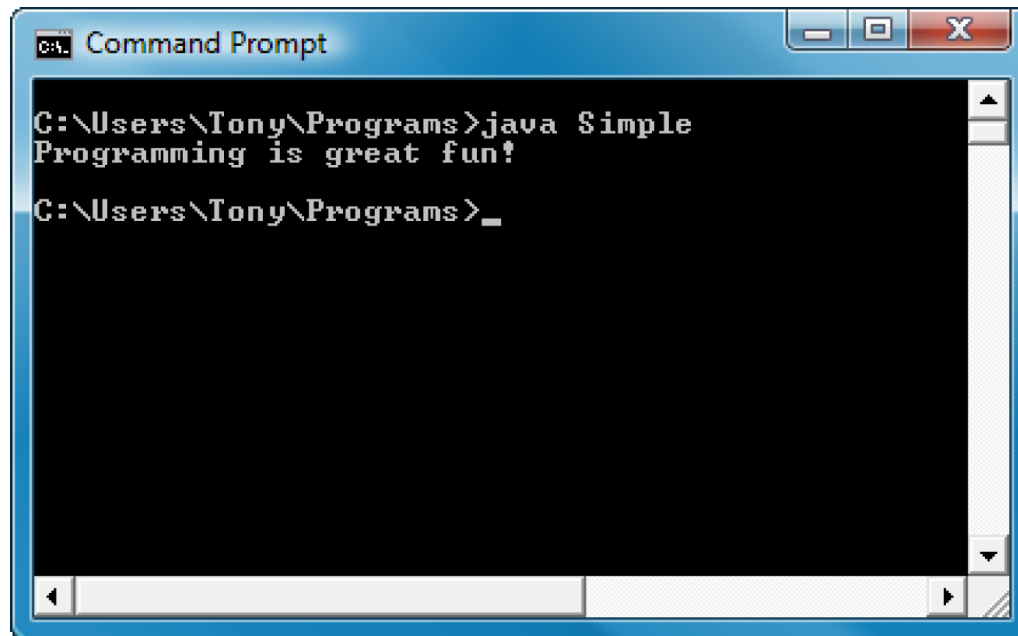
- Java applications must have a `main` method.
- For every left brace, or opening brace, there must be a corresponding right brace, or closing brace.
- Statements are terminated with semicolons.
 - Comments, class headers, method headers, and braces are not considered Java statements.

Special Characters

//	double slash	Marks the beginning of a single line comment.
()	open and close parenthesis	Used in a method header to mark the <i>parameter list</i>.
{ }	open and close curly braces	Encloses a group of statements, such as the contents of a class or a method.
“ ”	quotation marks	Encloses a string of characters, such as a message that is to be printed on the screen
;	semi-colon	Marks the end of a complete programming statement

Console Output

- Many of the programs that you will write will run in a console window.



```
C:\Users\Tony\Programs>java Simple
Programming is great fun!
C:\Users\Tony\Programs>_
```

Console Output

- The console window that starts a Java application is typically known as the *standard output* device.
- The *standard input* device is typically the keyboard.
- Java sends information to the standard output device by using a Java class stored in the standard Java library.

Console Output

- Java classes in the standard Java library are accessed using the Java Applications Programming Interface (API).
- The standard Java library is commonly referred to as the *Java API*.

Console Output

- The previous example uses the line:

```
System.out.println("Programming is great fun!");
```
- This line uses the `System` class from the standard Java library.
- The `System` class contains methods and objects that perform system level tasks.
- The `out` object, a member of the `System` class, contains the methods `print` and `println`.

Console Output

- The `print` and `println` methods actually perform the task of sending characters to the output device.
- The line:
`System.out.println("Programming is great fun!");`
is pronounced: System dot out dot println ...
- The value inside the parenthesis will be sent to the output device (in this case, a string).

Console Output

- The `println` method places a newline character at the end of whatever is being printed out.
- The following lines:

```
System.out.println("This is being printed out");  
System.out.println("on two separate lines.");
```

Would be printed out on separate lines since the first statement sends a newline command to the screen.

Console Output

- The `print` statement works very similarly to the `println` statement.
- However, the `print` statement does not put a newline character at the end of the output.
- The lines:

```
System.out.print("These lines will be");  
System.out.print("printed on");  
System.out.println("the same line.");
```

Will output:

These lines will beprinted onthe same line.

Notice the odd spacing? Why are some words run together?

Console Output

- For all of the previous examples, we have been printing out strings of characters.
- Later, we will see that much more can be printed.
- There are some special characters that can be put into the output.

```
System.out.print("This line will have a newline at the end.\n");
```

- The `\n` in the string is an escape sequence that represents the newline character.
- Escape sequences allow the programmer to print characters that otherwise would be unprintable.

Java Escape Sequences

<code>\n</code>	newline	Advances the cursor to the next line for subsequent printing
<code>\t</code>	tab	Causes the cursor to skip over to the next tab stop
<code>\b</code>	backspace	Causes the cursor to back up, or move left, one position
<code>\r</code>	carriage return	Causes the cursor to go to the beginning of the current line, not the next line
<code>\\</code>	backslash	Causes a backslash to be printed
<code>\'</code>	single quote	Causes a single quotation mark to be printed
<code>\"</code>	double quote	Causes a double quotation mark to be printed

Java Escape Sequences

- Even though the escape sequences are comprised of two characters, they are treated by the compiler as a single character.

```
System.out.print("These are our top sellers:\n");  
System.out.print("\tComputer games\n\tCoffee\n ");  
System.out.println("\tAspirin");
```

Would result in the following output:

```
These are our top seller:  
    Computer games  
    Coffee  
    Asprin
```

- With these escape sequences, complex text output can be achieved.

Variables and Literals

- A variable is a named storage location in the computer's memory.
- A literal is a value that is written into the code of a program.
- Programmers determine the number and type of variables a program will need.
- See example: [Variable.java](#)

Variables and Literals

This line is called
a *variable declaration*.

```
int value;
```

The following line is known
as an assignment statement.

```
value = 5;
```

0x000

0x001

0x002

0x003



The value 5
is stored in
memory.

This is a string *literal*. It will be printed *as is*.

```
System.out.print("The value is ");  
System.out.println(value);
```

The integer 5 will
be printed out here.
Notice no quote marks?

The + Operator

- The + operator can be used in two ways.
 - as a concatenation operator
 - as an addition operator
- If either side of the + operator is a string, the result will be a string.

```
System.out.println("Hello " + "World");  
System.out.println("The value is: " + 5);  
System.out.println("The value is: " + value);  
System.out.println("The value is: " + "\n" + 5);
```

String Concatenation

- Java commands that have string literals must be treated with care.
- A string literal value cannot span lines in a Java source code file.

```
System.out.println("This line is too long and now it  
has spanned more than one line, which will cause a  
syntax error to be generated by the compiler. ");
```

String Concatenation

- The String concatenation operator can be used to fix this problem.

```
System.out.println("These lines are " +  
                    "are now ok and will not " +  
                    "cause the error as before.");
```

- String concatenation can join various data types.

```
System.out.println("We can join a string to " +  
                    "a number like this: " + 5);
```

String Concatenation

- The Concatenation operator can be used to format complex String objects.

```
System.out.println("The following will be printed " +  
    "in a tabbed format: " +  
    "\n\tFirst = " + 5 * 6 + ", " +  
    "\n\tSecond = " (6 + 4) + ", " +  
    "\n\tThird = " + 16.7 + ".");
```

- Notice that if an addition operation is also needed, it must be put in parenthesis.

Identifiers

- Identifiers are programmer-defined names for:
 - classes
 - variables
 - methods
- Identifiers may not be any of the Java reserved keywords.

Identifiers

- Identifiers must follow certain rules:
 - An identifier may only contain:
 - letters a–z or A–Z,
 - the digits 0–9,
 - underscores (`_`), or
 - the dollar sign (`$`)
 - The first character may not be a digit.
 - Identifiers are case sensitive.
 - `itemsOrdered` is not the same as `itemsordered`.
 - Identifiers cannot include spaces.

Java Reserved Keywords

abstract	double	instanceof	static
assert	else	int	strictfp
boolean	enum	interface	super
break	extends	long	switch
byte	false	native	synchronized
case	for	new	this
catch	final	null	throw
char	finally	package	throws
class	float	private	transient
const	goto	protected	true
continue	if	public	try
default	implements	return	void
do	import	short	volatile
			while

Variable Names

- Variable names should be descriptive.
- Descriptive names allow the code to be more readable; therefore, the code is more maintainable.
- Which of the following is more descriptive?

```
double tr = 0.0725;  
double salesTaxRate = 0.0725;
```
- Java programs should be *self-documenting*.

Java Naming Conventions

- Variable names should begin with a lower case letter and then switch to title case thereafter:

Ex: `int caTaxRate`

- Class names should be all title case.

Ex: `public class BigLittle`

- More Java naming conventions can be found at:

<http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html>

- A general rule of thumb about naming variables and classes are that, with some exceptions, their names tend to be nouns or noun phrases.

Primitive Data Types

- Primitive data types are built into the Java language and are not derived from classes.
- There are 8 Java primitive data types.
 - byte
 - short
 - int
 - long
 - float
 - double
 - boolean
 - char

Numeric Data Types

byte	1 byte	Integers in the range -128 to +127
short	2 bytes	Integers in the range of -32,768 to +32,767
int	4 bytes	Integers in the range of -2,147,483,648 to +2,147,483,647
long	8 bytes	Integers in the range of -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
float	4 bytes	Floating-point numbers in the range of $\pm 3.410 \times 10^{-38}$ to ± 3.41038 , with 7 digits of accuracy
double	8 bytes	Floating-point numbers in the range of $\pm 1.710 \times 10^{-308}$ to ± 1.710308 , with 15 digits of accuracy

Variable Declarations

- Variable Declarations take the following form:
 - *DataType VariableName;*
 - `byte inches;`
 - `short month;`
 - `int speed;`
 - `long timeStamp;`
 - `float salesCommission;`
 - `double distance;`

Integer Data Types

- `byte`, `short`, `int`, and `long` are all integer data types.
- They can hold whole numbers such as 5, 10, 23, 89, etc.
- Integer data types cannot hold numbers that have a decimal point in them.
- Integers embedded into Java source code are called *integer literals*.
- See Example: [IntegerVariables.java](#)

Floating Point Data Types

- Data types that allow fractional values are called *floating-point* numbers.
 - 1.7 and -45.316 are floating-point numbers.
- In Java there are two data types that can represent floating-point numbers.
 - `float` - also called *single precision* (7 decimal points).
 - `double` - also called *double precision* (15 decimal points).

Floating Point Literals

- When floating point numbers are embedded into Java source code they are called *floating point literals*.
- The default type for floating point literals is `double`.
 - 29.75, 1.76, and 31.51 are `double` data types.
- Java is a *strongly-typed* language.
- See example: [Sale.java](#)

Floating Point Literals

- A `double` value is not compatible with a `float` variable because of its size and precision.
 - `float number;`
 - `number = 23.5; // Error!`
- A `double` can be forced into a `float` by appending the letter `F` or `f` to the literal.
 - `float number;`
 - `number = 23.5F; // This will work.`

Floating Point Literals

- Literals cannot contain embedded currency symbols or commas.
 - `grossPay = $1,257.00; // ERROR!`
 - `grossPay = 1257.00; // Correct.`
- Floating-point literals can be represented in *scientific notation*.
 - $47,281.97 == 4.728197 \times 10^4$.
- Java uses *E notation* to represent values in scientific notation.
 - $4.728197 \times 10^4 == 4.728197\text{E}4$.

Scientific and E Notation

Decimal Notation	Scientific Notation	E Notation
247.91	2.4791×10^2	2.4791E2
0.00072	7.2×10^{-4}	7.2E-4
2,900,000	2.9×10^6	2.9E6

See example: [SunFacts.java](#)

The boolean Data Type

- The Java boolean data type can have two possible values.
 - true
 - false
- The value of a boolean variable may only be copied into a boolean variable.

See example: [TrueFalse.java](#)

The `char` Data Type

- The Java `char` data type provides access to single characters.
- `char` literals are enclosed in single quote marks.
 - ‘a’, ‘Z’, ‘\n’, ‘1’
- Don’t confuse `char` literals with string literals.
 - `char` literals are enclosed in single quotes.
 - String literals are enclosed in double quotes.

See example: [Letters.java](#)

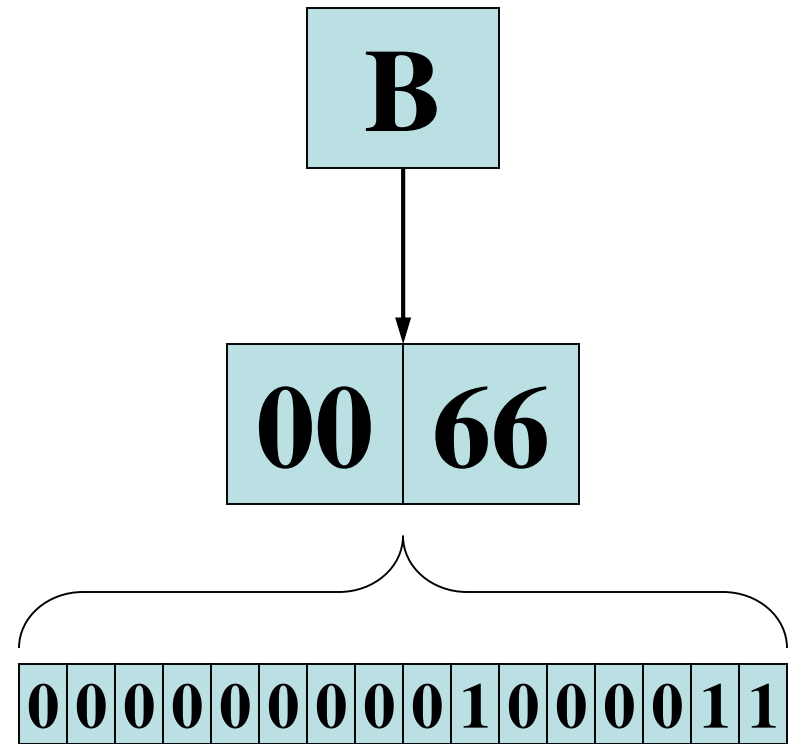
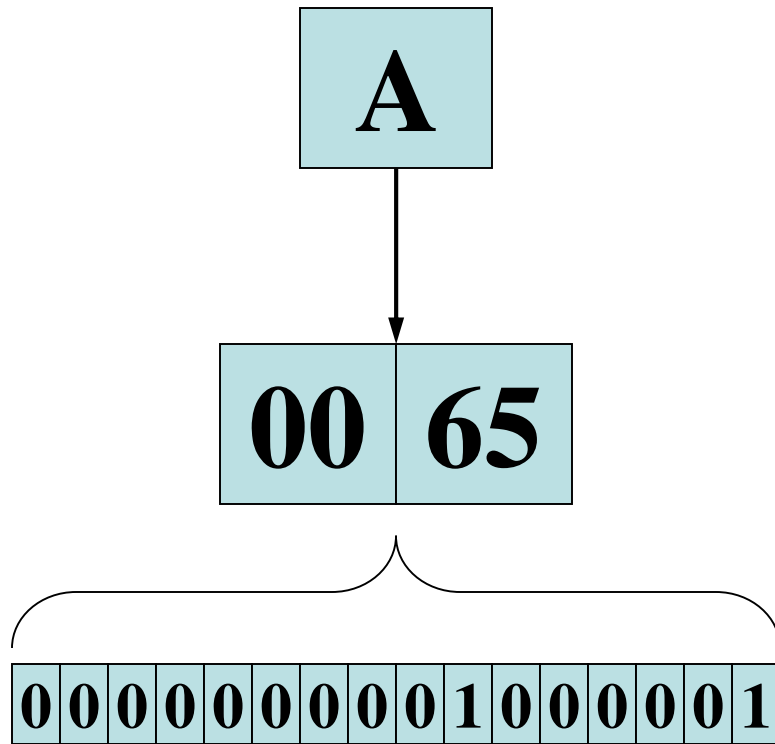
Unicode

- Internally, characters are stored as numbers.
- Character data in Java is stored as Unicode characters.
- The Unicode character set can consist of 65536 (2^{16}) individual characters.
- This means that each character takes up 2 bytes in memory.
- The first 256 characters in the Unicode character set are compatible with the ASCII* character set.

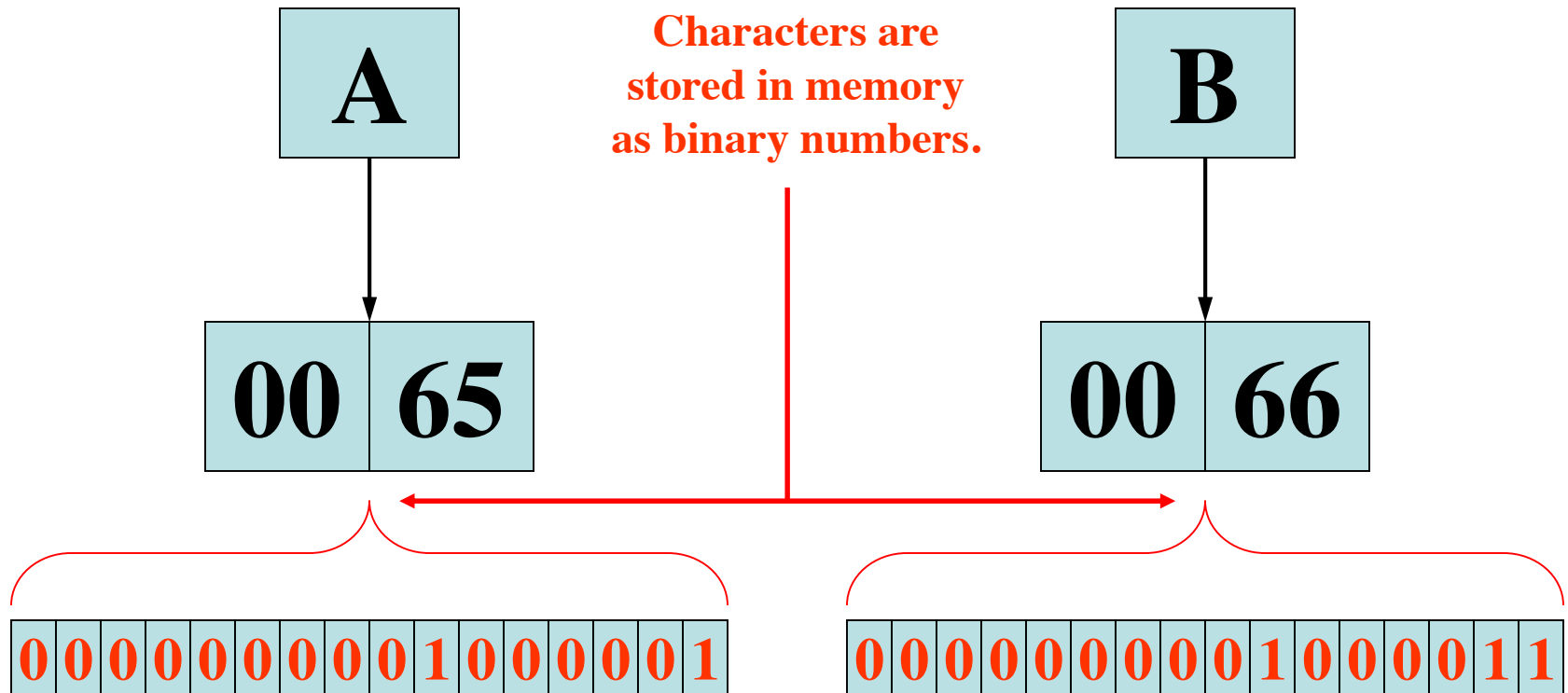
See example: [Letters2.java](#)

*American Standard Code for Information Interchange

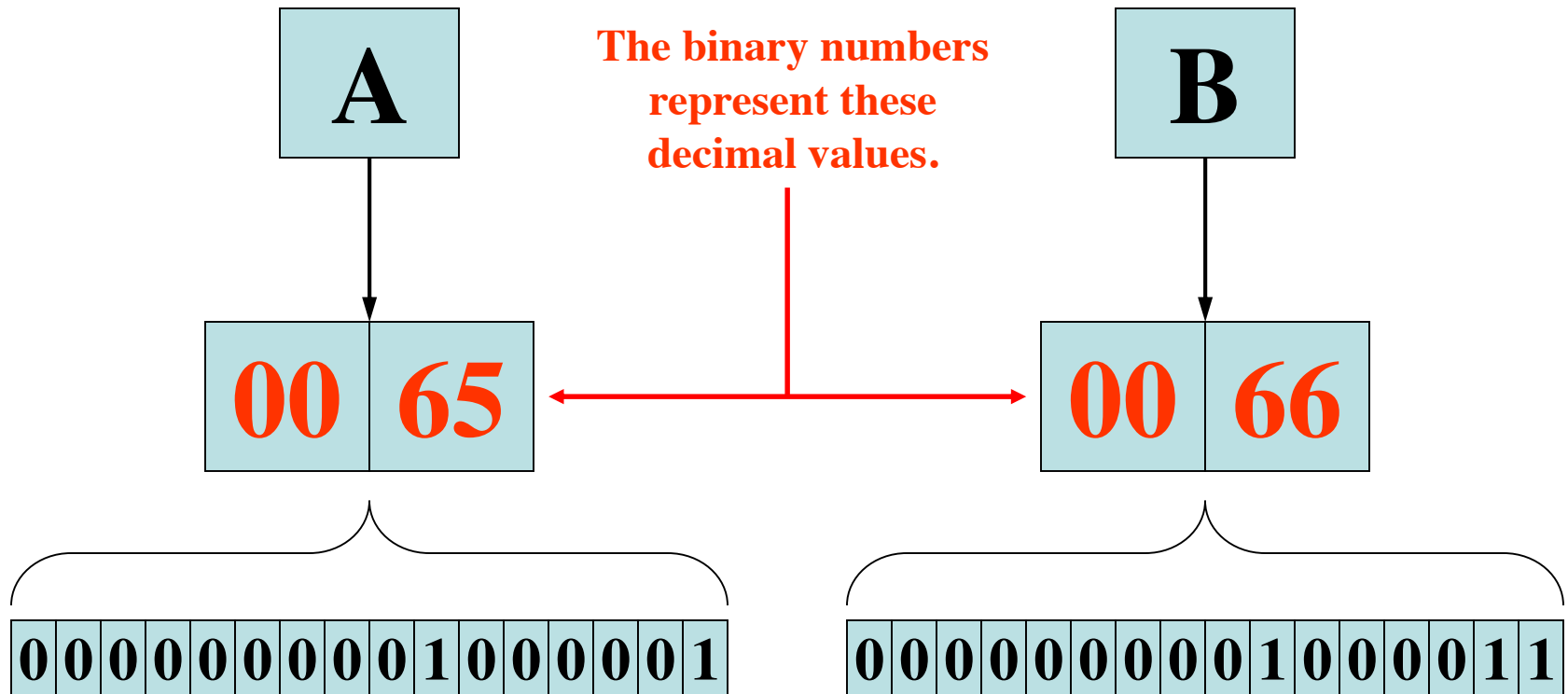
Unicode



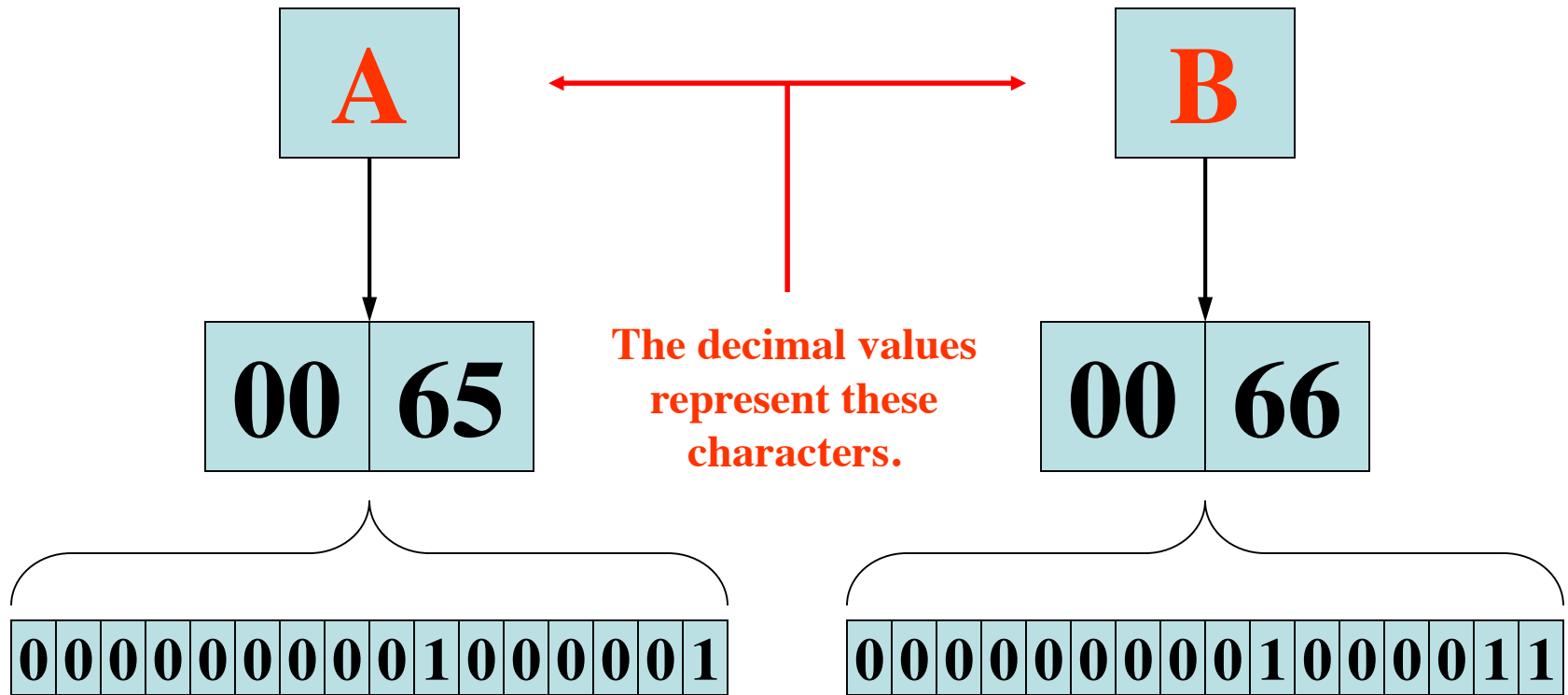
Unicode



Unicode



Unicode



Variable Assignment and Initialization

- In order to store a value in a variable, an *assignment statement* must be used.
- The *assignment operator* is the equal (=) sign.
- The operand on the left side of the assignment operator must be a variable name.
- The operand on the right side must be either a literal or expression that evaluates to a type that is compatible with the type of the variable.

Variable Assignment and Initialization

```
// This program shows variable assignment.

public class Initialize
{
    public static void main(String[] args)
    {
        int month, days;

        month = 2;
        days = 28;
        System.out.println("Month " + month + " has " +
                           days + " Days.");
    }
}
```

The variables must be declared before they can be used.

Variable Assignment and Initialization

```
// This program shows variable assignment.

public class Initialize
{
    public static void main(String[] args)
    {
        int month, days;

        month = 2;
        days = 28;
        System.out.println("Month " + month + " has " +
                           days + " Days.");
    }
}
```

Once declared, they can then receive a value (initialization); however the value must be compatible with the variable's declared type.

Variable Assignment and Initialization

```
// This program shows variable assignment.

public class Initialize
{
    public static void main(String[] args)
    {
        int month, days;

        month = 2;
        days = 28;
        System.out.println("Month " + month + " has " +
                           days + " Days.");
    }
}
```

After receiving a value, the variables can then be used in output statements or in other calculations.

Variable Assignment and Initialization

```
// This program shows variable initialization.

public class Initialize
{
    public static void main(String[] args)
    {
        int month = 2, days = 28;
        System.out.println("Month " + month + " has " +
                           days + " Days.");
    }
}
```

Local variables can be declared and initialized on the same line.

Variable Assignment and Initialization

- Variables can only hold one value at a time.
- Local variables do not receive a default value.
- Local variables must have a valid type in order to be used.

```
public static void main(String [] args)
{
    int month, days; //No value given...
    System.out.println("Month " + month + " has " +
                       days + " Days.");
}
```

Trying to use uninitialized variables will generate a Syntax Error when the code is compiled.

Arithmetic Operators

- Java has five (5) arithmetic operators.

Operator	Meaning	Type	Example
+	Addition	Binary	<code>total = cost + tax;</code>
-	Subtraction	Binary	<code>cost = total - tax;</code>
*	Multiplication	Binary	<code>tax = cost * rate;</code>
/	Division	Binary	<code>salePrice = original / 2;</code>
%	Modulus	Binary	<code>remainder = value % 5;</code>

Arithmetic Operators

- The operators are called binary operators because they must have two operands.
- Each operator must have a left and right operator.

See example: [Wages.java](#)

- The arithmetic operators work as one would expect.
- It is an error to try to divide any number by zero.
- When working with two integer operands, the division operator requires special attention.

Integer Division

- Division can be tricky.
In a Java program, what is the value of $1/2$?
- You might think the answer is 0.5...
- But, that's wrong.
- The answer is simply 0.
- Integer division will truncate any decimal remainder.

Operator Precedence

- Mathematical expressions can be very complex.
- There is a set order in which arithmetic operations will be carried out.

	Operator	Associativity	Example	Result
Higher Priority	- (unary negation)	Right to left	$x = -4 + 3;$	-1
	* / %	Left to right	$x = -4 + 4 \% 3 * 13 + 2;$	11
Lower Priority	+ -	Left to right	$x = 6 + 3 - 4 + 6 * 3;$	23

Grouping with Parenthesis

- When parenthesis are used in an expression, the inner most parenthesis are processed first.
- If two sets of parenthesis are at the same level, they are processed left to right.

•
$$x = ((4 * 5) / (5 - 2)) - 25; \quad // \text{ result} = -19$$

The diagram illustrates the order of operations for the expression $x = ((4 * 5) / (5 - 2)) - 25$. Red curly braces and numbers indicate the sequence of operations:

- 1**: The innermost parentheses $(4 * 5)$ are processed first.
- 2**: The innermost parentheses $(5 - 2)$ are processed second.
- 3**: The division operation $/$ is processed third, combining the results of steps 1 and 2.
- 4**: The final subtraction operation $- 25$ is processed last, combining the result of step 3 with 25.

Combined Assignment Operators

- Java has some combined assignment operators.
- These operators allow the programmer to perform an arithmetic operation and assignment with a single operator.
- Although not required, these operators are popular since they shorten simple equations.

Combined Assignment Operators

Operator	Example	Equivalent	Value of variable after operation
+=	<code>x += 5;</code>	<code>x = x + 5;</code>	The old value of x plus 5.
-=	<code>y -= 2;</code>	<code>y = y - 2;</code>	The old value of y minus 2
*=	<code>z *= 10;</code>	<code>z = z * 10;</code>	The old value of z times 10
/=	<code>a /= b;</code>	<code>a = a / b;</code>	The old value of a divided by b .
%=	<code>c %= 3;</code>	<code>c = c % 3;</code>	The remainder of the division of the old value of c divided by 3.

Creating Constants

- Many programs have data that does not need to be changed.
- Littering programs with literal values can make the program hard to read and maintain.
- Replacing literal values with constants remedies this problem.
- Constants allow the programmer to use a name rather than a value throughout the program.
- Constants also give a singular point for changing those values when needed.

Creating Constants

- Constants keep the program organized and easier to maintain.
- Constants are identifiers that can hold only a single value.
- Constants are declared using the keyword `final`.
- Constants need not be initialized when declared; however, they must be initialized before they are used or a compiler error will be generated.

Creating Constants

- Once initialized with a value, constants cannot be changed programmatically.
- By convention, constants are all upper case and words are separated by the underscore character.

```
final int CAL_SALES_TAX = 0.725;
```

The String Class

- Java has no primitive data type that holds a series of characters.
- The `String` class from the Java standard library is used for this purpose.
- In order to be useful, the a variable must be created to reference a `String` object.
`String number;`
- Notice the `S` in `String` is upper case.
- By convention, class names should always begin with an upper case character.

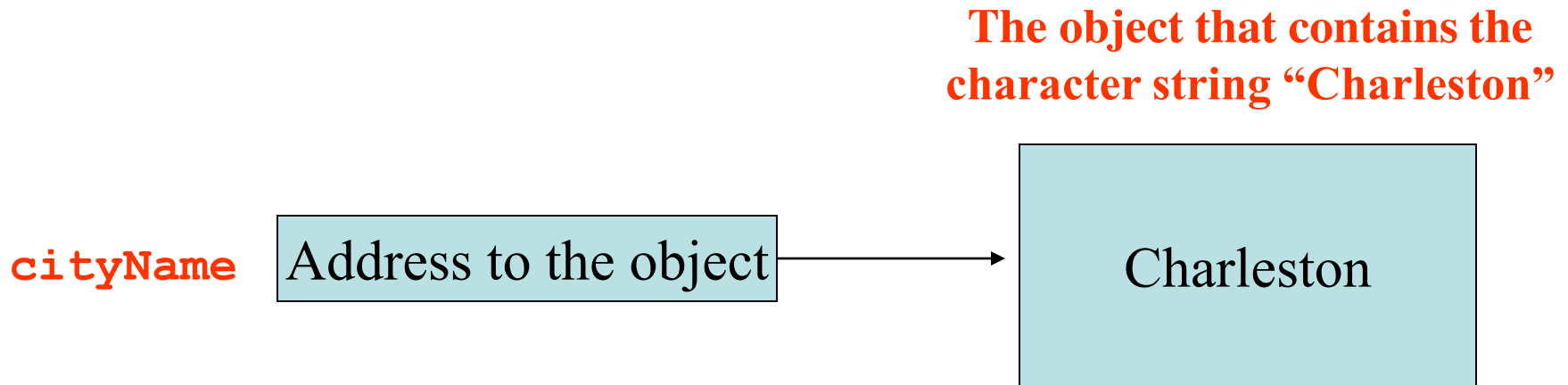
Primitive vs. Reference Variables

- Primitive variables actually contain the value that they have been assigned.
`number = 25;`
- The value 25 will be stored in the memory location associated with the variable `number`.
- Objects are not stored in variables, however. Objects are *referenced* by variables.

Primitive vs. Reference Variables

- When a variable references an object, it contains the memory address of the object's location.
- Then it is said that the variable *references* the object.

```
String cityName = "Charleston";
```



String Objects

- A variable can be assigned a String literal.

```
String value = "Hello";
```

- Strings are the only objects that can be created in this way.
 - A variable can be created using the *new* keyword.
- ```
String value = new String("Hello");
```
- This is the method that all other objects must use when they are created.

See example: [StringDemo.java](#)

# The String Methods

- Since `String` is a class, objects that are instances of it have methods.
- One of those methods is the `length` method.  

```
stringSize = value.length();
```
- This statement runs the `length` method on the object pointed to by the `value` variable.

See example: [StringLength.java](#)

# String Methods

- The `String` class contains many methods that help with the manipulation of `String` objects.
- `String` objects are *immutable*, meaning that they cannot be changed.
- Many of the methods of a `String` object can create new versions of the object.

See example: [StringMethods.java](#)

# Scope

- *Scope* refers to the part of a program that has access to a variable's contents.
- Variables declared inside a method (like the main method) are called *local variables*.
- Local variables' scope begins at the declaration of the variable and ends at the end of the method in which it was declared.

See example: [Scope.java](#) (This program contains an intentional error.)



# Commenting Code

- Java provides three methods for commenting code.

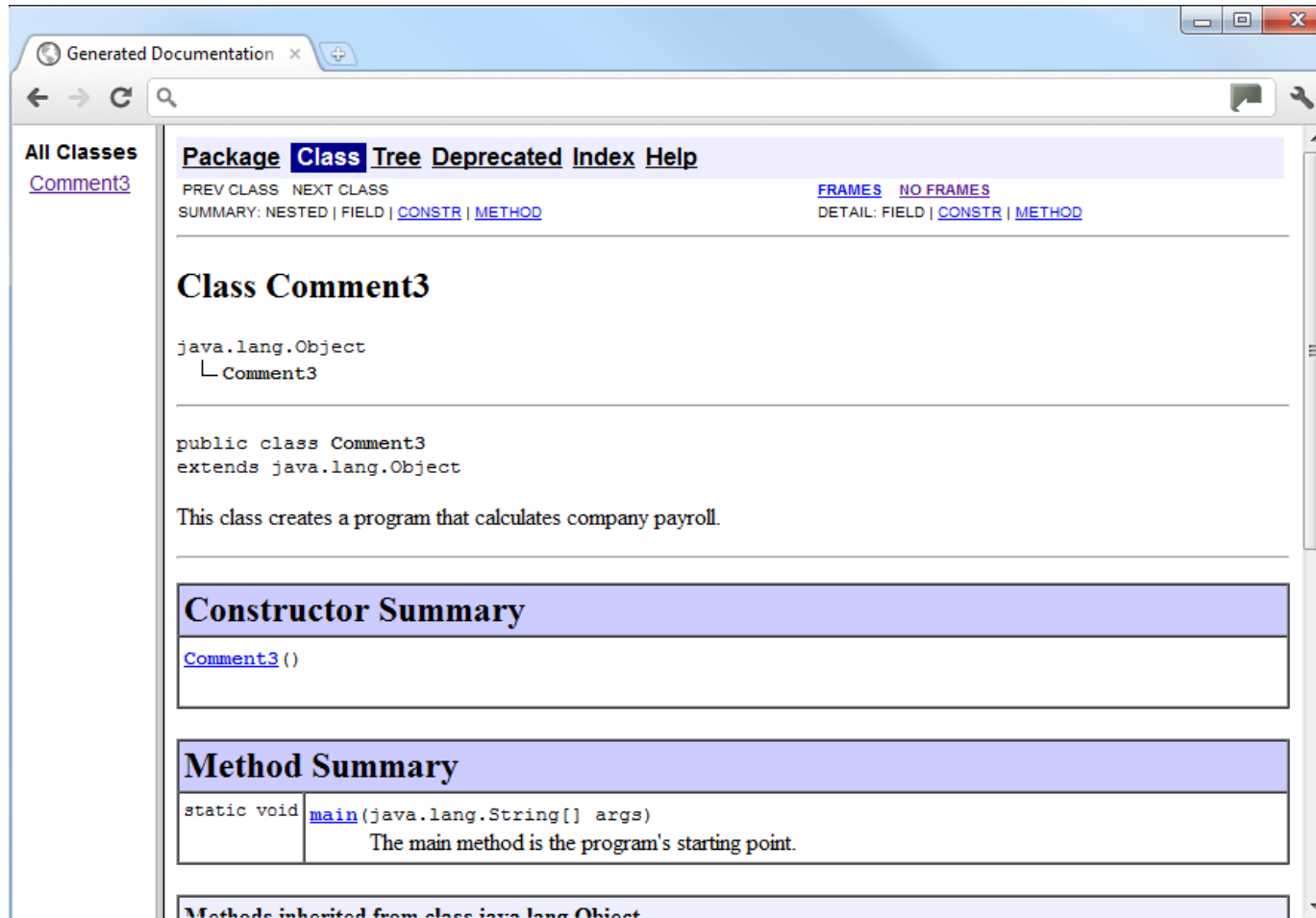
| Comment Style | Description                                                                                                                                                                                                                                                                        |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| //            | Single line comment. Anything after the // on the line will be ignored by the compiler.                                                                                                                                                                                            |
| /* ... */     | Block comment. Everything beginning with /* and ending with the first */ will be ignored by the compiler. This comment type cannot be nested.                                                                                                                                      |
| /** ... */    | Javadoc comment. This is a special version of the previous block comment that allows comments to be documented by the javadoc utility program. Everything beginning with the /** and ending with the first */ will be ignored by the compiler. This comment type cannot be nested. |

# Commenting Code

- Javadoc comments can be built into HTML documentation.
- See example: [Comment3.java](#)
- To create the documentation:
  - Run the `javadoc` program with the source file as an argument
  - Ex: **`javadoc Comment3.java`**
- The `javadoc` program will create `index.html` and several other documentation files in the same directory as the input file.

# Commenting Code

- Example [index.html](#):



# Programming Style

- Although Java has a strict syntax, whitespace characters are ignored by the compiler.
- The Java whitespace characters are:
  - space
  - tab
  - newline
  - carriage return
  - form feed

See example: [Compact.java](#)

# Indentation

- Programs should use proper indentation.
- Each block of code should be indented a few spaces from its surrounding block.
- Two to four spaces are sufficient.
- Tab characters should be avoided.
  - Tabs can vary in size between applications and devices.
  - Most programming text editors allow the user to replace the tab with spaces.

See example: [Readable.java](#)

# The Scanner Class

- To read input from the keyboard we can use the `Scanner` class.
- The `Scanner` class is defined in `java.util`, so we will use the following statement at the top of our programs:

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

# The Scanner Class

- Scanner objects work with `System.in`
- To create a Scanner object:  
`Scanner keyboard = new Scanner (System.in);`
- Scanner class methods are listed in Table 2-18 in the text.
- See example: [Payroll.java](#)

# Dialog Boxes

- A *dialog box* is a small graphical window that displays a message to the user or requests input.
- A variety of dialog boxes can be displayed using the `JOptionPane` class.
- Two of the dialog boxes are:
  - Message Dialog - a dialog box that displays a message.
  - Input Dialog - a dialog box that prompts the user for input.



# The JOptionPane Class

- The JOptionPane class is not automatically available to your Java programs.
- The following statement must be before the program's class header:  

```
import javax.swing.JOptionPane;
```
- This statement tells the compiler where to find the JOptionPane class.

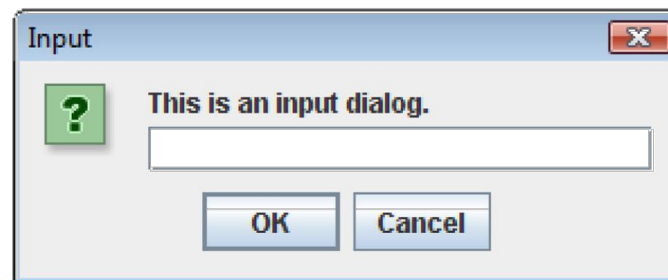
# The JOptionPane Class

The `JOptionPane` class provides methods to display each type of dialog box.

Message dialog



Input dialog

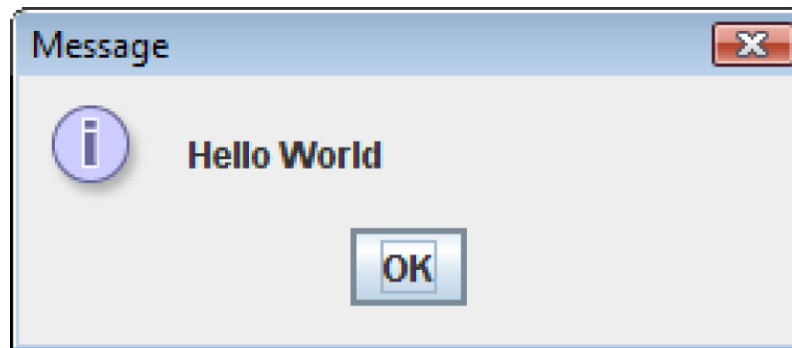


# Message Dialogs

- `JOptionPane.showMessageDialog` method is used to display a message dialog.

```
JOptionPane.showMessageDialog(null, "Hello World");
```

- The first argument will be discussed in Chapter 7.
- The second argument is the message that is to be displayed.



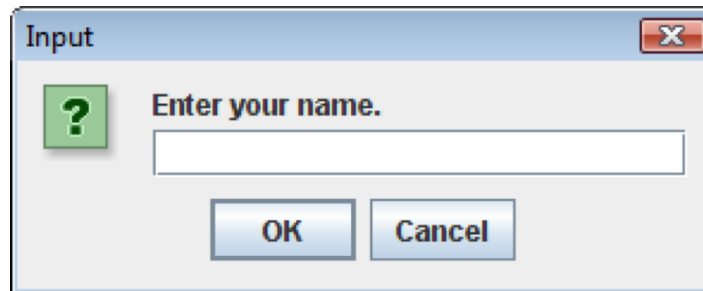
# Input Dialogs

- An input dialog is a quick and simple way to ask the user to enter data.
- The dialog displays a text field, an Ok button and a Cancel button.
- If Ok is pressed, the dialog returns the user's input.
- If Cancel is pressed, the dialog returns null.

# Input Dialogs

```
String name;
name = JOptionPane.showInputDialog(
 "Enter your name.");
```

- The argument passed to the method is the message to display.
- If the user clicks on the OK button, `name` references the string entered by the user.
- If the user clicks on the Cancel button, `name` references `null`.



# The `System.exit` Method

- A program that uses `JOptionPane` does not automatically stop executing when the end of the main method is reached.
- Java generates a *thread*, which is a process running in the computer, when a `JOptionPane` is created.
- If the `System.exit` method is not called, this thread continues to execute.

# The `System.exit` Method

- The `System.exit` method requires an integer argument.  
`System.exit(0);`
- This argument is an *exit code* that is passed back to the operating system.
- This code is usually ignored, however, it can be used outside the program:
  - to indicate whether the program ended successfully or as the result of a failure.
  - The value 0 traditionally indicates that the program ended successfully.

# Converting a String to a Number

- The `JOptionPane`'s `showInputDialog` method always returns the user's input as a `String`
- A `String` containing a number, such as “127.89”, can be converted to a numeric data type.



# The Parse Methods

- Each of the numeric wrapper classes, (covered in Chapter 10) has a method that converts a string to a number.
  - The `Integer` class has a method that converts a string to an `int`,
  - The `Double` class has a method that converts a string to a `double`, and
  - etc.
- These methods are known as *parse methods* because their names begin with the word “parse.”

# The Parse Methods

```
// Store 1 in bVar.
byte bVar = Byte.parseByte("1");

// Store 2599 in iVar.
int iVar = Integer.parseInt("2599");

// Store 10 in sVar.
short sVar = Short.parseShort("10");

// Store 15908 in lVar.
long lVar = Long.parseLong("15908");

// Store 12.3 in fVar.
float fVar = Float.parseFloat("12.3");

// Store 7945.6 in dVar.
double dVar = Double.parseDouble("7945.6");
```

# Reading an Integer with an Input Dialog

```
int number;
String str;
str = JOptionPane.showInputDialog(
 "Enter a number.");
number = Integer.parseInt(str);
```

# Reading a double with an Input Dialog

```
double price;
String str;
str = JOptionPane.showInputDialog(
 "Enter the retail price.");
price = Double.parseDouble(str);
```

See example: [PayrollDialog.java](#)

# CHAPTER 3

## Decision Structures

starting out with >>>

# JAVA™

From Control Structures through Objects

6TH EDITION



TONY GADDIS

# Chapter Topics

Chapter 3 discusses the following main topics:

- The `if` Statement
- The `if-else` Statement
- Nested `if` statements
- The `if-else-if` Statement
- Logical Operators
- Comparing `String` Objects

# Chapter Topics

Chapter 3 discusses the following main topics:

- More about Variable Declaration and Scope
- The Conditional Operator
- The `switch` Statement
- Displaying Formatted Output with  
`System.out.printf` and `String.format`

# The `if` Statement

- The `if` statement decides whether a section of code executes or not.
- The `if` statement uses a `boolean` to decide whether the next statement or block of statements executes.

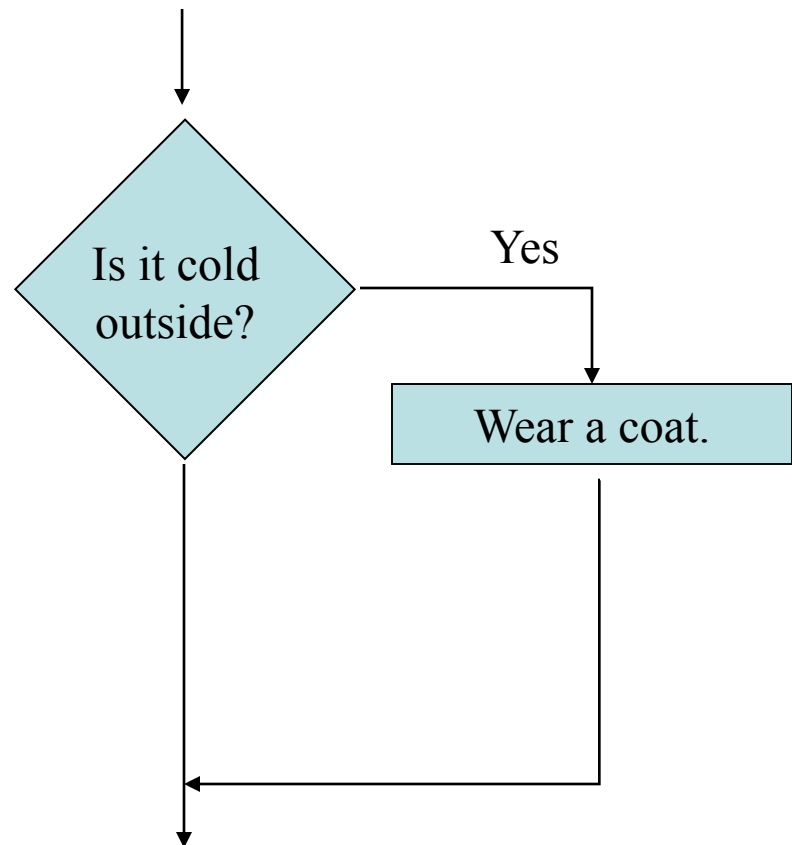
*if (boolean expression is true)  
    execute next statement.*



# Flowcharts

- If statements can be modeled as a flow chart.

```
if (coldOutside)
 wearCoat();
```

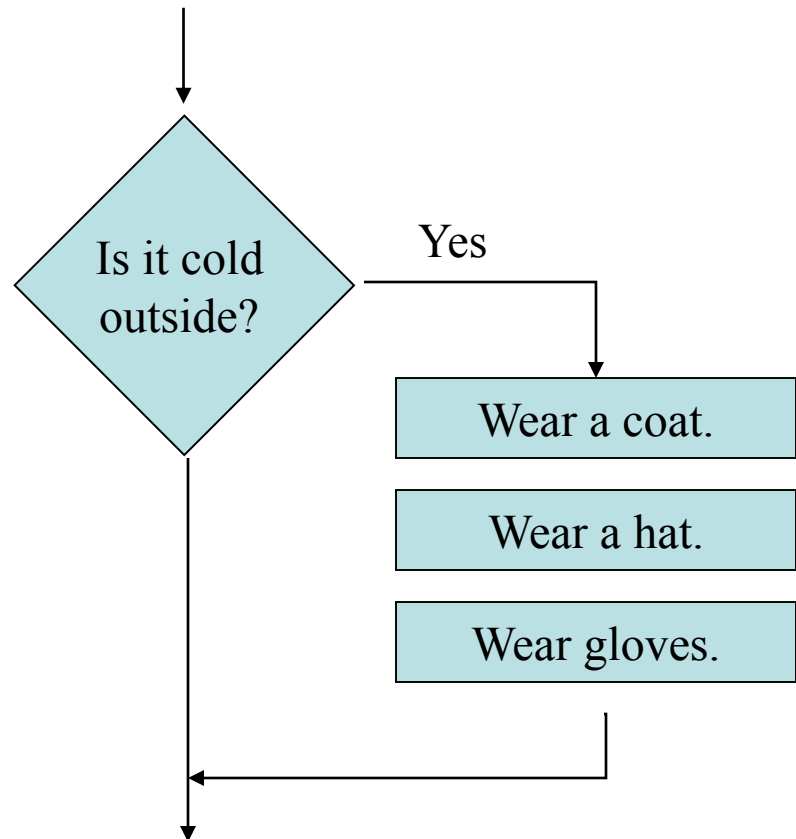


# Flowcharts

- A block `if` statement may be modeled as:

```
if (coldOutside)
{
 wearCoat();
 wearHat();
 wearGloves();
}
```

**Note the use of curly braces to block several statements together.**



# Relational Operators

- In most cases, the `boolean` expression, used by the `if` statement, uses *relational operators*.

| Relational Operator | Meaning                     |
|---------------------|-----------------------------|
| >                   | is greater than             |
| <                   | is less than                |
| >=                  | is greater than or equal to |
| <=                  | is less than or equal to    |
| ==                  | is equal to                 |
| !=                  | is not equal to             |

# Boolean Expressions

- A *boolean expression* is any variable or calculation that results in a *true* or *false* condition.

| Expression       | Meaning                          |
|------------------|----------------------------------|
| <b>x &gt; y</b>  | Is x greater than y?             |
| <b>x &lt; y</b>  | Is x less than y?                |
| <b>x &gt;= y</b> | Is x greater than or equal to y? |
| <b>x &lt;= y</b> | Is x less than or equal to y.    |
| <b>x == y</b>    | Is x equal to y?                 |
| <b>x != y</b>    | Is x not equal to y?             |

# if Statements and Boolean Expressions

```
if (x > y)
 System.out.println("X is greater than Y");
```

```
if (x == y)
 System.out.println("X is equal to Y");
```

```
if (x != y)
{
 System.out.println("X is not equal to Y");
 x = y;
 System.out.println("However, now it is.");
}
```

Example: [AverageScore.java](#)

# Programming Style and `if` Statements

- An `if` statement can span more than one line; however, it is still one statement.

```
if (average > 95)
 grade = 'A';
```

is functionally equivalent to

```
if (average > 95) grade = 'A';
```

# Programming Style and `if` Statements

- Rules of thumb:
  - The conditionally executed statement should be on the line after the `if` condition.
  - The conditionally executed statement should be indented one level from the `if` condition.
  - If an `if` statement does not have the block curly braces, it is ended by the first semicolon encountered after the `if` condition.

```
if (expression) ← No semicolon here.
 statement; ← Semicolon ends statement here.
```

# Block `if` Statements

- Conditionally executed statements can be grouped into a block by using curly braces `{ }` to enclose them.
- If curly braces are used to group conditionally executed statements, the `if` statement is ended by the closing curly brace.

```
if (expression)
```

```
{
```

```
 statement1;
```

```
 statement2;
```

```
}
```

← **Curly brace ends the statement.**



# Block `if` Statements

- Remember that when the curly braces are not used, then only the next statement after the `if` condition will be executed conditionally.

```
if (expression)
 statement1;
 statement2;
 statement3;
```

← **Only this statement is conditionally executed.**

# Flags

- A flag is a `boolean` variable that monitors some condition in a program.
- When a condition is true, the flag is set to `true`.
- The flag can be tested to see if the condition has changed.

```
if (average > 95)
 highScore = true;
```

- Later, this condition can be tested:

```
if (highScore)
 System.out.println("That's a high score!");
```

# Comparing Characters

- Characters can be tested with relational operators.
- Characters are stored in memory using the Unicode character format.
- Unicode is stored as a sixteen (16) bit number.
- Characters are *ordinal*, meaning they have an order in the Unicode character set.
- Since characters are ordinal, they can be compared to each other.

```
char c = 'A';
if (c < 'Z')
 System.out.println("A is less than Z");
```

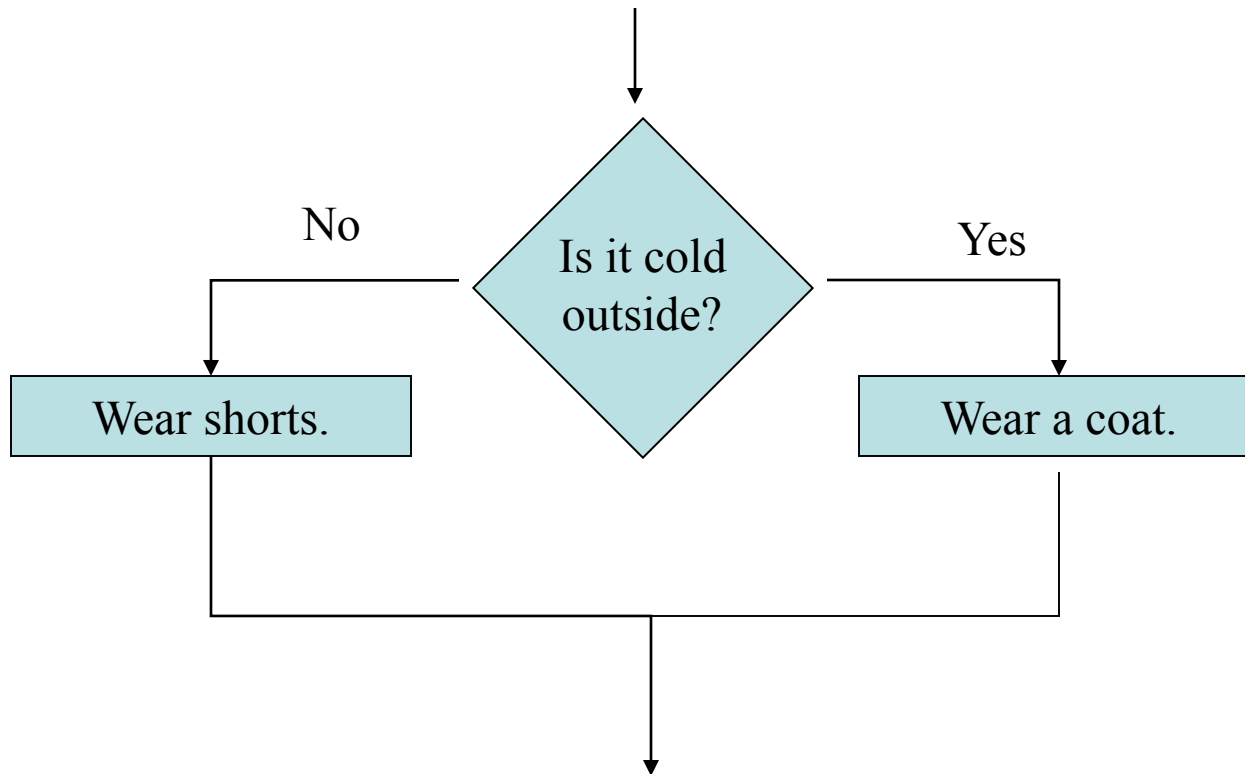
# if-else Statements

- The `if-else` statement adds the ability to conditionally execute code when the `if` condition is false.

```
if (expression)
 statementOrBlockIfTrue;
else
 statementOrBlockIfFalse;
```

- See example: [Division.java](#)

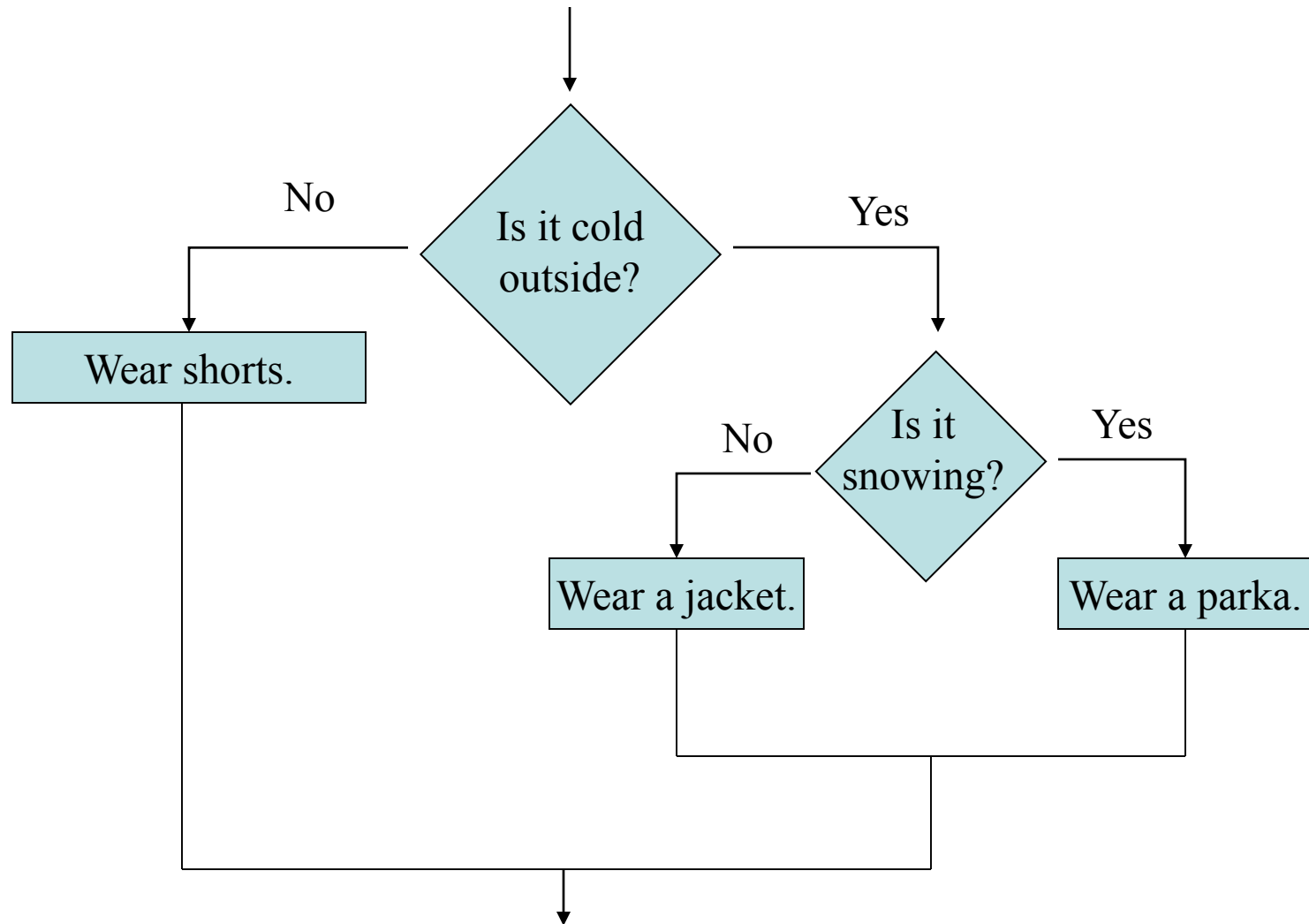
# if-else Statement Flowcharts



# Nested `if` Statements

- If an `if` statement appears inside another `if` statement (single or block) it is called a *nested if* statement.
- The nested `if` is executed only if the outer `if` statement results in a true condition.
- See example: [LoanQualifier.java](#)

# Nested `if` Statement Flowcharts



# Nested `if` Statements

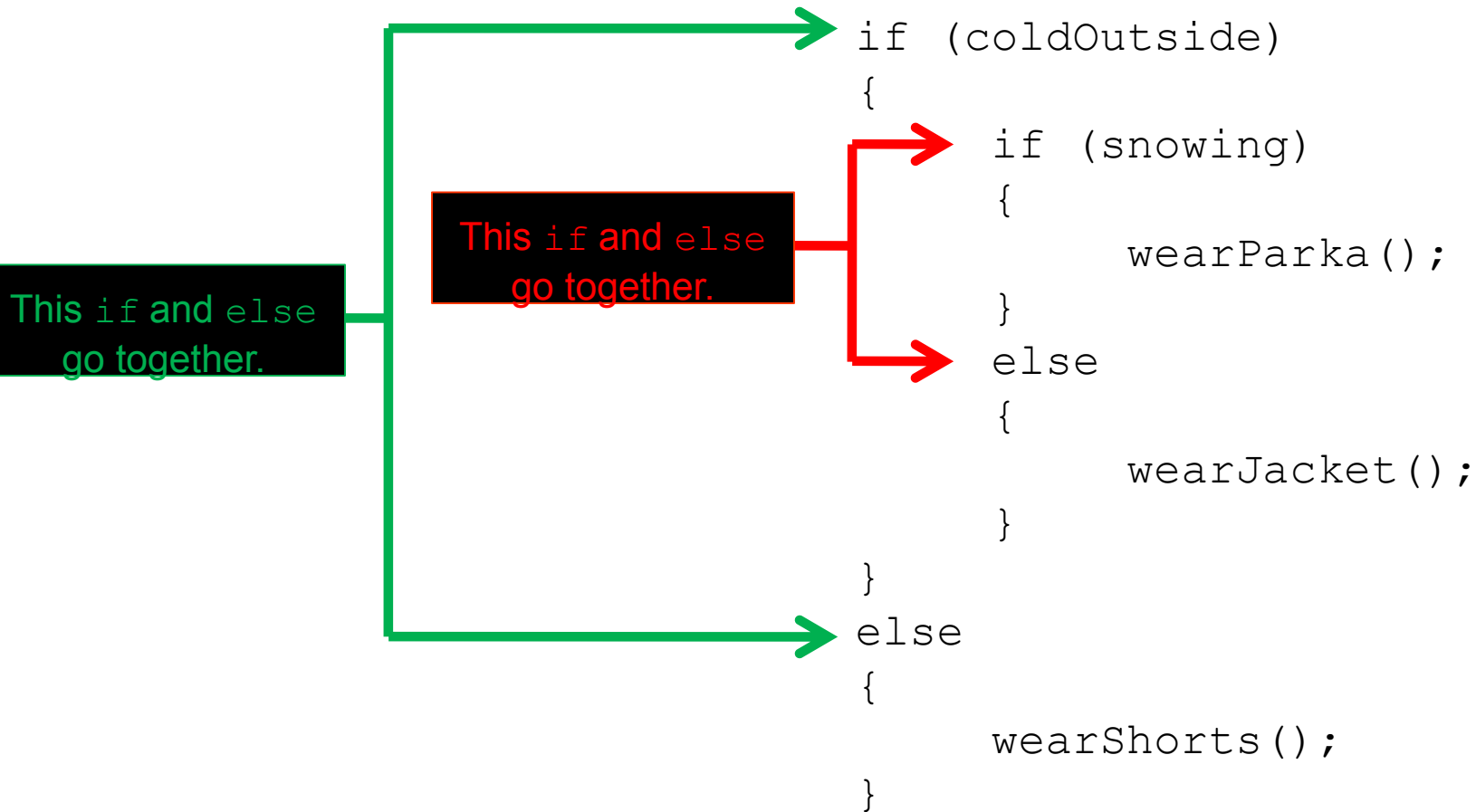
```
if (coldOutside)
{
 if (snowing)
 {
 wearParka();
 }
 else
 {
 wearJacket();
 }
}
else
{
 wearShorts();
}
```



## `if-else` Matching

- Curly brace use is not required if there is only one statement to be conditionally executed.
- However, sometimes curly braces can help make the program more readable.
- Additionally, proper indentation makes it much easier to match up else statements with their corresponding `if` statement.

# Alignment and Nested `if` Statements



# if-else-if Statements

```
if (expression_1)
{
 statement;
 statement;
 etc.
}
else if (expression_2)
{
 statement;
 statement;
 etc.
}
```

*If expression\_1 is true these statements are executed, and the rest of the structure is ignored.*

*Otherwise, if expression\_2 is true these statements are executed, and the rest of the structure is ignored.*

***Insert as many else if clauses as necessary***

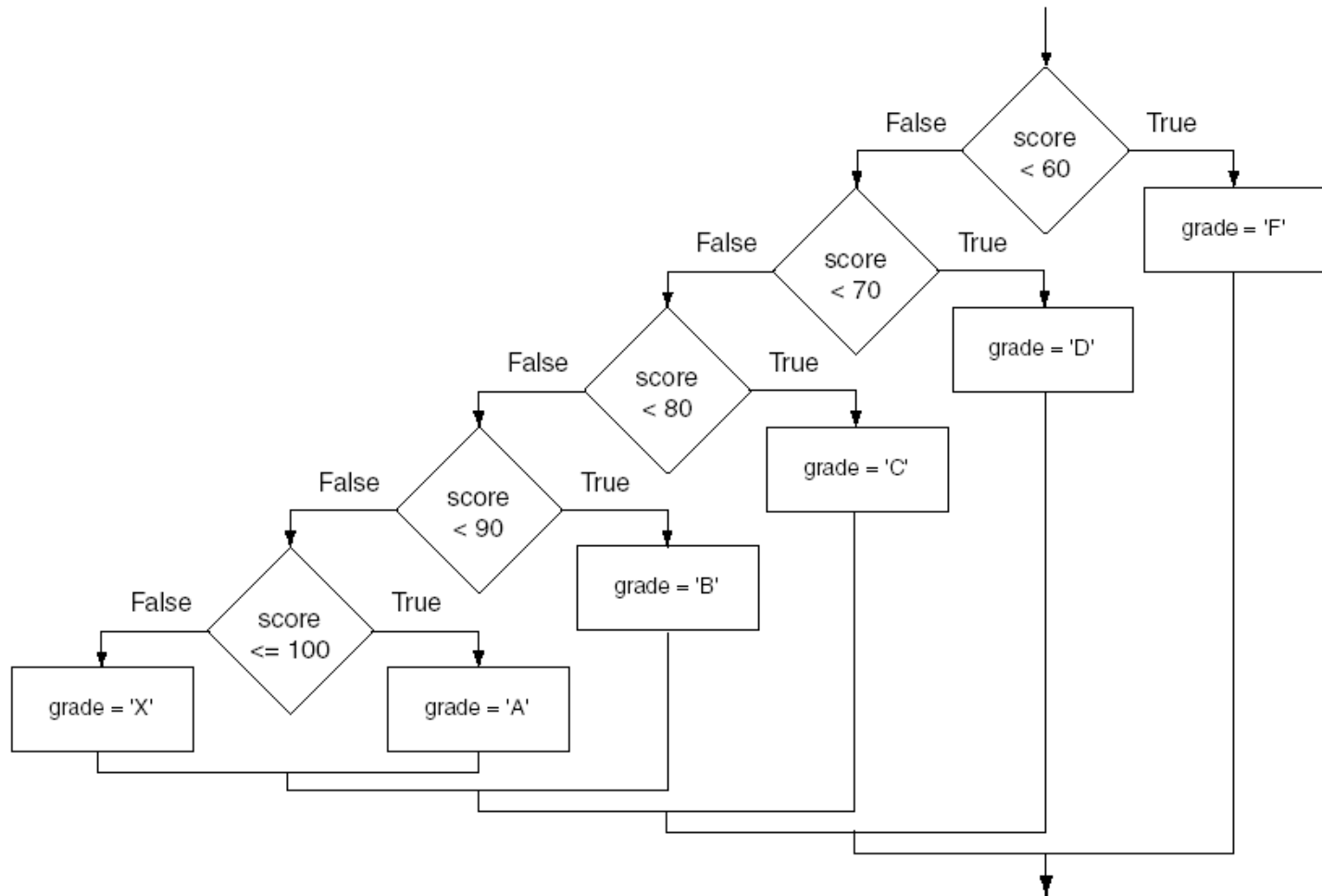
```
else
{
 statement;
 statement;
 etc.
}
```

*These statements are executed if none of the expressions above are true.*

# `if-else-if` Statements

- Nested `if` statements can become very complex.
- The `if-else-if` statement makes certain types of nested decision logic simpler to write.
- Care must be used since `else` statements match up with the immediately preceding unmatched `if` statement.
- See example: [TestResults.java](#)

# if-else-if Flowchart



# Logical Operators

- Java provides two binary *logical operators* (`&&` and `||`) that are used to combine `boolean` expressions.
- Java also provides one *unary* (`!`) logical operator to reverse the truth of a `boolean` expression.

# Logical Operators

| Operator          | Meaning    | Effect                                                                                                                                                                                                                                 |
|-------------------|------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>&amp;&amp;</b> | <b>AND</b> | Connects two <code>boolean</code> expressions into one. Both expressions must be true for the overall expression to be true.                                                                                                           |
| <b>  </b>         | <b>OR</b>  | Connects two <code>boolean</code> expressions into one. One or both expressions must be true for the overall expression to be true. It is only necessary for one to be true, and it does not matter which one.                         |
| <b>!</b>          | <b>NOT</b> | The <b>!</b> operator reverses the truth of a <code>boolean</code> expression. If it is applied to an expression that is true, the operator returns false. If it is applied to an expression that is false, the operator returns true. |

# The && Operator

- The logical AND operator (&&) takes two operands that must both be `boolean` expressions.
- The resulting combined expression is true if (and *only* if) both operands are true.
- See example: [LogicalAnd.java](#)

| Expression 1 | Expression 2 | Expression1 && Expression2 |
|--------------|--------------|----------------------------|
| true         | false        | false                      |
| false        | true         | false                      |
| false        | false        | false                      |
| true         | true         | true                       |



# The || Operator

- The logical OR operator ( || ) takes two operands that must both be `boolean` expressions.
- The resulting combined expression is false if (and *only* if) both operands are false.
- Example: [LogicalOr.java](#)

| Expression 1 | Expression 2 | Expression1    Expression2 |
|--------------|--------------|----------------------------|
| true         | false        | true                       |
| false        | true         | true                       |
| false        | false        | false                      |
| true         | true         | true                       |

# The ! Operator

- The ! operator performs a logical NOT operation.
- If an *expression* is true, *!expression* will be false.

```
if (!(temperature > 100))
 System.out.println("Below the maximum temperature.");
```

- If `temperature > 100` evaluates to false, then the output statement will be run.

| Expression 1 | !Expression1 |
|--------------|--------------|
| true         | false        |
| false        | true         |

# Short Circuiting

- Logical AND and logical OR operations perform *short-circuit evaluation* of expressions.
- Logical AND will evaluate to false as soon as it sees that one of its operands is a false expression.
- Logical OR will evaluate to true as soon as it sees that one of its operands is a true expression.

# Order of Precedence

- The `!` operator has a higher order of precedence than the `&&` and `||` operators.
- The `&&` and `||` operators have a lower precedence than relational operators like `<` and `>`.
- Parenthesis can be used to force the precedence to be changed.

# Order of Precedence

| Order of Precedence | Operators           | Description                                                              |
|---------------------|---------------------|--------------------------------------------------------------------------|
| 1                   | (unary negation) !  | Unary negation, logical NOT                                              |
| 2                   | * / %               | Multiplication, Division, Modulus                                        |
| 3                   | + -                 | Addition, Subtraction                                                    |
| 4                   | < > <= >=           | Less-than, Greater-than, Less-than or equal to, Greater-than or equal to |
| 5                   | == !=               | Is equal to, Is not equal to                                             |
| 6                   | &&                  | Logical AND                                                              |
| 7                   |                     | Logical OR                                                               |
| 8                   | = += -=<br>*= /= %= | Assignment and combined assignment operators.                            |

# Comparing `String` Objects

- In most cases, you cannot use the relational operators to compare two `String` objects.
- Reference variables contain the address of the object they represent.
- Unless the references point to the same object, the relational operators will not return true.
- See example: [StringCompare.java](#)
- See example: [StringCompareTo.java](#)

# Ignoring Case in String Comparisons

- In the `String` class the `equals` and `compareTo` methods are case sensitive.
- In order to compare two `String` objects that might have different case, use:
  - `equalsIgnoreCase`, or
  - `compareToIgnoreCase`
- See example: [SecretWord.java](#)

# Variable Scope

- In Java, a local variable does not have to be declared at the beginning of the method.
- The scope of a local variable begins at the point it is declared and terminates at the end of the method.
- When a program enters a section of code where a variable has scope, that variable has *come into scope*, which means the variable is visible to the program.
- See example: [VariableScope.java](#)



# The Conditional Operator

- The *conditional operator* is a ternary (three operand) operator.
- You can use the conditional operator to write a simple statement that works like an `if-else` statement.

# The Conditional Operator

- The format of the operators is:

*BooleanExpression ? Value1 : Value2*

- This forms a conditional expression.
- If *BooleanExpression* is true, the value of the conditional expression is *Value1*.
- If *BooleanExpression* is false, the value of the conditional expression is *Value2*.

# The Conditional Operator

- Example:

```
z = x > y ? 10 : 5;
```

- This line is functionally equivalent to:

```
if (x > y)
```

```
 z = 10;
```

```
else
```

```
 z = 5;
```

# The Conditional Operator

- Many times the conditional operator is used to supply a value.

```
number = x > y ? 10 : 5;
```

- This is functionally equivalent to:

```
if(x > y)
 number = 10;
else
 number = 5;
```

- See example: [ConsultantCharges.java](#)

# The `switch` Statement

- The `if-else` statement allows you to make true / false branches.
- The `switch` statement allows you to use an ordinal value to determine how a program will branch.
- The `switch` statement can evaluate an *integer* type or *character* type variable and make decisions based on the value.

# The `switch` Statement

- The `switch` statement takes the form:

```
switch (SwitchExpression)
{
 case CaseExpression:
 // place one or more statements here
 break;
 case CaseExpression:
 // place one or more statements here
 break;

 // case statements may be repeated
 //as many times as necessary
 default:
 // place one or more statements here
}
```

# The `switch` Statement

```
switch (SwitchExpression)
{

 ...
}
```

- The `switch` statement will evaluate the *SwitchExpression*, which can be a `byte`, `short`, `int`, `long`, or `char`. If you are using Java 7, the *SwitchExpression* can also be a string.
- If there is an associated `case` statement that matches that value, program execution will be transferred to that `case` statement.

# The `switch` Statement

- Each case statement will have a corresponding *CaseExpression* that must be unique.

```
case CaseExpression:
 // place one or more statements here
 break;
```

- If the *SwitchExpression* matches the *CaseExpression*, the Java statements between the colon and the `break` statement will be executed.



# The case Statement

- The `break` statement ends the case statement.
- The `break` statement is optional.
- If a case does not contain a `break`, then program execution continues into the next case.
  - See example: [NoBreaks.java](#)
  - See example: [PetFood.java](#)
- The `default` section is optional and will be executed if no *CaseExpression* matches the *SwitchExpression*.
- See example: [SwitchDemo.java](#)

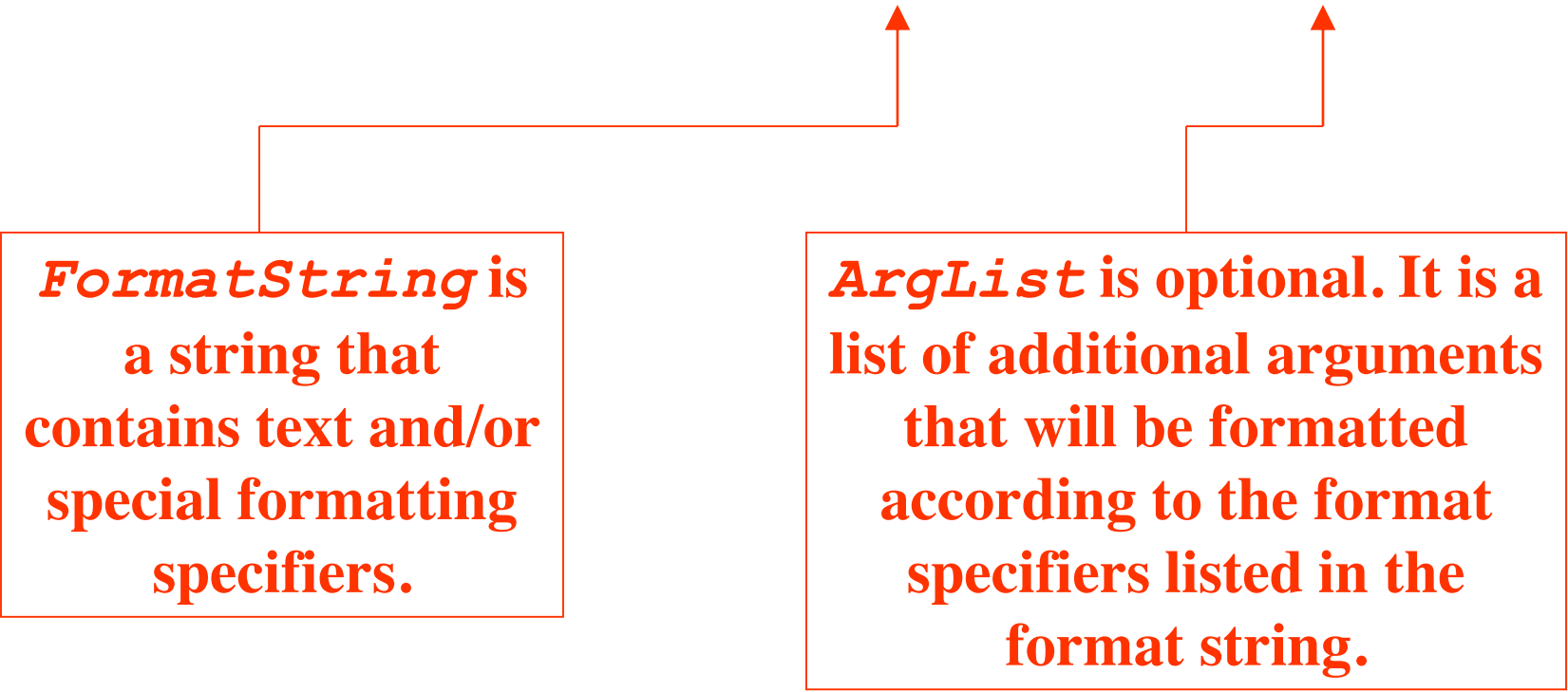
# The `System.out.printf` Method

- You can use the `System.out.printf` method to perform formatted console output.
- The general format of the method is:

```
System.out.printf(FormatString, ArgList);
```

# The `System.out.printf` Method

```
System.out.printf(FormatString, ArgList);
```



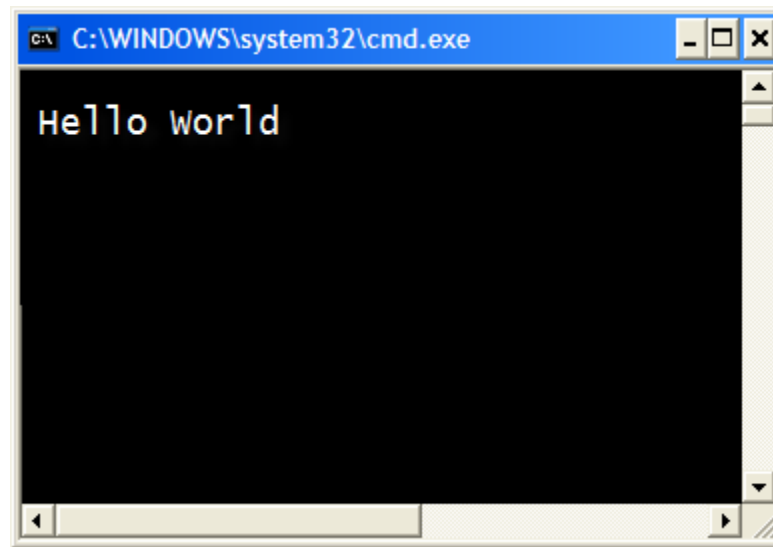
***FormatString*** is a string that contains text and/or special formatting specifiers.

***ArgList*** is optional. It is a list of additional arguments that will be formatted according to the format specifiers listed in the format string.

# The `System.out.printf` Method

- A simple example:

```
System.out.printf("Hello World\n");
```

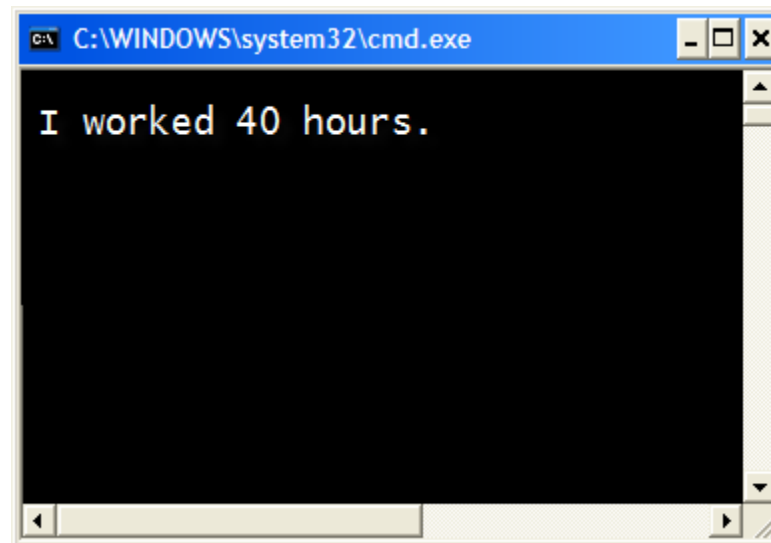


# The `System.out.printf` Method

- Another example:

```
int hours = 40;
```

```
System.out.printf("I worked %d hours.\n", hours);
```



# The `System.out.printf` Method

```
int hours = 40;
System.out.printf("I worked %d hours.\n", hours);
```

The diagram illustrates the flow of data in the `printf` statement. A red circle highlights the `%d` format specifier. A red arrow points from the `hours` variable in the first line to the `%d` specifier. Another red arrow points from the `hours` variable in the second line to the `%d` specifier. A third red arrow points from the `%d` specifier to the `hours` variable in the first line, indicating that the value of `hours` is used to replace the `%d` specifier.

**The `%d` format specifier indicates that a decimal integer will be printed.**

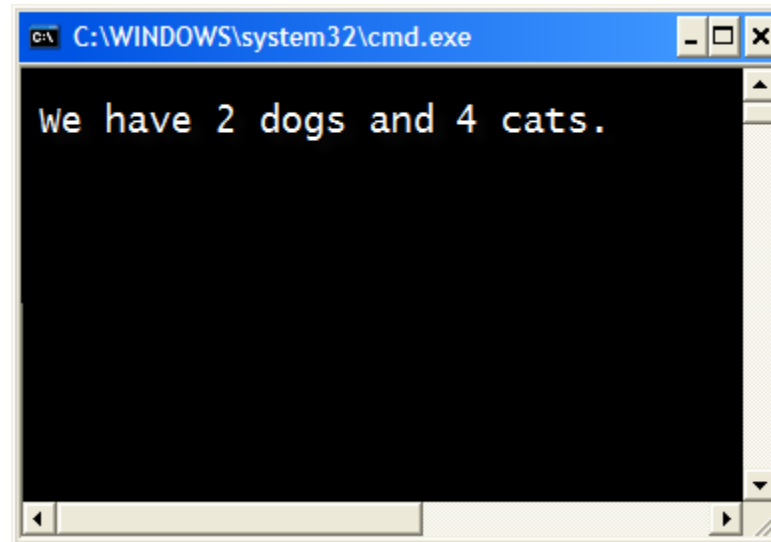
**The contents of the `hours` variable will be printed in the location of the `%d` format specifier.**

# The `System.out.printf` Method

- Another example:

```
int dogs = 2, cats = 4;
```

```
System.out.printf("We have %d dogs and %d cats.\n",
 dogs, cats);
```

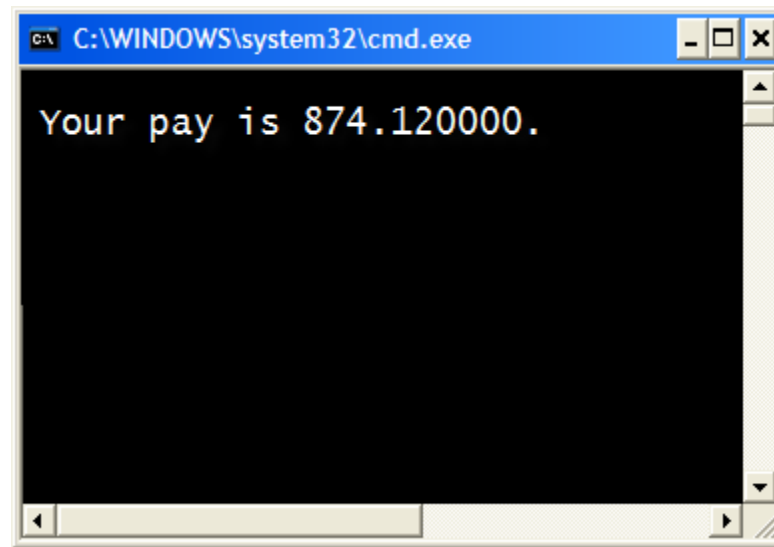


# The `System.out.printf` Method

- Another example:

```
double grossPay = 874.12;
```

```
System.out.printf("Your pay is %f.\n", grossPay);
```





# The `System.out.printf` Method

- Another example:

```
double grossPay = 874.12;
```

```
System.out.printf("Your pay is %f.\n", grossPay);
```

The **%f** format specifier indicates that a floating-point value will be printed.

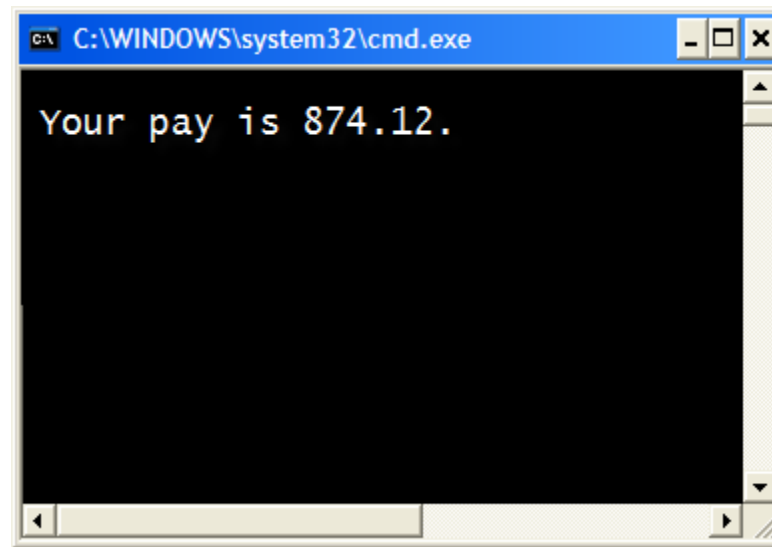
The contents of the **grossPay** variable will be printed in the location of the **%f** format specifier.

# The `System.out.printf` Method

- Another example:

```
double grossPay = 874.12;
```

```
System.out.printf("Your pay is %.2f.\n", grossPay);
```



# The `System.out.printf` Method

- Another example:

```
double grossPay = 874.12;
```

```
System.out.printf("Your pay is %.2f.\n", grossPay);
```



**The `%.2f` format specifier indicates that a floating-point value will be printed, rounded to two decimal places.**

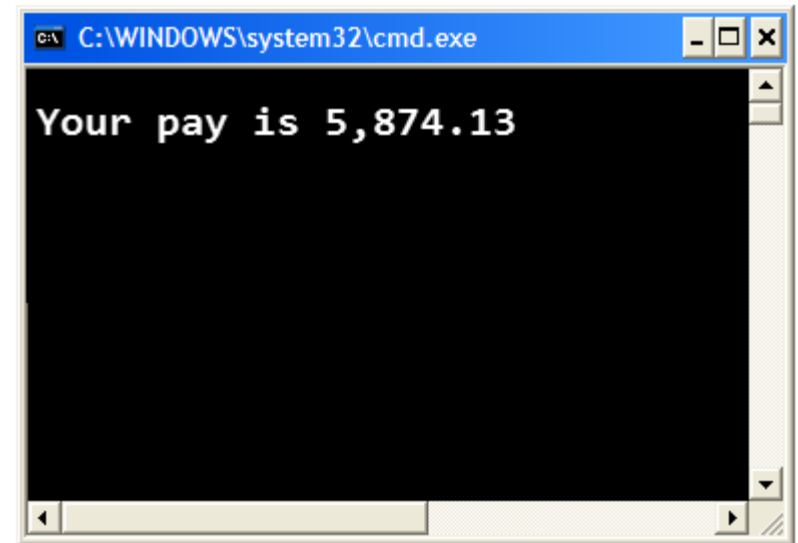
# The `System.out.printf` Method

- Another example:

```
double grossPay = 5874.127;
```

```
System.out.printf("Your pay is %, .2f.\n", grossPay);
```

**The `%, .2f` format specifier indicates that a floating-point value will be printed with comma separators, rounded to two decimal places.**

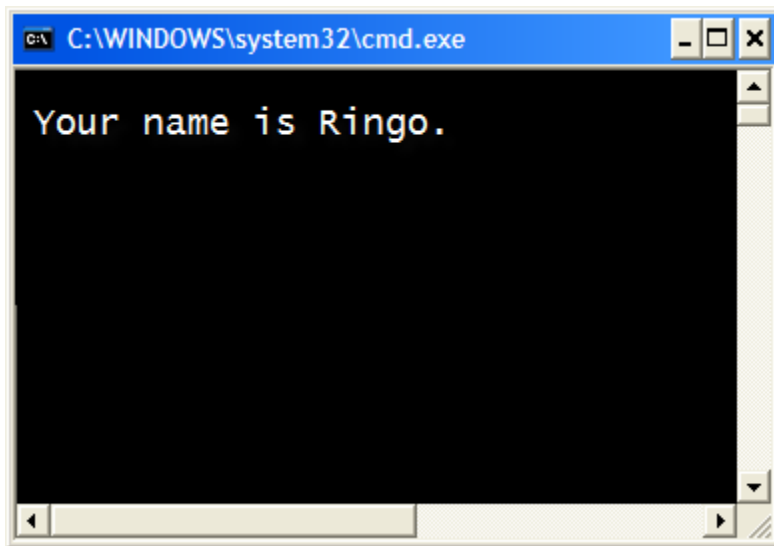


# The `System.out.printf` Method

- Another example:

```
String name = "Ringo";
```

```
System.out.printf("Your name is %s.\n", name);
```

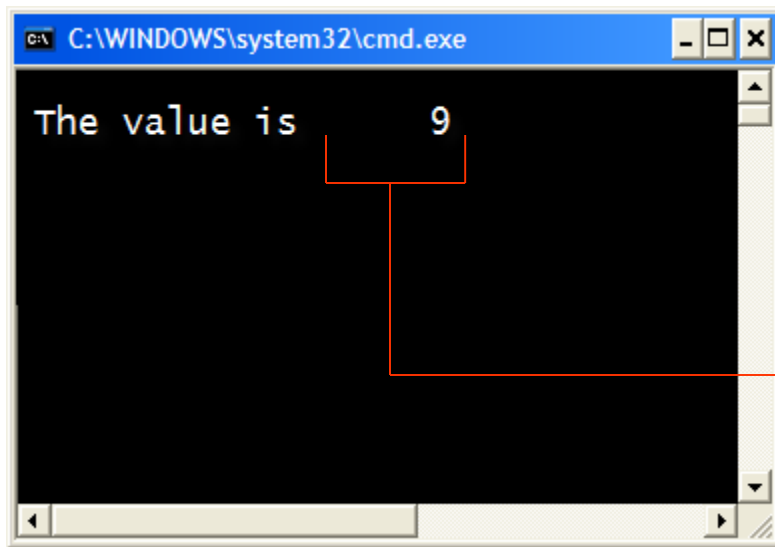


**The `%s` format specifier indicates that a string will be printed.**

# The `System.out.printf` Method

- Specifying a field width:

```
int number = 9;
System.out.printf("The value is %6d\n", number);
```

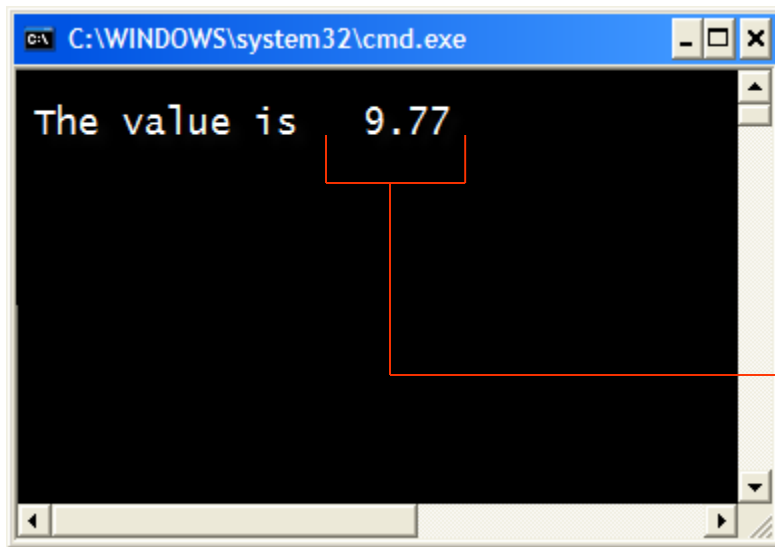


**The `%6d` format specifier indicates the integer will appear in a field that is 6 spaces wide.**

# The `System.out.printf` Method

- Another example:

```
double number = 9.76891;
System.out.printf("The value is %6.2f\n", number);
```



**The `%6.2f` format specifier indicates the number will appear in a field that is 6 spaces wide, and be rounded to 2 decimal places.**

# The `System.out.printf` Method

- See examples:
  - [Columns.java](#)
  - [CurrencyFormat.java](#)



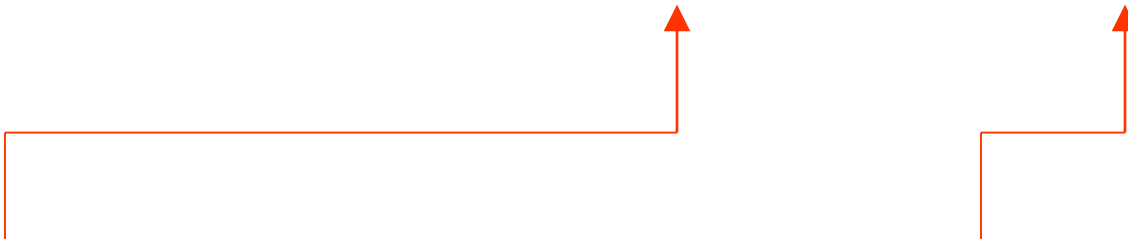
# The `String.format` Method

- The `String.format` method works exactly like the `System.out.printf` method, except that it does not display the formatted string on the screen.
- Instead, it returns a reference to the formatted string.
- You can assign the reference to a variable, and then use it later.

# The `String.format` Method

- The general format of the method is:

```
String.format(FormatString, ArgumentList);
```



***FormatString*** is  
a string that  
contains text and/or  
special formatting  
specifiers.

***ArgumentList*** is  
optional. It is a list of  
additional arguments that  
will be formatted according  
to the format specifiers  
listed in the format string.

# The `String.format` Method

- See examples:
  - [CurrencyFormat2.java](#)
  - [CurrencyFormat3.java](#)

# CHAPTER 4

## Loops and Files

starting out with >>>

# JAVA™

From Control Structures through Objects

6TH EDITION



TONY GADDIS

# Chapter Topics

Chapter 4 discusses the following main topics:

- The Increment and Decrement Operators
- The `while` Loop
- Using the `while` Loop for Input Validation
- The `do-while` Loop
- The `for` Loop
- Running Totals and Sentinel Values

# Chapter Topics

Chapter 4 discusses the following main topics:

- Nested Loops
- The `break` and `continue` Statements
- Deciding Which Loop to Use
- Introduction to File Input and Output
- Generating Random Numbers with the `Random` class

# The Increment and Decrement Operators

- There are numerous times where a variable must simply be incremented or decremented.

```
number = number + 1;
```

```
number = number - 1;
```

- Java provide shortened ways to increment and decrement a variable's value.
- Using the `++` or `--` unary operators, this task can be completed quickly.

```
number++; or ++number;
```

```
number--; or --number;
```

- Example: [IncrementDecrement.java](#)

# Differences Between Prefix and Postfix

- When an increment or decrement are the only operations in a statement, there is no difference between prefix and postfix notation.
- When used in an expression:
  - prefix notation indicates that the variable will be incremented or decremented prior to the rest of the equation being evaluated.
  - postfix notation indicates that the variable will be incremented or decremented after the rest of the equation has been evaluated.
- Example: [Prefix.java](#)



# The `while` Loop

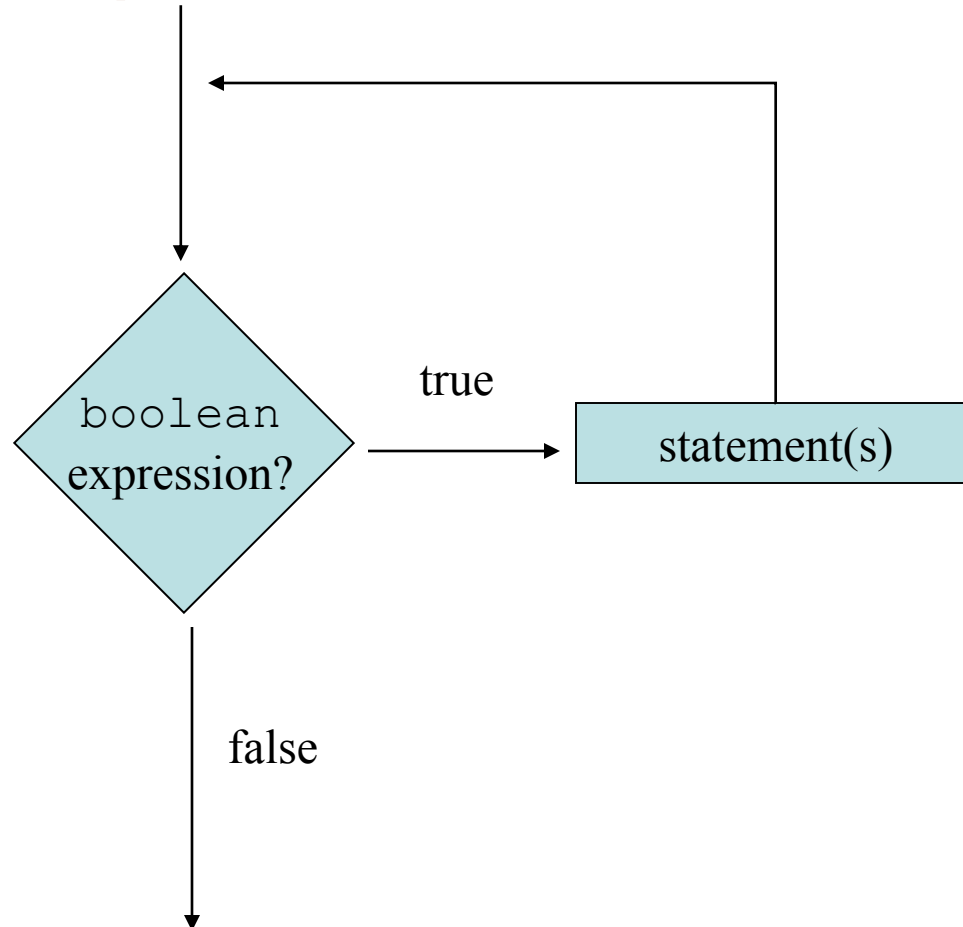
- Java provides three different looping structures.
- The `while` loop has the form:

```
while (condition)
{
 statements;
}
```
- While the condition is true, the statements will execute repeatedly.
- The `while` loop is a *pretest* loop, which means that it will test the value of the condition prior to executing the loop.

# The `while` Loop

- Care must be taken to set the condition to false somewhere in the loop so the loop will end.
- Loops that do not end are called *infinite loops*.
- A `while` loop executes 0 or more times. If the condition is false, the loop will not execute.
- Example: [WhileLoop.java](#)

# The `while` loop Flowchart



# Infinite Loops

- In order for a `while` loop to end, the condition must become false. The following loop will not end:

```
int x = 20;
while(x > 0)
{
 System.out.println("x is greater than 0");
}
```

- The variable `x` never gets decremented so it will always be greater than 0.
- Adding the `x--` above fixes the problem.

# Infinite Loops

- This version of the loop decrements  $x$  during each iteration:

```
int x = 20;
while(x > 0)
{
 System.out.println("x is greater than 0");
 x--;
}
```

# Block Statements in Loops

- Curly braces are required to enclose block statement while loops. (like block `if` statements)

```
while (condition)
{
 statement;
 statement;
 statement;
}
```

# The `while` Loop for Input Validation

- *Input validation* is the process of ensuring that user input is valid.

```
System.out.print("Enter a number in the " +
 "range of 1 through 100: ");
number = keyboard.nextInt();
// Validate the input.
while (number < 1 || number > 100)
{
 System.out.println("That number is invalid.");
 System.out.print("Enter a number in the " +
 "range of 1 through 100: ");
 number = keyboard.nextInt();
}
```

- Example: [SoccerTeams.java](#)

# The do-while Loop

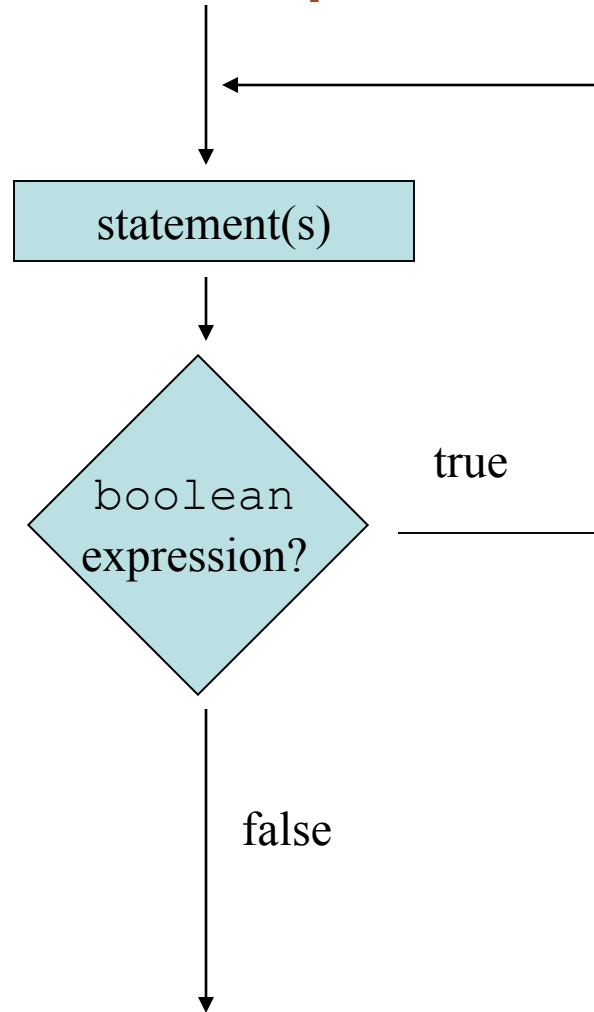
- The do-while loop is a *post-test* loop, which means it will execute the loop prior to testing the condition.
- The do-while loop (sometimes called called a do loop) takes the form:

```
do
{
 statement(s) ;
}while (condition) ;
```

- Example: [TestAverage1.java](#)



# The do-while Loop Flowchart

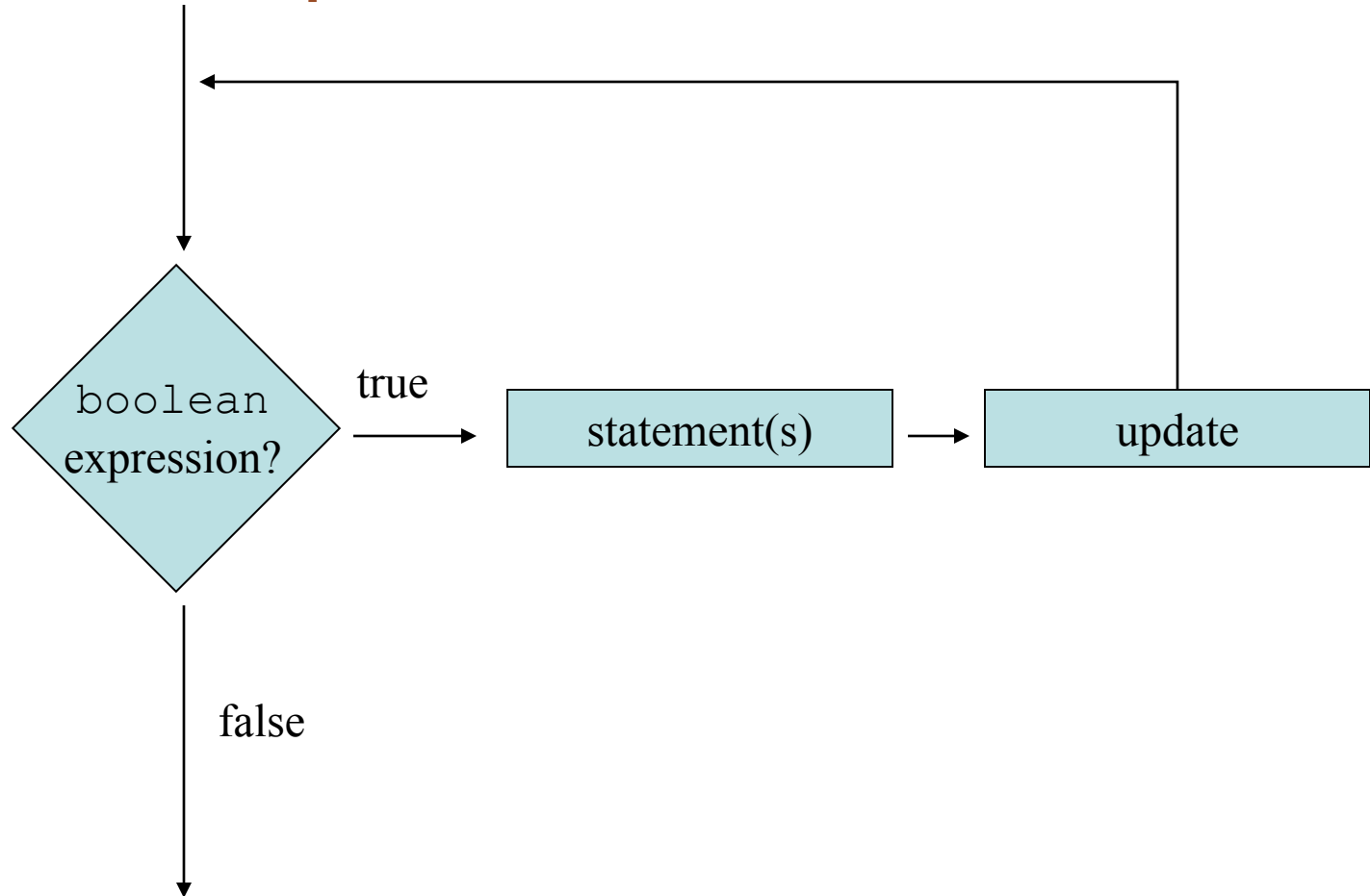


# The `for` Loop

- The `for` loop is a pre-test loop.
- The `for` loop allows the programmer to initialize a control variable, test a condition, and modify the control variable all in one line of code.
- The `for` loop takes the form:

```
for(initialization; test; update)
{
 statement(s) ;
}
```
- See example: [Squares.java](#)

# The `for` Loop Flowchart



# The Sections of The `for` Loop

- The *initialization section* of the `for` loop allows the loop to initialize its own control variable.
- The *test section* of the `for` statement acts in the same manner as the condition section of a `while` loop.
- The *update section* of the `for` loop is the last thing to execute at the end of each loop.
- Example: [UserSquares.java](#)

# The `for` Loop Initialization

- The initialization section of a `for` loop is optional; however, it is usually provided.
- Typically, `for` loops initialize a counter variable that will be tested by the test section of the loop and updated by the update section.
- The initialization section can initialize multiple variables.
- Variables declared in this section have scope only for the `for` loop.

# The Update Expression

- The update expression is usually used to increment or decrement the counter variable(s) declared in the initialization section of the for loop.
- The update section of the loop executes last in the loop.
- The update section may update multiple variables.
- Each variable updated is executed as if it were on a line by itself.

# Modifying The Control Variable

- You should avoid updating the control variable of a `for` loop within the body of the loop.
- The update section should be used to update the control variable.
- Updating the control variable in the `for` loop body leads to hard to maintain code and difficult debugging.

# Multiple Initializations and Updates

- The `for` loop may initialize and update multiple variables.

```
for(int i = 5, int j = 0; i < 10 || j < 20; i++, j+=2)
{
 statement(s);
}
```

- Note that the only parts of a `for` loop that are mandatory are the semicolons.

```
for(;;)
{
 statement(s);
} // infinite loop
```

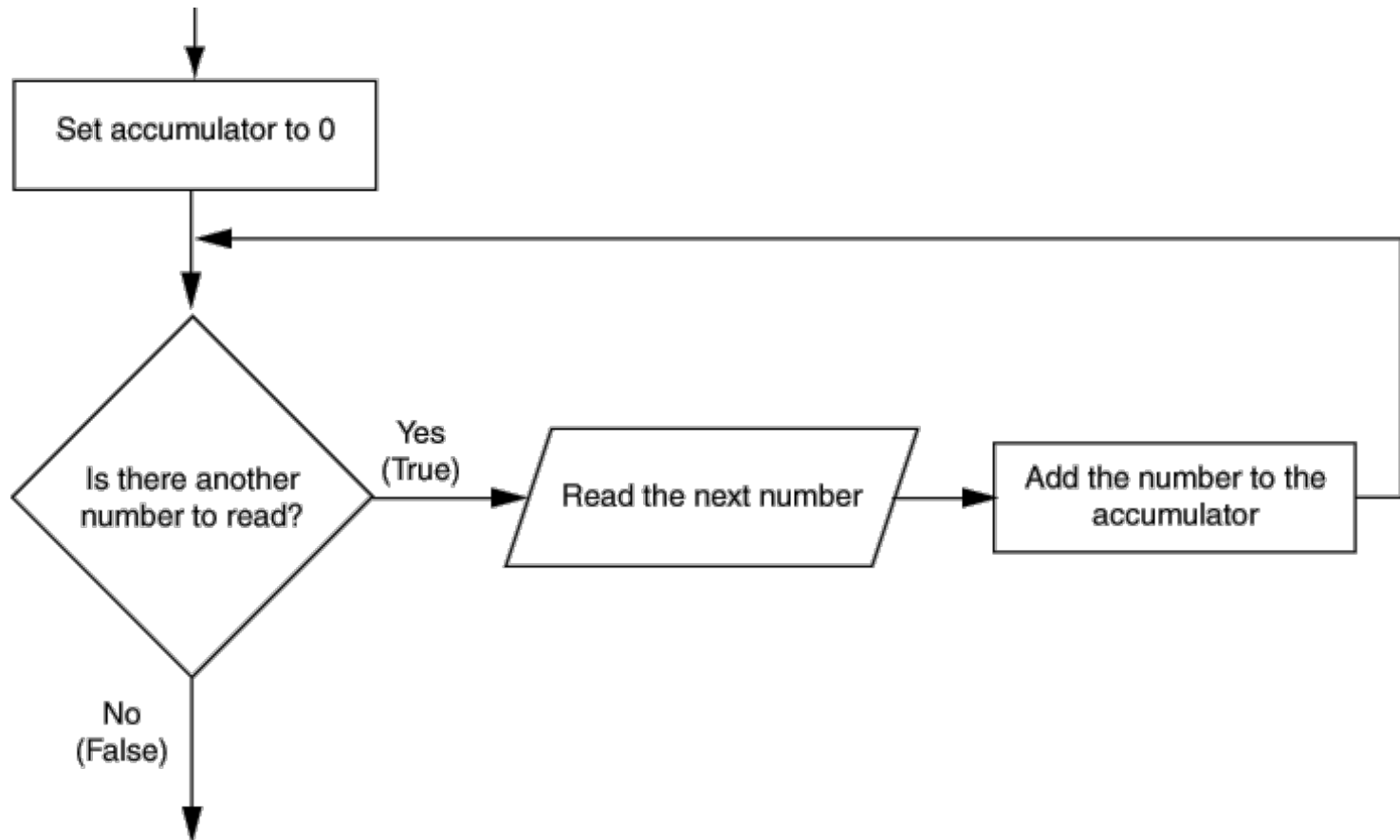
- If left out, the test section defaults to true.



# Running Totals

- Loops allow the program to keep running totals while evaluating data.
- Imagine needing to keep a running total of user input.
- Example: [TotalSales.java](#)

# Logic for Calculating a Running Total



# Sentinel Values

- Sometimes the end point of input data is not known.
- A *sentinel value* can be used to notify the program to stop acquiring input.
- If it is a user input, the user could be prompted to input data that is not normally in the input data range (i.e.  $-1$  where normal input would be positive.)
- Programs that get file input typically use the end-of-file marker to stop acquiring input data.
- Example: [SoccerPoints.java](#)

# Nested Loops

- Like `if` statements, loops can be nested.
- If a loop is nested, the inner loop will execute all of its iterations for each time the outer loop executes once.

```
for(int i = 0; i < 10; i++)
 for(int j = 0; j < 10; j++)
 loop statements;
```

- The loop statements in this example will execute 100 times.
- Example: [Clock.java](#)

# The `break` Statement

- The `break` statement can be used to abnormally terminate a loop.
- The use of the `break` statement in loops bypasses the normal mechanisms and makes the code hard to read and maintain.
- It is considered bad form to use the `break` statement in this manner.

# The `continue` Statement

- The `continue` statement will cause the currently executing iteration of a loop to terminate and the next iteration will begin.
- The `continue` statement will cause the evaluation of the condition in `while` and `for` loops.
- Like the `break` statement, the `continue` statement should be avoided because it makes the code hard to read and debug.

# Deciding Which Loops to Use

- The `while` loop:
  - Pretest loop
  - Use it where you do not want the statements to execute if the condition is false in the beginning.
- The `do-while` loop:
  - Post-test loop
  - Use it where you want the statements to execute at least one time.
- The `for` loop:
  - Pretest loop
  - Use it where there is some type of counting variable that can be evaluated.

# File Input and Output

- Reentering data all the time could get tedious for the user.
- The data can be saved to a file.
  - Files can be *input files* or *output files*.
- Files:
  - Files have to be opened.
  - Data is then written to the file.
  - The file must be closed prior to program termination.
- In general, there are two types of files:
  - binary
  - text



# Writing Text To a File

- To open a file for text output you create an instance of the `PrintWriter` class.

```
PrintWriter outputFile = new PrintWriter("StudentData.txt");
```



**Pass the name of the file that you wish to open as an argument to the `PrintWriter` constructor.**

**Warning: if the file already exists, it will be erased and replaced with a new file.**

# The `PrintWriter` Class

- The `PrintWriter` class allows you to write data to a file using the `print` and `println` methods, as you have been using to display data on the screen.
- Just as with the `System.out` object, the `println` method of the `PrintWriter` class will place a newline character after the written data.
- The `print` method writes data without writing the newline character.

# The PrintWriter Class

**Open the file.**

```
PrintWriter outputFile = new PrintWriter("Names.txt");
outputFile.println("Chris");
outputFile.println("Kathryn");
outputFile.println("Jean");
outputFile.close();
```

**Close the file.**

**Write data to the file.**

# The `PrintWriter` Class

- To use the `PrintWriter` class, put the following `import` statement at the top of the source file:

```
import java.io.*;
```

- See example: [FileWriteDemo.java](#)

# Exceptions

- When something unexpected happens in a Java program, an *exception* is thrown.
- The method that is executing when the exception is thrown must either handle the exception or pass it up the line.
- Handling the exception will be discussed later.
- To pass it up the line, the method needs a `throws` clause in the method header.

# Exceptions

- To insert a `throws` clause in a method header, simply add the word *throws* and the name of the expected exception.
- `PrintWriter` objects can throw an `IOException`, so we write the `throws` clause like this:

```
public static void main(String[] args) throws IOException
```

# Appending Text to a File

- To avoid erasing a file that already exists, create a `FileWriter` object in this manner:

```
FileWriter fw =
 new FileWriter("names.txt", true);
```

- Then, create a `PrintWriter` object in this manner:

```
PrintWriter fw = new PrintWriter(fw);
```

# Specifying a File Location

- On a Windows computer, paths contain backslash (\) characters.
- Remember, if the backslash is used in a string literal, it is the escape character so you must use two of them:

```
PrintWriter outFile =
 new PrintWriter("A:\\PriceList.txt");
```



# Specifying a File Location

- This is only necessary if the backslash is in a string literal.
- If the backslash is in a `String` object then it will be handled properly.
- Fortunately, Java allows Unix style filenames using the forward slash (/) to separate directories:

```
PrintWriter outFile = new
 PrintWriter("/home/rharrison/names.txt");
```

# Reading Data From a File

- You use the `File` class and the `Scanner` class to read data from a file:

**Pass the name of the file as an argument to the `File` class constructor.**

```
File myFile = new File("Customers.txt");
Scanner inputFile = new Scanner(myFile);
```

**Pass the `File` object as an argument to the `Scanner` class constructor.**

# Reading Data From a File

```
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter the filename: ");
String filename = keyboard.nextLine();
File file = new File(filename);
Scanner inputFile = new Scanner(file);
```

- The lines above:
  - Creates an instance of the `Scanner` class to read from the keyboard
  - Prompt the user for a filename
  - Get the filename from the user
  - Create an instance of the `File` class to represent the file
  - Create an instance of the `Scanner` class that reads from the file

# Reading Data From a File

- Once an instance of `Scanner` is created, data can be read using the same methods that you have used to read keyboard input (`nextLine`, `nextInt`, `nextDouble`, etc).

```
// Open the file.
File file = new File("Names.txt");
Scanner inputFile = new Scanner(file);
// Read a line from the file.
String str = inputFile.nextLine();
// Close the file.
inputFile.close();
```

# Exceptions

- The `Scanner` class can throw an `IOException` when a `File` object is passed to its constructor.
- So, we put a `throws IOException` clause in the header of the method that instantiates the `Scanner` class.
- See Example: [ReadFirstLine.java](#)

# Detecting The End of a File

- The Scanner class's hasNext () method will return true if another item can be read from the file.

```
// Open the file.
File file = new File(filename);
Scanner inputFile = new Scanner(file);
// Read until the end of the file.
while (inputFile.hasNext())
{
 String str = inputFile.nextLine();
 System.out.println(str);
}
inputFile.close(); // close the file when done.
```

# Detecting the End of a File

- See example: [FileReadDemo.java](#)

# Generating Random Numbers with the Random Class

- Some applications, such as games and simulations, require the use of randomly generated numbers.
- The Java API has a class, `Random`, for this purpose. To use the `Random` class, use the following `import` statement and create an instance of the class.

```
import java.util.Random;
```

```
Random randomNumbers = new Random();
```



# Some Methods of the Random Class

| Method                      | Description                                                                                                                                                                |
|-----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>nextDouble()</code>   | Returns the next random number as a <code>double</code> . The number will be within the range of 0.0 and 1.0.                                                              |
| <code>nextFloat()</code>    | Returns the next random number as a <code>float</code> . The number will be within the range of 0.0 and 1.0.                                                               |
| <code>nextInt()</code>      | Returns the next random number as an <code>int</code> . The number will be within the range of an <code>int</code> , which is $-2,147,483,648$ to $+2,147,483,648$ .       |
| <code>nextInt(int n)</code> | This method accepts an integer argument, <code>n</code> . It returns a random number as an <code>int</code> . The number will be within the range of 0 to <code>n</code> . |

See example: [RollDice.java](#)

## CHAPTER 5

# Methods

starting out with >>>

# JAVA™

From Control Structures through Objects

6TH EDITION



TONY GADDIS

# Chapter Topics

Chapter 5 discusses the following main topics:

- Introduction to Methods
- Passing Arguments to a Method
- More About Local Variables
- Returning a Value from a Method
- Problem Solving with Methods

# Why Write Methods?

- Methods are commonly used to break a problem down into small manageable pieces. This is called *divide and conquer*.
- Methods simplify programs. If a specific task is performed in several places in the program, a method can be written once to perform that task, and then be executed anytime it is needed. This is known as *code reuse*.

# void Methods and Value-Returning Methods

- A `void` method is one that simply performs a task and then terminates.

```
System.out.println("Hi!");
```

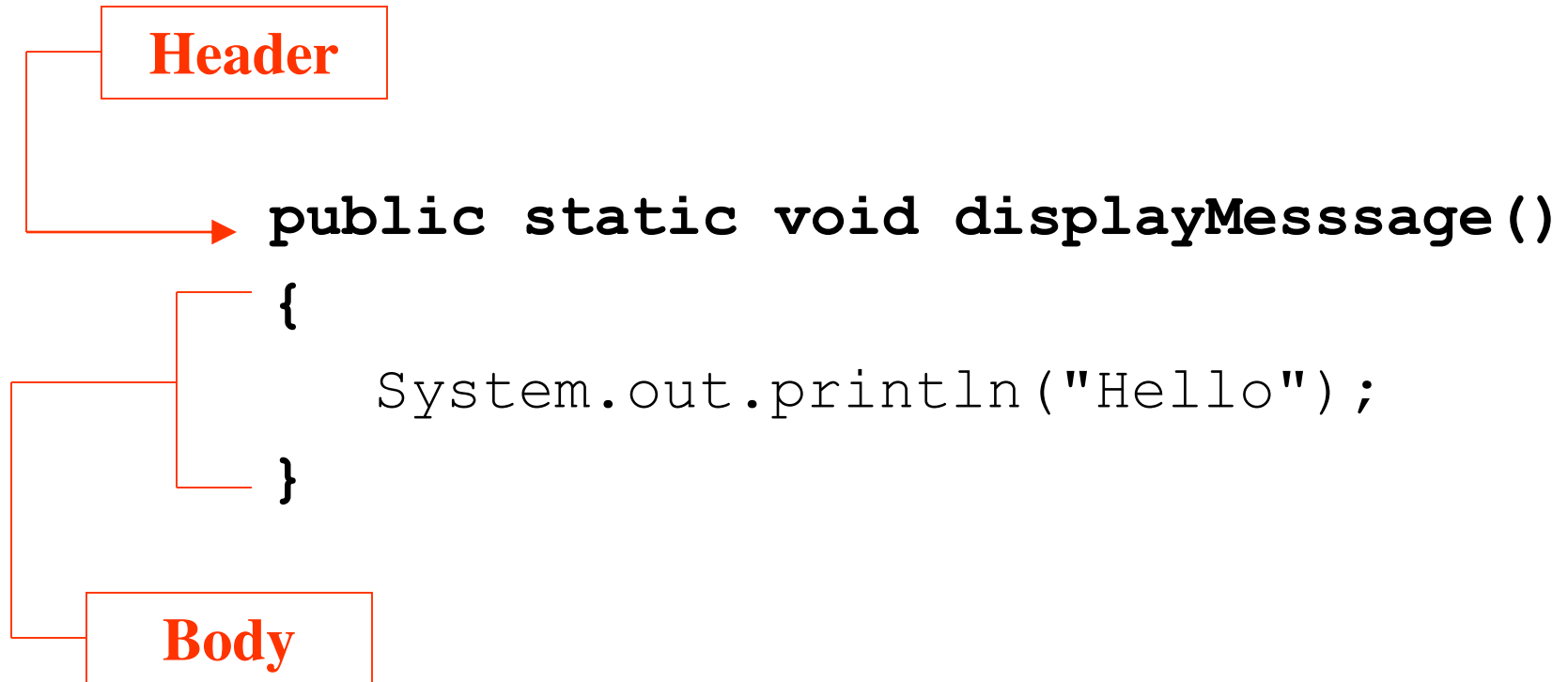
- A value-returning method not only performs a task, but also sends a value back to the code that called it.

```
int number = Integer.parseInt("700");
```

# Defining a `void` Method

- To create a method, you must write a definition, which consists of a *header* and a *body*.
- The method header, which appears at the beginning of a method definition, lists several important things about the method, including the method's name.
- The method body is a collection of statements that are performed when the method is executed.

# Two Parts of Method Declaration



# Parts of a Method Header

Method  
Modifiers

Return  
Type

Method  
Name

Parentheses



```
public static void displayMessage ()
{
 System.out.println("Hello");
}
```



# Parts of a Method Header

- Method modifiers
  - `public`—method is publicly available to code outside the class
  - `static`—method belongs to a class, not a specific object.
- Return type—`void` or the data type from a value-returning method
- Method name—name that is descriptive of what the method does
- Parentheses—contain nothing or a list of one or more variable declarations if the method is capable of receiving arguments.

# Calling a Method

- A method executes when it is called.
- The `main` method is automatically called when a program starts, but other methods are executed by method call statements.

**`displayMessage()` ;**

- Notice that the method modifiers and the `void` return type are not written in the method call statement. Those are only written in the method header.
- Examples: [SimpleMethod.java](#), [LoopCall.java](#), [CreditCard.java](#), [DeepAndDeeper.java](#)

# Documenting Methods

- A method should always be documented by writing comments that appear just before the method's definition.
- The comments should provide a brief explanation of the method's purpose.
- The documentation comments begin with `/*` and end with `*/`.

# Passing Arguments to a Method

- Values that are sent into a method are called arguments.

```
System.out.println("Hello");
number = Integer.parseInt(str);
```

- The data type of an argument in a method call must correspond to the variable declaration in the parentheses of the method declaration. The parameter is the variable that holds the value being passed into a method.
- By using parameter variables in your method declarations, you can design your own methods that accept data this way. See example: [PassArg.java](#)

# Passing 5 to the `displayValue` Method

`displayValue(5) ;`    The argument 5 is copied into the parameter variable **num**.



```
public static void displayValue(int
 num)
{
 System.out.println("The value is " + num) ;
}
```

The method will display      The value is 5

# Argument and Parameter Data Type Compatibility

- When you pass an argument to a method, be sure that the argument's data type is compatible with the parameter variable's data type.
- Java will automatically perform widening conversions, but narrowing conversions will cause a compiler error.

```
double d = 1.0;
displayValue(d);
```

**Error! Can't convert  
double to int**



# Passing Multiple Arguments

The argument 5 is copied into the **num1** parameter.

The argument 10 is copied into the **num2** parameter.

`showSum(5, 10);`      **NOTE: Order matters!**



```
public static void showSum(double num1, double num2)
{
 double sum; //to hold the sum
 sum = num1 + num2;
 System.out.println("The sum is " + sum);
}
```

The diagram consists of a red line starting from the space between the arguments '5' and '10' in the function call. It extends horizontally to the right and then branches into two vertical lines with downward-pointing arrows. The first arrow points to the parameter 'num1' in the method signature, and the second arrow points to the parameter 'num2'.

# Arguments are Passed by Value

- In Java, all arguments of the primitive data types are *passed by value*, which means that only a copy of an argument's value is passed into a parameter variable.
- A method's parameter variables are separate and distinct from the arguments that are listed inside the parentheses of a method call.
- If a parameter variable is changed inside a method, it has no affect on the original argument.
- See example: [PassByValue.java](#)



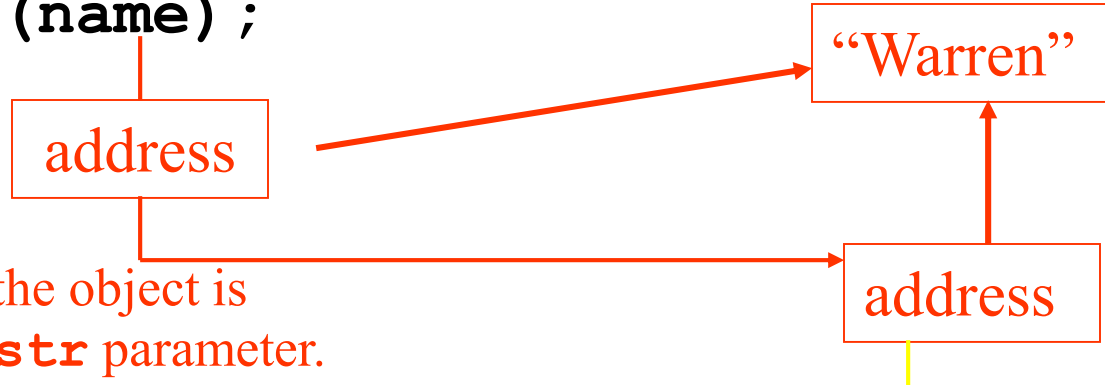
# Passing Object References to a Method

- Recall that a class type variable does not hold the actual data item that is associated with it, but holds the memory address of the object. A variable associated with an object is called a reference variable.
- When an object such as a `String` is passed as an argument, it is actually a reference to the object that is passed.

# Passing a Reference as an Argument

Both variables reference the same object

```
showLength(name) ;
```



The address of the object is copied into the **str** parameter.

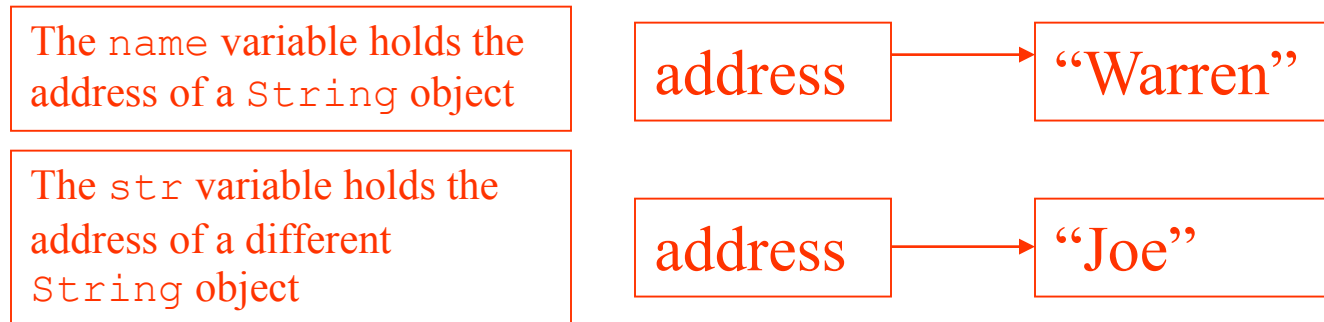
```
public static void showLength(String str)
{
 System.out.println(str + " is " +
 str.length() + " characters long.");
 str = "Joe" // see next slide
}
```

# Strings are Immutable Objects

- Strings are immutable objects, which means that they cannot be changed. When the line

```
str = "Joe";
```

is executed, it cannot change an immutable object, so creates a new object.



- See example: [PassString.java](#)

# @param Tag in Documentation Comments

- You can provide a description of each parameter in your documentation comments by using the `@param` tag.
- General format

```
@param parameterName Description
```
- See example: [TwoArgs2.java](#)
- All `@param` tags in a method's documentation comment must appear after the general description. The description can span several lines.

# More About Local Variables

- A local variable is declared inside a method and is not accessible to statements outside the method.
- Different methods can have local variables with the same names because the methods cannot see each other's local variables.
- A method's local variables exist only while the method is executing. When the method ends, the local variables and parameter variables are destroyed and any values stored are lost.
- Local variables are not automatically initialized with a default value and must be given a value before they can be used.
- See example: [LocalVars.java](#)

# Returning a Value from a Method

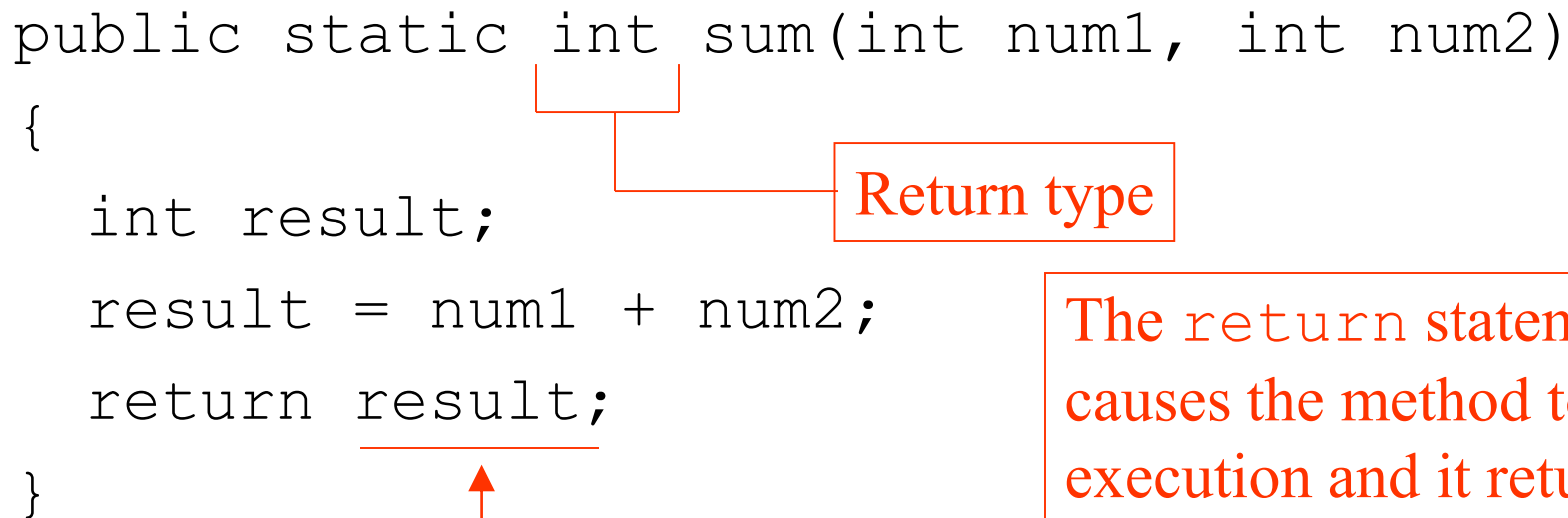
- Data can be passed into a method by way of the parameter variables. Data may also be returned from a method, back to the statement that called it.

```
int num = Integer.parseInt("700");
```

- The string “700” is passed into the `parseInt` method.
- The `int` value 700 is returned from the method and assigned to the `num` variable.

# Defining a Value-Returning Method

```
public static int sum(int num1, int num2)
{
 int result;
 result = num1 + num2;
 return result;
}
```

A diagram illustrating the components of a value-returning method. A red bracket connects the 'int' return type in the method signature to the 'int' data type of the 'result' variable. Another red bracket connects the 'result' variable in the 'return' statement to a text box explaining that the expression must match the return type. A third red arrow points from the 'result' variable in the 'return' statement to a text box explaining that the return statement ends execution and returns a value.

Return type

The **return statement** causes the method to end execution and it returns a value back to the statement that called the method.

This expression must be of the same data type as the return type

# Calling a Value-Returning Method

```
total = sum(value1, value2);
```

20

40

```
public static int sum(int num1, int num2)
```

```
{
```

60

```
 int result;
```

```
 result = num1 + num2;
```

```
 return result;
```

```
}
```



# @return Tag in Documentation Comments

- You can provide a description of the return value in your documentation comments by using the `@return` tag.
- General format  
`@return Description`
- See example: [ValueReturn.java](#)
- The `@return` tag in a method's documentation comment must appear after the general description. The description can span several lines.

# Returning a boolean Value

- Sometimes we need to write methods to test arguments for validity and return true or false

```
public static boolean isValid(int number)
{
 boolean status;
 if(number >= 1 && number <= 100)
 status = true;
 else
 status = false;
 return status;
}
```

## Calling code:

```
int value = 20;
If(isValid(value))
 System.out.println("The value is within range");
else
 System.out.println("The value is out of range");
```

# Returning a Reference to a String Object

```
customerName = fullName("John", "Martin");
```

```
public static String fullName(String first, String last)
{
 String name;
 name = first + " " + last;
 return name;
}
```

address

"John Martin"

Local variable `name` holds the reference to the object. The return statement sends a copy of the reference back to the call statement and it is stored in `customerName`.

See example:

[ReturnString.java](#)

# Problem Solving with Methods

- A large, complex problem can be solved a piece at a time by methods.
- The process of breaking a problem down into smaller pieces is called *functional decomposition*.
- See example: [SalesReport.java](#)
- If a method calls another method that has a `throws` clause in its header, then the calling method should have the same `throws` clause.

# Calling Methods that Throw Exceptions

- Note that the `main` and `getTotalSales` methods in *SalesReport.java* have a `throws IOException` clause.
- All methods that use a `Scanner` object to open a file must throw or handle `IOException`.
- You will learn how to handle exceptions in Chapter 12.
- For now, understand that Java required any method that interacts with an external entity, such as the file system to either throw an exception to be handles elsewhere in your application or to handle the exception locally.

## CHAPTER 6

# A First Look at Classes

starting out with >>>

# JAVA™

From Control Structures through Objects

6TH EDITION



TONY GADDIS

# Chapter Topics

Chapter 6 discusses the following main topics:

- Objects and Classes
- Writing a Simple Class, Step by Step
- Instance Fields and Methods
- Constructors
- Passing Objects as Arguments
- Overloading Methods and Constructors
- Scope of Instance Fields
- Packages and `import` Statements

# Objects and Classes

- An object exists in memory, and performs a specific task.
- Objects have two general capabilities:
  - Objects can store data. The pieces of data stored in an object are known as *fields*.
  - Objects can perform operations. The operations that an object can perform are known as *methods*.



# Objects and Classes

- You have already used the following objects:
  - `Scanner` objects, for reading input
  - `Random` objects, for generating random numbers
  - `PrintWriter` objects, for writing data to files
- When a program needs the services of a particular type of object, it creates that object in memory, and then calls that object's methods as necessary.

# Objects and Classes

- Classes: Where Objects Come From
  - A *class* is code that describes a particular type of object. It specifies the data that an object can hold (the object's fields), and the actions that an object can perform (the object's methods).
  - You can think of a class as a code "blueprint" that can be used to create a particular type of object.

# Objects and Classes

- When a program is running, it can use the class to create, in memory, as many objects of a specific type as needed.
- Each object that is created from a class is called an *instance* of the class.

# Objects and Classes

*Example:*

This expression creates a  
Scanner object in memory.

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

The object's memory address  
is assigned to the keyboard  
variable.

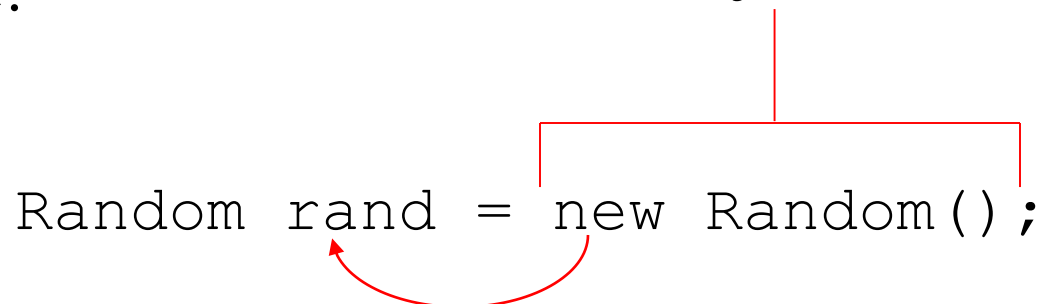


# Objects and Classes

*Example:*

This expression creates a  
Random object in memory.

```
Random rand = new Random();
```



The object's memory address is  
assigned to the `rand` variable.



# Objects and Classes

*Example:*

This expression creates a  
PrintWriter object in memory.

```
PrintWriter outputFile = new PrintWriter("numbers.txt");
```

The object's memory address is assigned to  
the `outputFile` variable.



# Objects and Classes

- The Java API provides many classes
  - So far, the classes that you have created objects from are provided by the Java API.
  - Examples:
    - Scanner
    - Random
    - PrintWriter
- See [ObjectDemo.java](#)

# Writing a Class, Step by Step

- A `Rectangle` object will have the following fields:
  - `length`. The `length` field will hold the rectangle's length.
  - `width`. The `width` field will hold the rectangle's width.



# Writing a Class, Step by Step

- The `Rectangle` class will also have the following methods:
  - **`setLength`**. The `setLength` method will store a value in an object's `length` field.
  - **`setWidth`**. The `setWidth` method will store a value in an object's `width` field.
  - **`getLength`**. The `getLength` method will return the value in an object's `length` field.
  - **`getWidth`**. The `getWidth` method will return the value in an object's `width` field.
  - **`getArea`**. The `getArea` method will return the area of the rectangle, which is the result of the object's `length` multiplied by its `width`.

# UML Diagram

- Unified Modeling Language (UML) provides a set of standard diagrams for graphically depicting object-oriented systems.

Class name goes here



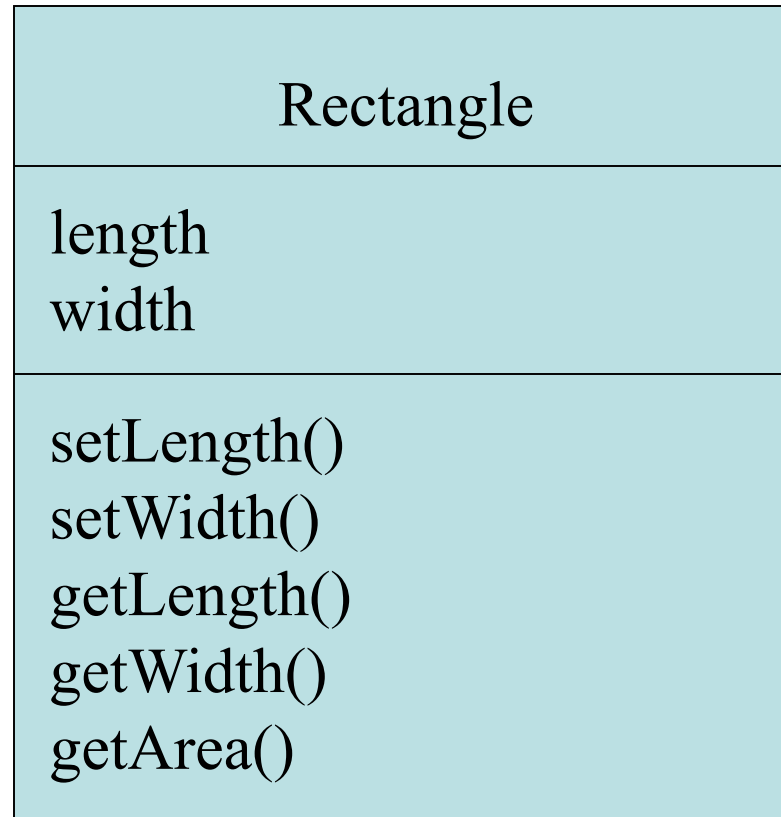
Fields are listed here



Methods are listed here



# UML Diagram for Rectangle class



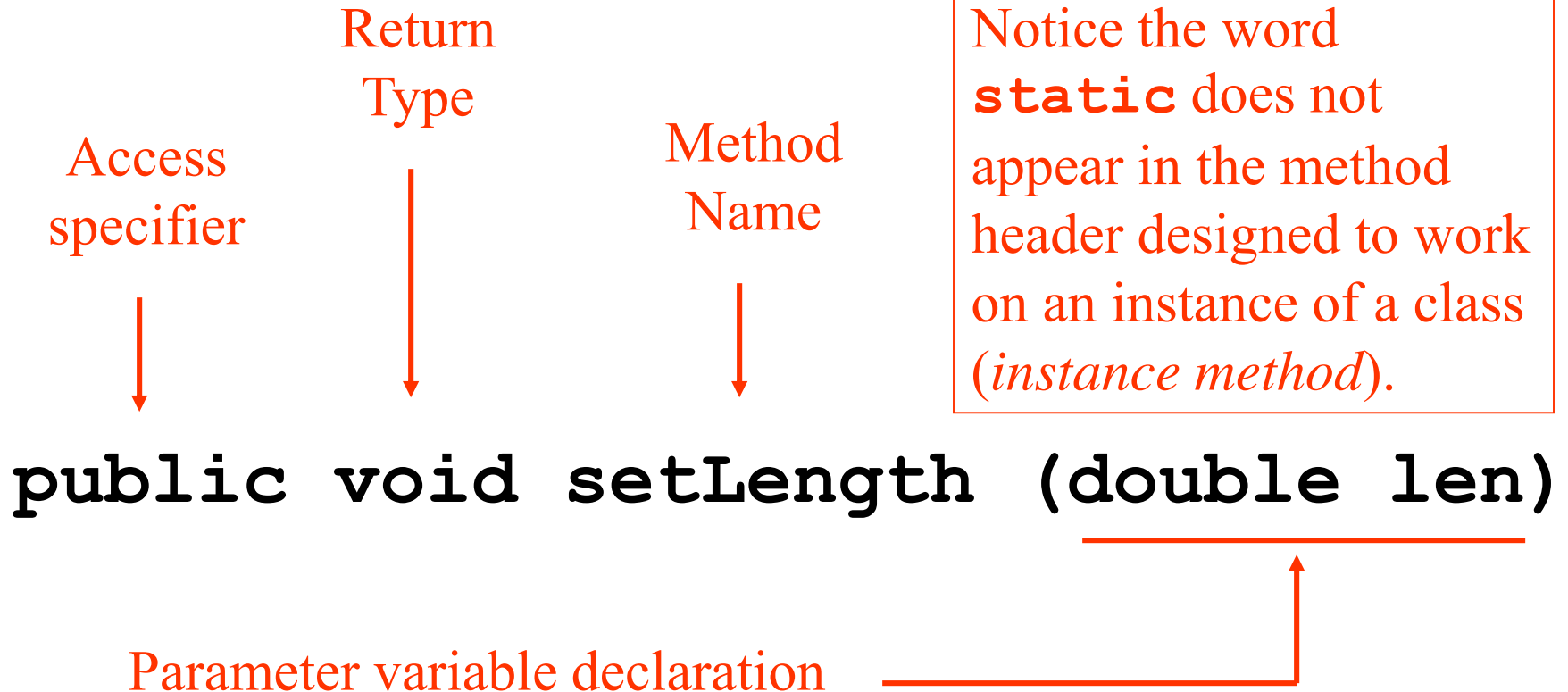
# Writing the Code for the Class Fields

```
public class Rectangle
{
 private double length;
 private double width;
}
```

# Access Specifiers

- An access specifier is a Java keyword that indicates how a field or method can be accessed.
- `public`
  - When the `public` access specifier is applied to a class member, the member can be accessed by code inside the class or outside.
- `private`
  - When the `private` access specifier is applied to a class member, the member cannot be accessed by code outside the class. The member can be accessed only by methods that are members of the same class.

# Header for the `setLength` Method



# Writing and Demonstrating the setLength Method

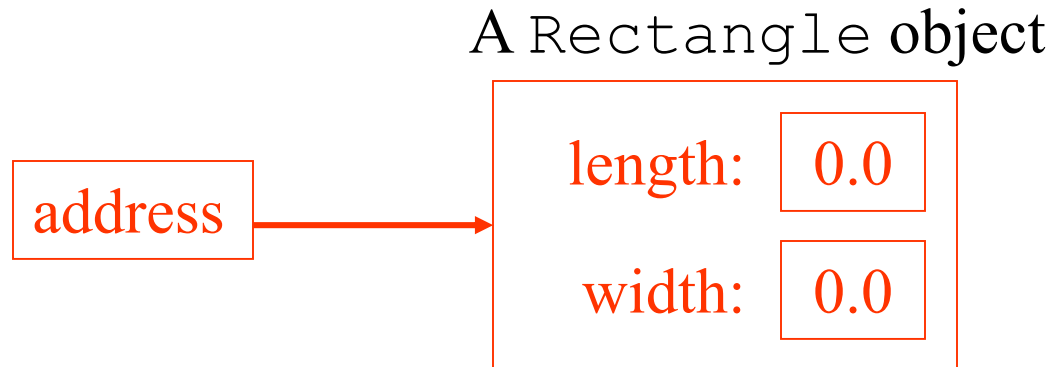
```
/**
 The setLength method stores a value in the
 length field.
 @param len The value to store in length.
*/
public void setLength(double len)
{
 length = len;
}
```

Examples: [Rectangle.java](#), [LengthDemo.java](#)

# Creating a Rectangle object

```
Rectangle box = new Rectangle ();
```

The box  
variable holds  
the address of  
the Rectangle  
object.

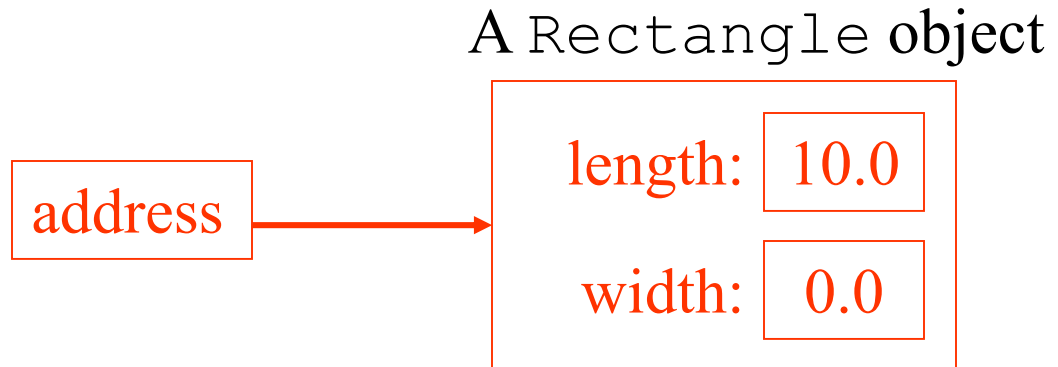




# Calling the `setLength` Method

```
box.setLength(10.0);
```

The `box` variable holds the address of the `Rectangle` object.



*This is the state of the `box` object after the `setLength` method executes.*

# Writing the `getLength` Method

```
/**
 The getLength method returns a Rectangle
 object's length.
 @return The value in the length field.
 */
public double getLength()
{
 return length;
}
```

Similarly, the `setWidth` and `getWidth` methods can be created.

Examples: [Rectangle.java](#), [LengthWidthDemo.java](#)

# Writing and Demonstrating the `getArea` Method

```
/**
 The getArea method returns a Rectangle
 object's area.
 @return The product of length times width.
*/
public double getArea()
{
 return length * width;
}
```

Examples: [Rectangle.java](#), [RectangleDemo.java](#)

# Accessor and Mutator Methods

- Because of the concept of data hiding, fields in a class are private.
- The methods that retrieve the data of fields are called *accessors*.
- The methods that modify the data of fields are called *mutators*.
- Each field that the programmer wishes to be viewed by other classes needs an accessor.
- Each field that the programmer wishes to be modified by other classes needs a mutator.

# Accessors and Mutators

- For the `Rectangle` example, the accessors and mutators are:
  - **`setLength`** : Sets the value of the `length` field.  
`public void setLength(double len) ...`
  - **`setWidth`** : Sets the value of the `width` field.  
`public void setLength(double w) ...`
  - **`getLength`** : Returns the value of the `length` field.  
`public double getLength() ...`
  - **`getWidth`** : Returns the value of the `width` field.  
`public double getWidth() ...`
- Other names for these methods are *getters* and *setters*.

# Data Hiding

- An object hides its internal, private fields from code that is outside the class that the object is an instance of.
- Only the class's methods may directly access and make changes to the object's internal data.
- Code outside the class must use the class's public methods to operate on an object's private fields.

# Data Hiding

- Data hiding is important because classes are typically used as components in large software systems, involving a team of programmers.
- Data hiding helps enforce the integrity of an object's internal data.

# Stale Data

- Some data is the result of a calculation.
- Consider the area of a rectangle.
  - $length \times width$
- It would be impractical to use an *area* variable here.
- Data that requires the calculation of various factors has the potential to become *stale*.
- To avoid stale data, it is best to calculate the value of that data within a method rather than store it in a variable.



# Stale Data

- Rather than use an area variable in a Rectangle class:

```
public double getArea()
{
 return length * width;
}
```

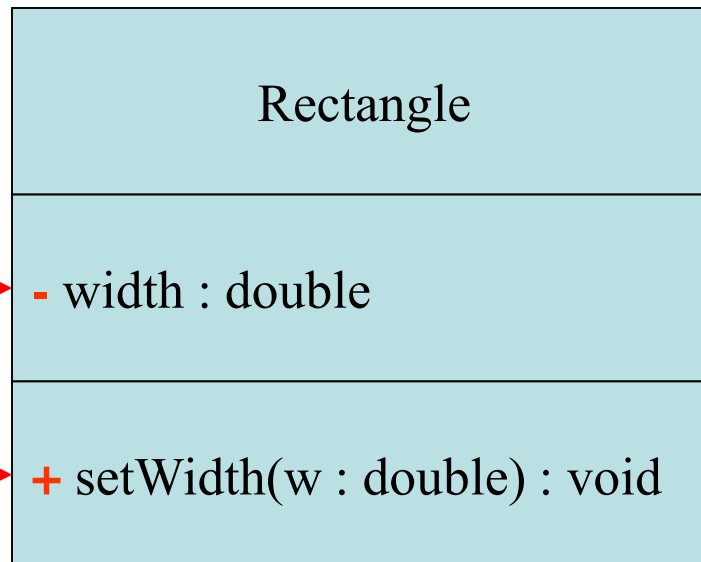
- This dynamically calculates the value of the rectangle's area when the method is called.
- Now, any change to the length or width variables will not leave the area of the rectangle stale.

# UML Data Type and Parameter Notation

- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.

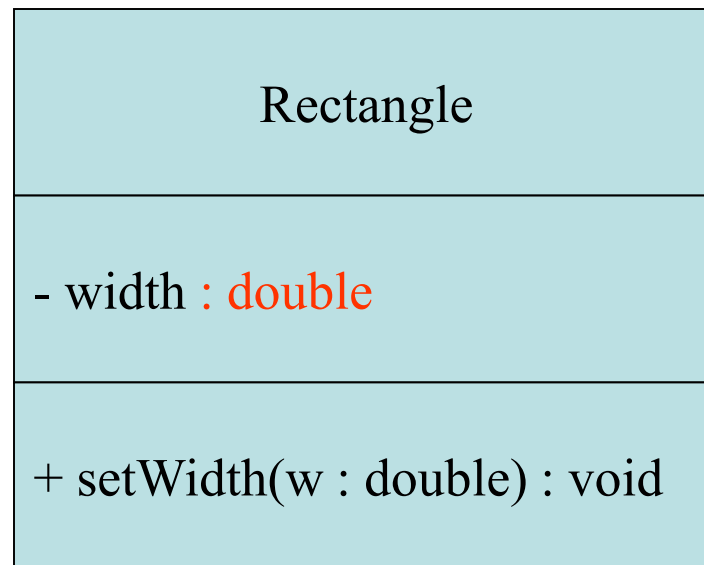
Access modifiers  
are denoted as:

+ public  
- private



# UML Data Type and Parameter Notation

- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.

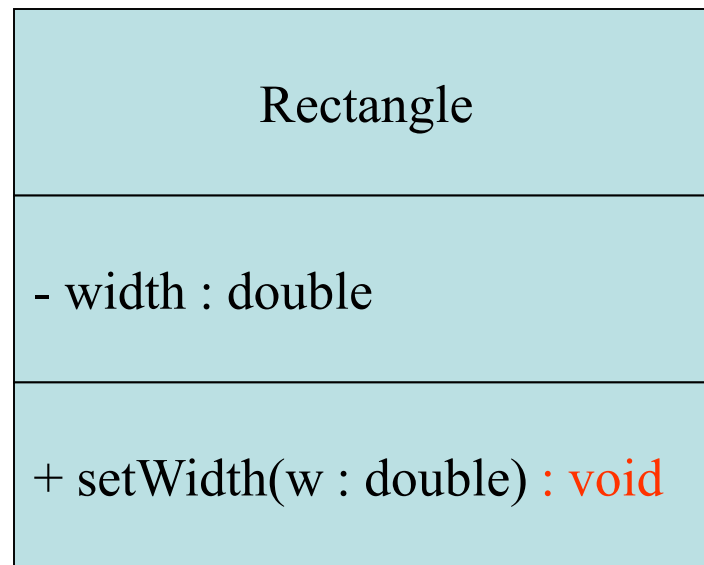


Variable types are placed after the variable name, separated by a colon.



# UML Data Type and Parameter Notation

- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.



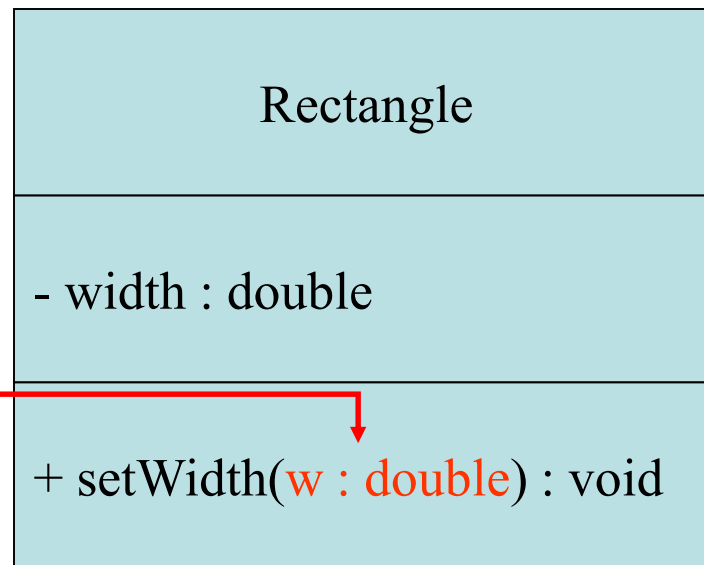
Method return types are placed after the method declaration name, separated by a colon.



# UML Data Type and Parameter Notation

- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.

Method parameters are shown inside the parentheses using the same notation as variables.



# Converting the UML Diagram to Code

- Putting all of this information together, a Java class file can be built easily using the UML diagram.
- The UML diagram parts match the Java class file structure.

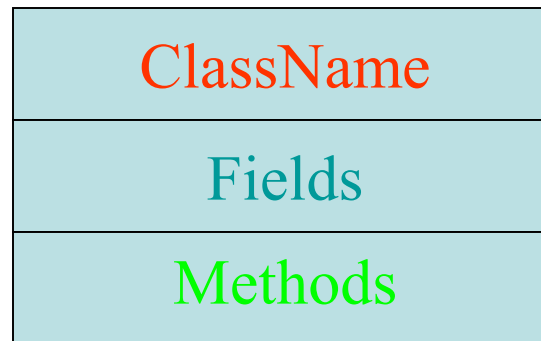
class header

{

Fields

Methods

}



# Converting the UML Diagram to Code

The structure of the class can be compiled and tested without having bodies for the methods. Just be sure to put in dummy return values for methods that have a return type other than void.

| Rectangle                                                                                                                                   |
|---------------------------------------------------------------------------------------------------------------------------------------------|
| - width : double<br>- length : double                                                                                                       |
| + setWidth(w : double) : void<br>+ setLength(len : double): void<br>+ getWidth() : double<br>+ getLength() : double<br>+ getArea() : double |

```
public class Rectangle
{
 private double width;
 private double length;

 public void setWidth(double w)
 {
 }
 public void setLength(double len)
 {
 }
 public double getWidth()
 {
 return 0.0;
 }
 public double getLength()
 {
 return 0.0;
 }
 public double getArea()
 {
 return 0.0;
 }
}
```

# Converting the UML Diagram to Code

Once the class structure has been tested, the method bodies can be written and tested.

| Rectangle                                                                                                                                   |
|---------------------------------------------------------------------------------------------------------------------------------------------|
| - width : double<br>- length : double                                                                                                       |
| + setWidth(w : double) : void<br>+ setLength(len : double): void<br>+ getWidth() : double<br>+ getLength() : double<br>+ getArea() : double |

```
public class Rectangle
{
 private double width;
 private double length;

 public void setWidth(double w)
 {
 width = w;
 }
 public void setLength(double len)
 {
 length = len;
 }
 public double getWidth()
 {
 return width;
 }
 public double getLength()
 {
 return length;
 }
 public double getArea()
 {
 return length * width;
 }
}
```



# Class Layout Conventions

- The layout of a source code file can vary by employer or instructor.
- A common layout is:
  - Fields listed first
  - Methods listed second
    - Accessors and mutators are typically grouped.
- There are tools that can help in formatting layout to specific standards.

# Instance Fields and Methods

- Fields and methods that are declared as previously shown are called *instance fields* and *instance methods*.
- Objects created from a class each have their own copy of instance fields.
- Instance methods are methods that are not declared with a special keyword, `static`.

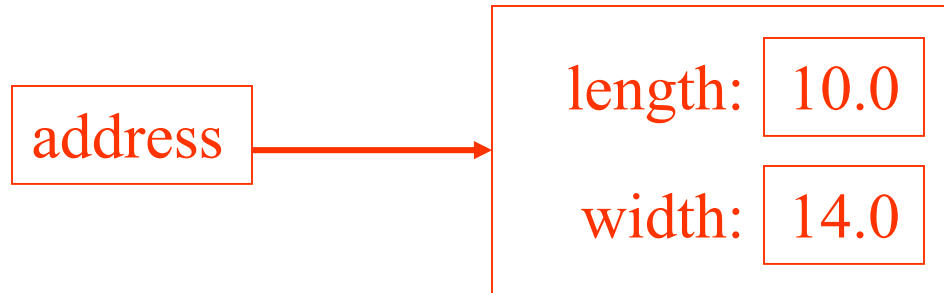
# Instance Fields and Methods

- Instance fields and instance methods require an object to be created in order to be used.
- See example: [RoomAreas.java](#)
- Note that each room represented in this example can have different dimensions.

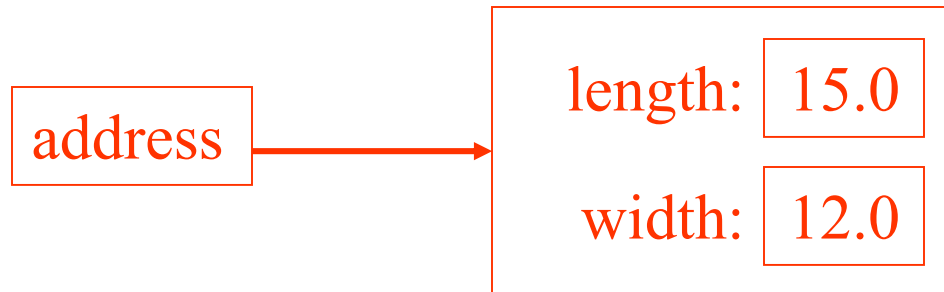
```
Rectangle kitchen = new Rectangle();
Rectangle bedroom = new Rectangle();
Rectangle den = new Rectangle();
```

# States of Three Different Rectangle Objects

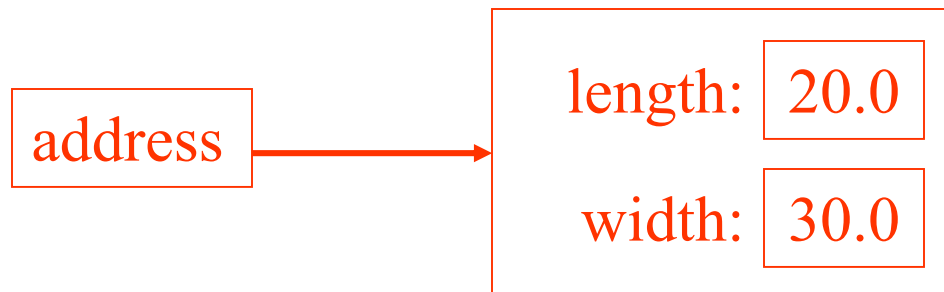
The kitchen variable holds the address of a Rectangle Object.



The bedroom variable holds the address of a Rectangle Object.



The den variable holds the address of a Rectangle Object.



# Constructors

- Classes can have special methods called *constructors*.
- A constructor is a method that is automatically called when an object is created.
- Constructors are used to perform operations at the time an object is created.
- Constructors typically initialize instance fields and perform other object initialization tasks.

# Constructors

- Constructors have a few special properties that set them apart from normal methods.
  - Constructors have the same name as the class.
  - Constructors have no return type (not even `void`).
  - Constructors may not return any values.
  - Constructors are typically public.

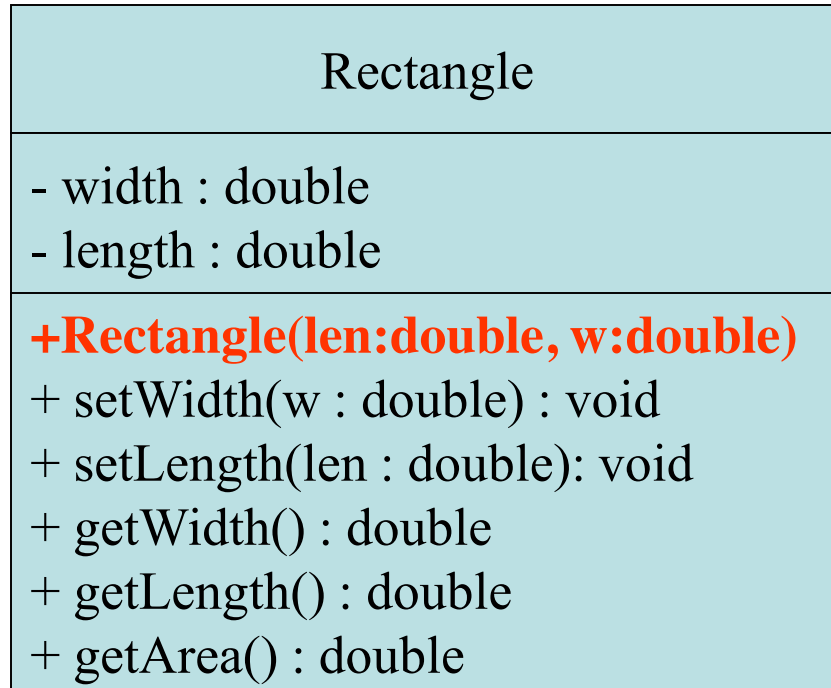
# Constructor for Rectangle Class

```
/**
 * Constructor
 * @param len The length of the rectangle.
 * @param w The width of the rectangle.
 */
public Rectangle(double len, double w)
{
 length = len;
 width = w;
}
```

Examples: [Rectangle.java](#), [ConstructorDemo.java](#)

# Constructors in UML

- In UML, the most common way constructors are defined is:



Notice there is no return type listed for constructors.



# Uninitialized Local Reference Variables

- Reference variables can be declared without being initialized.

```
Rectangle box;
```

- This statement does not create a `Rectangle` object, so it is an uninitialized local reference variable.
- A local reference variable must reference an object before it can be used, otherwise a compiler error will occur.

```
box = new Rectangle(7.0, 14.0);
```

- `box` will now reference a `Rectangle` object of length 7.0 and width 14.0.

# The Default Constructor

- When an object is created, its constructor is always called.
- If you do not write a constructor, Java provides one when the class is compiled. The constructor that Java provides is known as the *default constructor*.
  - It sets all of the object's numeric fields to 0.
  - It sets all of the object's `boolean` fields to `false`.
  - It sets all of the object's reference variables to the special value *null*.

# The Default Constructor

- The default constructor is a constructor with no parameters, used to initialize an object in a default configuration.
- The only time that Java provides a default constructor is when you do not write any constructor for a class.
  - See example: First version of [Rectangle.java](#)
- A default constructor is not provided by Java if a constructor is already written.
  - See example: [Rectangle.java](#) with Constructor

# Writing Your Own No-Arg Constructor

- A constructor that does not accept arguments is known as a *no-arg constructor*.
- The default constructor (provided by Java) is a no-arg constructor.
- We can write our own no-arg constructor

```
public Rectangle()
{
 length = 1.0;
 width = 1.0;
}
```

# The `String` Class Constructor

- One of the `String` class constructors accepts a string literal as an argument.
- This string literal is used to initialize a `String` object.
- For instance:

```
String name = new String("Michael Long");
```

# The `String` Class Constructor

- This creates a new reference variable *name* that points to a `String` object that represents the name “Michael Long”
- Because they are used so often, `String` objects can be created with a shorthand:

```
String name = "Michael Long";
```

# Passing Objects as Arguments

- When you pass a object as an argument, the thing that is passed into the parameter variable is the object's memory address.
- As a result, parameter variable references the object, and the receiving method has access to the object.
- See [DieArgument.java](#)

# Overloading Methods and Constructors

- Two or more methods in a class may have the same name as long as their parameter lists are different.
- When this occurs, it is called *method overloading*. This also applies to constructors.
- Method overloading is important because sometimes you need several different ways to perform the same operation.



# Overloaded Method `add`

```
public int add(int num1, int num2)
{
 int sum = num1 + num2;
 return sum;
}
```

```
public String add (String str1, String str2)
{
 String combined = str1 + str2;
 return combined;
}
```

# Method Signature and Binding

- A method signature consists of the method's name and the data types of the method's parameters, in the order that they appear. The return type is not part of the signature.

```
add(int, int)
add(String, String)
```

*Signatures of the  
add methods of  
previous slide*



- The process of matching a method call with the correct method is known as *binding*. The compiler uses the method signature to determine which version of the overloaded method to bind the call to.

# Rectangle Class Constructor Overload

If we were to add the no-arg constructor we wrote previously to our `Rectangle` class in addition to the original constructor we wrote, what would happen when we execute the following calls?

```
Rectangle box1 = new Rectangle();
Rectangle box2 = new Rectangle(5.0, 10.0);
```

# Rectangle Class Constructor Overload

If we were to add the no-arg constructor we wrote previously to our `Rectangle` class in addition to the original constructor we wrote, what would happen when we execute the following calls?

```
Rectangle box1 = new Rectangle();
Rectangle box2 = new Rectangle(5.0, 10.0);
```

The first call would use the no-arg constructor and `box1` would have a length of 1.0 and width of 1.0.

The second call would use the original constructor and `box2` would have a length of 5.0 and a width of 10.0.

# The BankAccount Example

[BankAccount.java](#)

[AccountTest.java](#)

Overloaded Constructors

Overloaded deposit methods

Overloaded withdraw methods

Overloaded setBalance methods

| BankAccount                       |
|-----------------------------------|
| -balance:double                   |
| +BankAccount()                    |
| +BankAccount(startBalance:double) |
| +BankAccount(str:String):         |
| +deposit(amount:double):void      |
| +deposit(str:String):void         |
| +withdraw(amount:double):void     |
| +withdraw(str:String):void        |
| +setBalance(b:double):void        |
| +setBalance(str:String):void      |
| +getBalance():double              |

# Scope of Instance Fields

- Variables declared as instance fields in a class can be accessed by any instance method in the same class as the field.
- If an instance field is declared with the `public` access specifier, it can also be accessed by code outside the class, as long as an instance of the class exists.

# Shadowing

- A parameter variable is, in effect, a local variable.
- Within a method, variable names must be unique.
- A method may have a local variable with the same name as an instance field.
- This is called *shadowing*.
- The local variable will *hide* the value of the instance field.
- Shadowing is discouraged and local variable names should not be the same as instance field names.

# Packages and `import` Statements

- Classes in the Java API are organized into *packages*.
- Explicit and Wildcard `import` statements
  - Explicit imports name a specific class
    - `import java.util.Scanner;`
  - Wildcard imports name a package, followed by an `*`
    - `import java.util.*;`
- The `java.lang` package is automatically made available to any Java class.



# Some Java Standard Packages

**Table 6-2** A few of the standard Java packages

| Package                    | Description                                                                                                                           |
|----------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| <code>java.applet</code>   | Provides the classes necessary to create an applet.                                                                                   |
| <code>java.awt</code>      | Provides classes for the Abstract Windowing Toolkit. These classes are used in drawing images and creating graphical user interfaces. |
| <code>java.io</code>       | Provides classes that perform various types of input and output.                                                                      |
| <code>java.lang</code>     | Provides general classes for the Java language. This package is automatically imported.                                               |
| <code>java.net</code>      | Provides classes for network communications.                                                                                          |
| <code>java.security</code> | Provides classes that implement security features.                                                                                    |
| <code>java.sql</code>      | Provides classes for accessing databases using structured query language.                                                             |
| <code>java.text</code>     | Provides various classes for formatting text.                                                                                         |
| <code>java.util</code>     | Provides various utility classes.                                                                                                     |
| <code>javax.swing</code>   | Provides classes for creating graphical user interfaces.                                                                              |

# Object Oriented Design

## Finding Classes and Their Responsibilities

- Finding the classes
  - Get written description of the problem domain
  - Identify all nouns, each is a potential class
  - Refine list to include only classes relevant to the problem
- Identify the responsibilities
  - Things a class is responsible for knowing
  - Things a class is responsible for doing
  - Refine list to include only classes relevant to the problem

# Object Oriented Design

## Finding Classes and Their Responsibilities

- Identify the responsibilities
  - Things a class is responsible for knowing
  - Things a class is responsible for doing
  - Refine list to include only classes relevant to the problem

## CHAPTER 7

# Arrays and the ArrayList Class

starting out with >>>

# JAVA™

From Control Structures through Objects

6TH EDITION



TONY GADDIS

# Chapter Topics

Chapter 7 discusses the following main topics:

- Introduction to Arrays
- Processing Array Contents
- Passing Arrays as Arguments to Methods
- Some Useful Array Algorithms and Operations
- Returning Arrays from Methods
- String Arrays
- Arrays of Objects

# Chapter Topics

Chapter 7 discusses the following main topics:

- The Sequential Search Algorithm
- Parallel Arrays
- Two-Dimensional Arrays
- Arrays with Three or More Dimensions
- The Selection Sort and the Binary Search
- Command-Line Arguments
- The `ArrayList` Class

# Introduction to Arrays

- Primitive variables are designed to hold only one value at a time.
- Arrays allow us to create a collection of like values that are indexed.
- An array can store any type of data but only one type of data at a time.
- An array is a list of data elements.

# Creating Arrays

- An array is an object so it needs an object reference.

```
// Declare a reference to an array that will hold integers.
int[] numbers;
```

- The next step creates the array and assigns its address to the `numbers` variable.

```
// Create a new array that will hold 6 integers.
numbers = new int[6];
```

|         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|
| 0       | 0       | 0       | 0       | 0       | 0       |
| index 0 | index 1 | index 2 | index 3 | index 4 | index 5 |

Array element values are initialized to 0.

Array indexes always start at 0.



# Creating Arrays

- It is possible to declare an array reference and create it in the same statement.

```
int[] numbers = new int[6];
```

- Arrays may be of any type.

```
float[] temperatures = new float[100];
```

```
char[] letters = new char[41];
```

```
long[] units = new long[50];
```

```
double[] sizes = new double[1200];
```

# Creating Arrays

- The array size must be a non-negative number.
- It may be a literal value, a constant, or variable.

```
final int ARRAY_SIZE = 6;
int[] numbers = new int[ARRAY_SIZE];
```

- Once created, an array size is fixed and cannot be changed.

# Accessing the Elements of an Array

|            |            |            |            |            |            |
|------------|------------|------------|------------|------------|------------|
| 20         | 0          | 0          | 0          | 0          | 0          |
| numbers[0] | numbers[1] | numbers[2] | numbers[3] | numbers[4] | numbers[5] |

- An array is accessed by:
  - the reference name
  - a subscript that identifies which element in the array to access.

`numbers[0] = 20; //pronounced "numbers sub zero"`

# Inputting and Outputting Array Elements

- Array elements can be treated as any other variable.
- They are simply accessed by the same name and a subscript.
- See example: [ArrayDemo1.java](#)
- Array subscripts can be accessed using variables (such as for loop counters).
- See example: [ArrayDemo2.java](#)

# Bounds Checking

- Array indexes always start at zero and continue to (array length - 1).

```
int values = new int[10];
```

- This array would have indexes 0 through 9.
- See example: [InvalidSubscript.java](#)
- In `for` loops, it is typical to use *i*, *j*, and *k* as counting variables.
  - It might help to think of *i* as representing the word *index*.

# Off-by-One Errors

- It is very easy to be off-by-one when accessing arrays.

```
// This code has an off-by-one error.
int[] numbers = new int[100];
for (int i = 1; i <= 100; i++)
 numbers[i] = 99;
```

- Here, the equal sign allows the loop to continue on to index 100, where 99 is the last index in the array.
- This code would throw an `ArrayIndexOutOfBoundsException`.

# Array Initialization

- When relatively few items need to be initialized, an initialization list can be used to initialize the array.

```
int[] days = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

- The numbers in the list are stored in the array in order:
  - days[0] is assigned 31,
  - days[1] is assigned 28,
  - days[2] is assigned 31,
  - days[3] is assigned 30,
  - etc.
- See example: [ArrayInitialization.java](#)

# Alternate Array Declaration

- Previously we showed arrays being declared:

```
int[] numbers;
```

- However, the brackets can also go here:

```
int numbers[];
```

- These are equivalent but the first style is typical.

- Multiple arrays can be declared on the same line.

```
int[] numbers, codes, scores;
```

- With the alternate notation each variable must have brackets.

```
int numbers[], codes[], scores;
```

- The `scores` variable in this instance is simply an `int` variable.



# Processing Array Contents

- Processing data in an array is the same as any other variable.

```
grossPay = hours[3] * payRate;
```

- Pre and post increment works the same:

```
int[] score = {7, 8, 9, 10, 11};
```

```
++score[2]; // Pre-increment operation
```

```
score[4]++; // Post-increment operation
```

- See example: [PayArray.java](#)

# Processing Array Contents

- Array elements can be used in relational operations:

```
if(cost[20] < cost[0])
{
 //statements
}
```

- They can be used as loop conditions:

```
while(value[count] != 0)
{
 //statements
}
```

# Array Length

- Arrays are objects and provide a public field named `length` that is a constant that can be tested.

```
double[] temperatures = new double[25];
```

- The length of this array is 25.

- The length of an array can be obtained via its `length` constant.

```
int size = temperatures.length;
```

- The variable `size` will contain 25.

# The Enhanced `for` Loop

- Simplified array processing (read only)
- Always goes through all elements
- General format:

```
for (datatype elementVariable : array)
 statement;
```

# The Enhanced `for` Loop

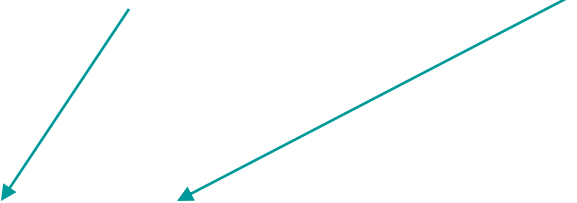
## Example:

```
int[] numbers = {3, 6, 9};
For(int val : numbers)
{
 System.out.println("The next value is " +
 val);
}
```

# Array Size

- The `length` constant can be used in a loop to provide automatic bounding.

Index subscripts start at 0 and end at one *less than* the array length.



```
for(int i = 0; i < temperatures.length; i++)
{
 System.out.println("Temperature " + i + ": "
 + temperatures[i]);
}
```

# Array Size

- You can let the user specify the size of an array:

```
int numTests;
int[] tests;
Scanner keyboard = new Scanner(System.in);
System.out.print("How many tests do you have? ");
numTests = keyboard.nextInt();
tests = new int[numTests];
```

- See example: [DisplayTestScores.java](#)

# Reassigning Array References

- An array reference can be assigned to another array of the same type.

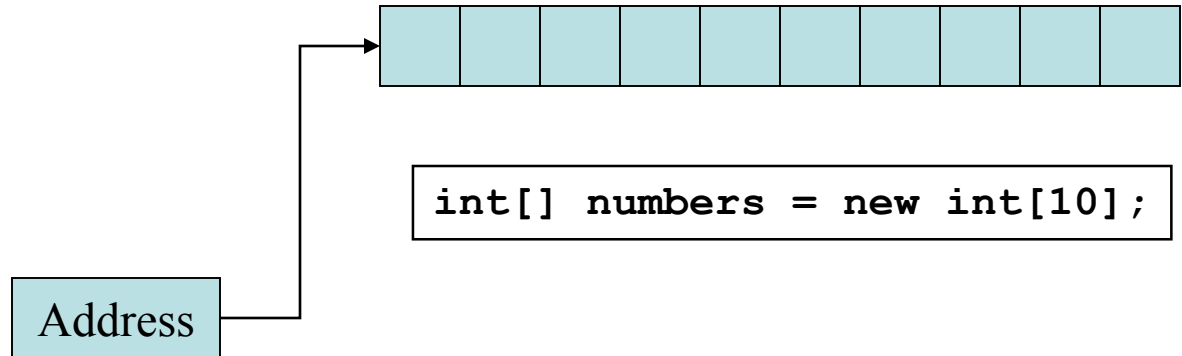
```
// Create an array referenced by the numbers variable.
int[] numbers = new int[10];
// Reassign numbers to a new array.
numbers = new int[5];
```

- If the first (10 element) array no longer has a reference to it, it will be garbage collected.



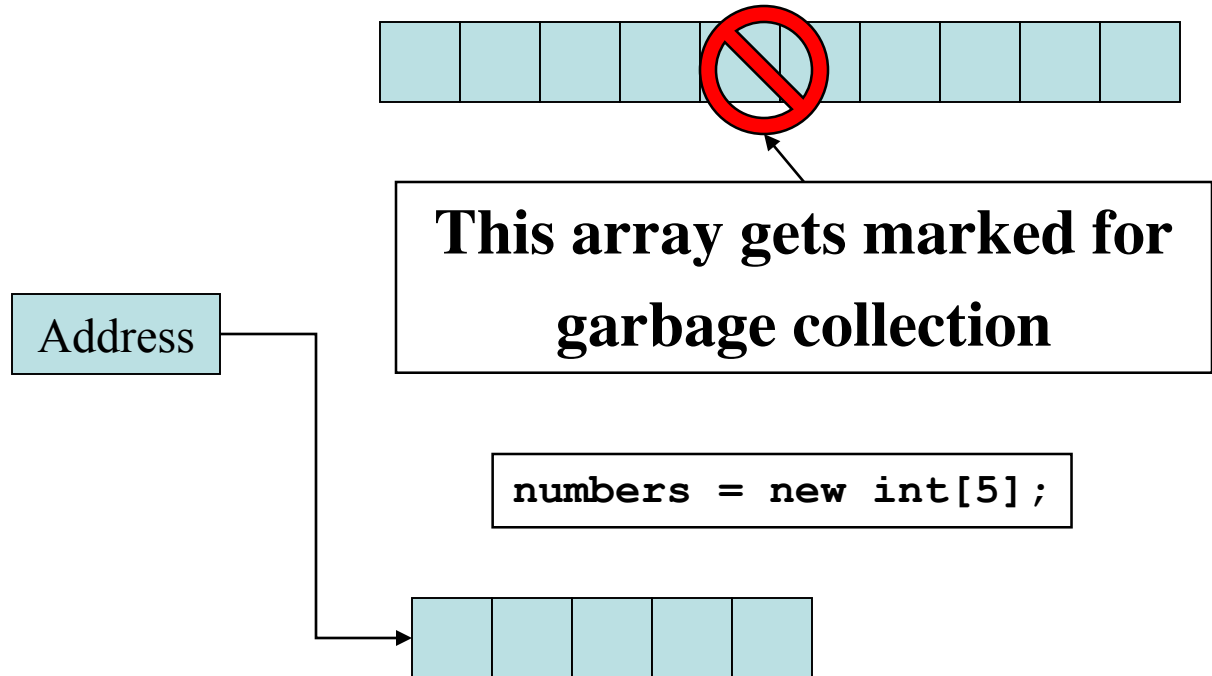
# Reassigning Array References

The `numbers` variable holds the address of an `int` array.



# Reassigning Array References

The `numbers` variable holds the address of an `int` array.

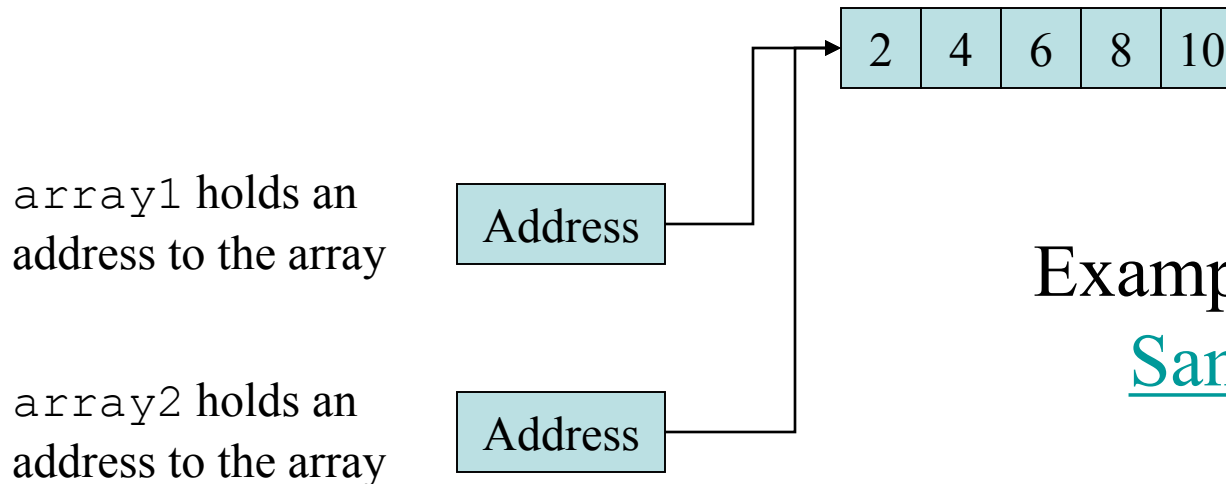


# Copying Arrays

- This is *not* the way to copy an array.

```
int[] array1 = { 2, 4, 6, 8, 10 };
```

```
int[] array2 = array1; // This does not copy array1.
```



Example:

[SameArray.java](#)

# Copying Arrays

- You cannot copy an array by merely assigning one reference variable to another.
- You need to copy the individual elements of one array to another.

```
int[] firstArray = {5, 10, 15, 20, 25 };
int[] secondArray = new int[5];
for (int i = 0; i < firstArray.length; i++)
 secondArray[i] = firstArray[i];
```

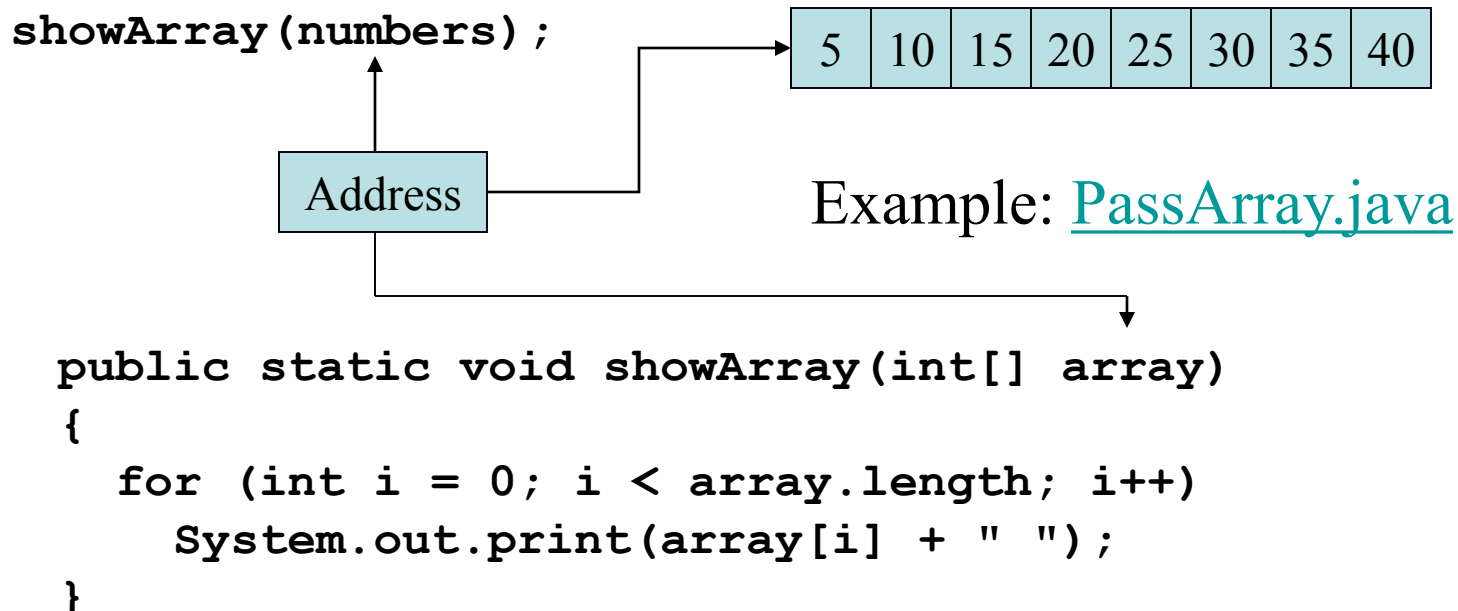
- This code copies each element of `firstArray` to the corresponding element of `secondArray`.

# Passing Array Elements to a Method

- When a single element of an array is passed to a method it is handled like any other variable.
- See example: [PassElements.java](#)
- More often you will want to write methods to process array data by passing the entire array, not just one element at a time.

# Passing Arrays as Arguments

- Arrays are objects.
- Their references can be passed to methods like any other object reference variable.



# Comparing Arrays

- The == operator determines only whether array references point to the same array object.

```
int[] firstArray = { 5, 10, 15, 20, 25 };
int[] secondArray = { 5, 10, 15, 20, 25 };

if (firstArray == secondArray) // This is a mistake.
 System.out.println("The arrays are the same.");
else
 System.out.println("The arrays are not the same.");
```

# Comparing Arrays: Example

```
int[] firstArray = { 2, 4, 6, 8, 10 };
int[] secondArray = { 2, 4, 6, 8, 10 };
boolean arraysEqual = true;
int i = 0;

// First determine whether the arrays are the same size.
if (firstArray.length != secondArray.length)
 arraysEqual = false;

// Next determine whether the elements contain the same data.
while (arraysEqual && i < firstArray.length)
{
 if (firstArray[i] != secondArray[i])
 arraysEqual = false;
 i++;
}

if (arraysEqual)
 System.out.println("The arrays are equal.");
else
 System.out.println("The arrays are not equal.");
```



# Useful Array Operations

- Finding the Highest Value

```
int [] numbers = new int[50];
int highest = numbers[0];
for (int i = 1; i < numbers.length; i++)
{
 if (numbers[i] > highest)
 highest = numbers[i];
}
```

- Finding the Lowest Value

```
int lowest = numbers[0];
for (int i = 1; i < numbers.length; i++)
{
 if (numbers[i] < lowest)
 lowest = numbers[i];
}
```

# Useful Array Operations

- Summing Array Elements:

```
int total = 0; // Initialize accumulator
for (int i = 0; i < units.length; i++)
 total += units[i];
```

- Averaging Array Elements:

```
double total = 0; // Initialize accumulator
double average; // Will hold the average
for (int i = 0; i < scores.length; i++)
 total += scores[i];
average = total / scores.length;
```

- Example: [SalesData.java](#), [Sales.java](#)

# Partially Filled Arrays

- Typically, if the amount of data that an array must hold is unknown:
  - size the array to the largest expected number of elements.
  - use a counting variable to keep track of how much valid data is in the array.

```
...
int[] array = new int[100];
int count = 0;
...
 System.out.print("Enter a number or -1 to quit: ");
 number = keyboard.nextInt();
 while (number != -1 && count <= 99)
 {
 array[count] = number;
 count++;
 System.out.print("Enter a number or -1 to quit: ");
 number = keyboard.nextInt();
 }
...
```

input, number and keyboard were previously declared and keyboard references a Scanner object

# Arrays and Files

- Saving the contents of an array to a file:

```
int[] numbers = {10, 20, 30, 40, 50};
```

```
PrintWriter outputFile =
 new PrintWriter ("Values.txt");
```

```
for (int i = 0; i < numbers.length; i++)
 outputFile.println(numbers[i]);
```

```
outputFile.close();
```

# Arrays and Files

- Reading the contents of a file into an array:

```
final int SIZE = 5; // Assuming we know the size.
int[] numbers = new int[SIZE];
int i = 0;
File file = new File ("Values.txt");
Scanner inputFile = new Scanner(file);
while (inputFile.hasNext() && i < numbers.length)
{
 numbers[i] = inputFile.nextInt();
 i++;
}
inputFile.close();
```

# Returning an Array Reference

- A method can return a reference to an array.
- The return type of the method must be declared as an array of the right type.

```
public static double[] getArray()
{
 double[] array = { 1.2, 2.3, 4.5, 6.7, 8.9 };
 return array;
}
```

- The `getArray` method is a public static method that returns an array of doubles.
- See example: [ReturnArray.java](#)

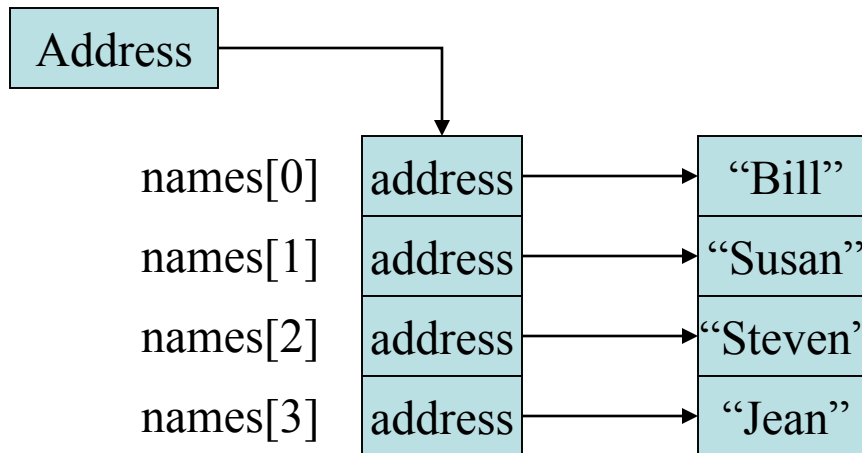
# String Arrays

- Arrays are not limited to primitive data.
- An array of `String` objects can be created:

```
String[] names = { "Bill", "Susan", "Steven", "Jean" };
```

The `names` variable holds the address to the array.

A `String` array is an array of references to `String` objects.



Example:

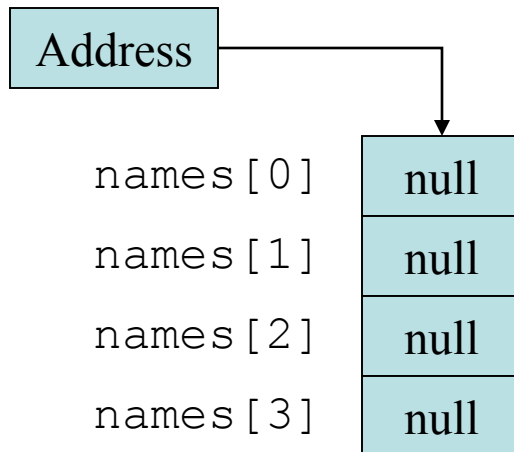
[MonthDays.java](#)

# String Arrays

- If an initialization list is not provided, the new keyword must be used to create the array:

```
String[] names = new String[4];
```

The names variable holds  
the address to the array.



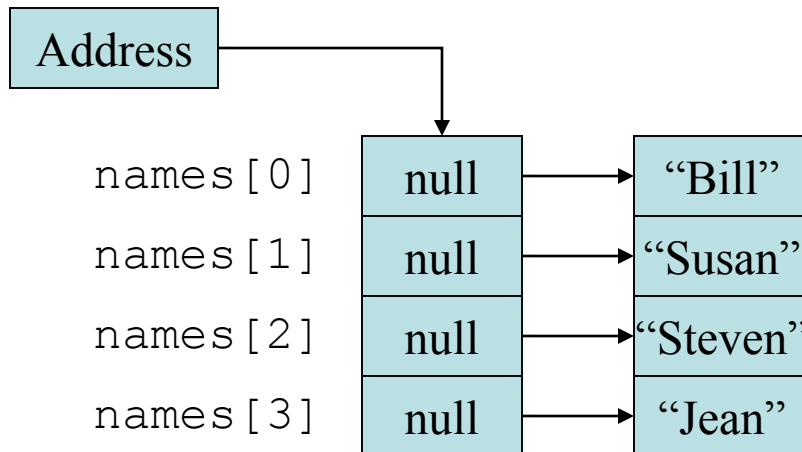


# String Arrays

- When an array is created in this manner, each element of the array must be initialized.

The `names` variable holds the address to the array.

```
names[0] = "Bill";
names[1] = "Susan";
names[2] = "Steven";
names[3] = "Jean";
```



# Calling `String` Methods On Array Elements

- `String` objects have several methods, including:
  - `toUpperCase`
  - `compareTo`
  - `equals`
  - `charAt`
- Each element of a `String` array is a `String` object.
- Methods can be used by using the array name and index as before.

```
System.out.println(names[0].toUpperCase());
char letter = names[3].charAt(0);
```

# The length Field & The length Method

- Arrays have a **final field** named `length`.
- String objects have a **method** named `length`.
- To display the length of each string held in a `String` array:

```
for (int i = 0; i < names.length; i++)
 System.out.println(names[i].length());
```

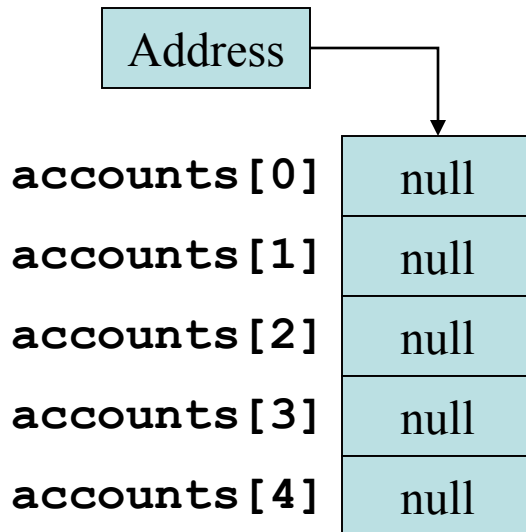
- An array's length is a **field**
  - You do not write a set of parentheses after its name.
- A `String`'s length is a **method**
  - You do write the parentheses after the name of the `String` class's `length` method.

# Arrays of Objects

- Because `Strings` are objects, we know that arrays can contain objects.

```
BankAccount[] accounts = new BankAccount[5];
```

The `accounts` variable holds the address of an `BankAccount` array.



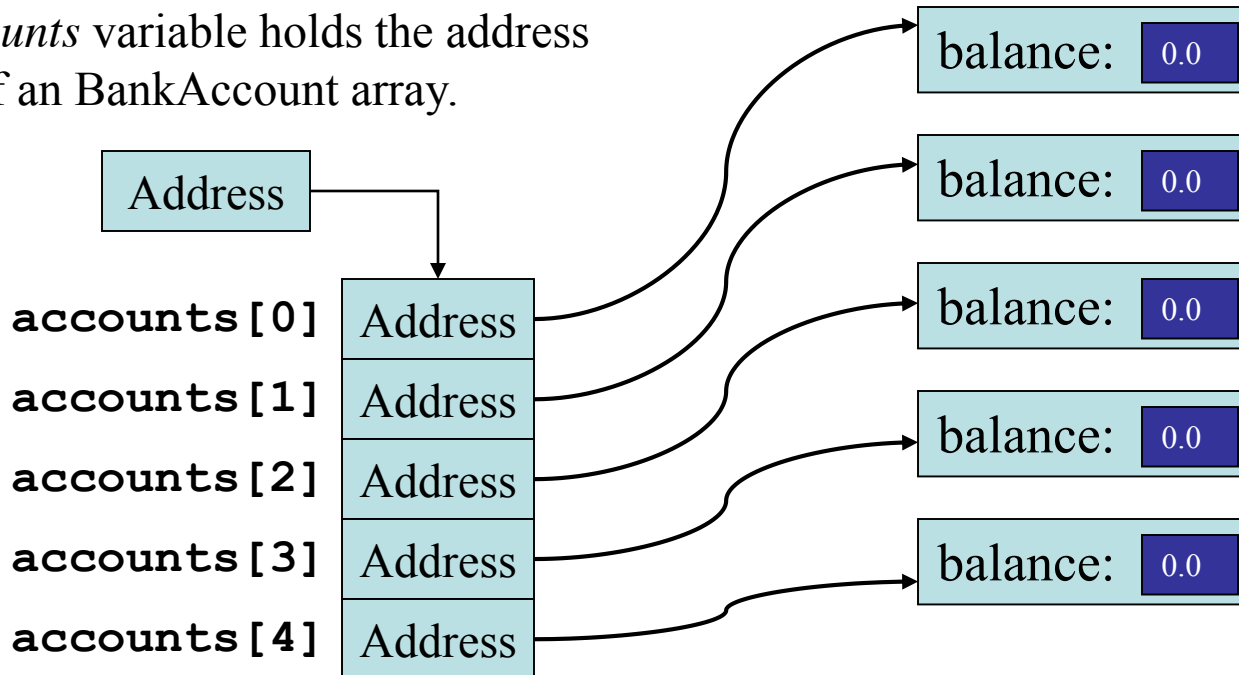
The array is an array of references to `BankAccount` objects.

# Arrays of Objects

- Each element needs to be initialized.  

```
for (int i = 0; i < accounts.length; i++)
 accounts[i] = new BankAccount();
```
- See example: [ObjectArray.java](#)

The *accounts* variable holds the address of an BankAccount array.



# The Sequential Search Algorithm

- A search algorithm is a method of locating a specific item in a larger collection of data.
- The *sequential search algorithm* uses a loop to:
  - sequentially step through an array,
  - compare each element with the search value, and
  - stop when
    - the value is found or
    - the end of the array is encountered.
- See example: [SearchArray.java](#)

# Two-Dimensional Arrays

- A two-dimensional array is an array of arrays.
- It can be thought of as having rows and columns.

|       | column 0 | column 1 | column 2 | column 3 |
|-------|----------|----------|----------|----------|
| row 0 |          |          |          |          |
| row 1 |          |          |          |          |
| row 2 |          |          |          |          |
| row 3 |          |          |          |          |

# Two-Dimensional Arrays

- Declaring a two-dimensional array requires two sets of brackets and two size declarators
  - The first one is for the number of rows
  - The second one is for the number of columns.

```
double[][] scores = new double[3][4];
```



- The two sets of brackets in the data type indicate that the scores variable will reference a two-dimensional array.
- Notice that each size declarator is enclosed in its own set of brackets.



# Accessing Two-Dimensional Array Elements

- When processing the data in a two-dimensional array, each element has two subscripts:
  - one for its row and
  - another for its column.

# Accessing Two-Dimensional Array Elements

The `scores` variable holds the address of a 2D array of `doubles`.

|         |       | column 0     | column 1     | column 2     | column 3     |
|---------|-------|--------------|--------------|--------------|--------------|
| Address |       | scores[0][0] | scores[0][1] | scores[0][2] | scores[0][3] |
|         | row 0 | scores[0][0] | scores[0][1] | scores[0][2] | scores[0][3] |
|         | row 1 | scores[1][0] | scores[1][1] | scores[1][2] | scores[1][3] |
|         | row 2 | scores[2][0] | scores[2][1] | scores[2][2] | scores[2][3] |

# Accessing Two-Dimensional Array Elements

Accessing one of the elements in a two-dimensional array requires the use of both subscripts.

The `scores` variable holds the address of a 2D array of `doubles`.

```
scores[2][1] = 95;
```

| Address |  | column 0 | column 1 | column 2 | column 3 |
|---------|--|----------|----------|----------|----------|
| row 0   |  | 0        | 0        | 0        | 0        |
| row 1   |  | 0        | 0        | 0        | 0        |
| row 2   |  | 0        | 95       | 0        | 0        |

# Accessing Two-Dimensional Array Elements

- Programs that process two-dimensional arrays can do so with nested loops.
- To fill the scores array:

```
for (int row = 0; row < 3; row++)
{
 for (int col = 0; col < 4; col++)
 {
 System.out.print("Enter a score: ");
 scores[row][col] = keyboard.nextDouble();
 }
}
```

Number of rows, not the largest subscript

Number of columns, not the largest subscript

keyboard references a Scanner object

# Accessing Two-Dimensional Array Elements

- To print out the `scores` array:

```
for (int row = 0; row < 3; row++)
{
 for (int col = 0; col < 4; col++)
 {
 System.out.println(scores[row][col]);
 }
}
```

- See example: [CorpSales.java](#)

# Initializing a Two-Dimensional Array

- Initializing a two-dimensional array requires enclosing each row's initialization list in its own set of braces.

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

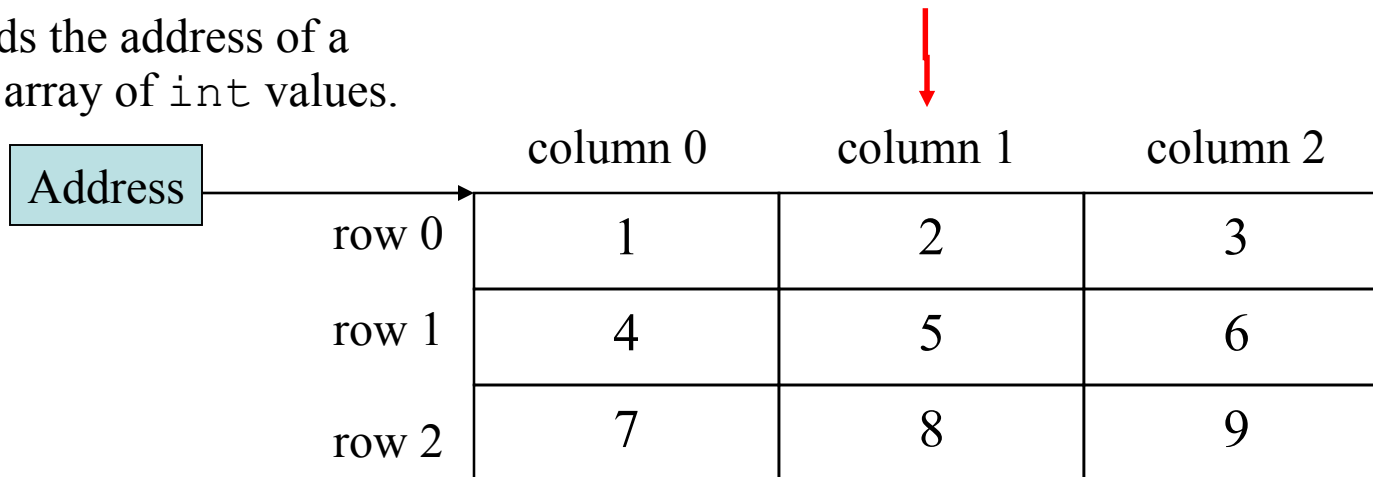
- Java automatically creates the array and fills its elements with the initialization values.
  - row 0    {1, 2, 3}
  - row 1    {4, 5, 6}
  - row 2    {7, 8, 9}
- Declares an array with three rows and three columns.

# Initializing a Two-Dimensional Array

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

produces:

The `numbers` variable holds the address of a 2D array of `int` values.



|       | column 0 | column 1 | column 2 |
|-------|----------|----------|----------|
| row 0 | 1        | 2        | 3        |
| row 1 | 4        | 5        | 6        |
| row 2 | 7        | 8        | 9        |

# The length Field

- Two-dimensional arrays are arrays of one-dimensional arrays.
- The length field of the array gives the number of rows in the array.
- Each row has a length constant tells how many columns is in that row.
- Each row can have a different number of columns.



# The length Field

- To access the length fields of the array:

```
int[][] numbers = { { 1, 2, 3, 4 },
 { 5, 6, 7 },
 { 9, 10, 11, 12 } };
```

```
for (int row = 0; row < numbers.length; row++)
{
 for (int col = 0; col < numbers[row].length; col++)
 System.out.println(numbers[row][col]);
}
```

Number of rows

Number of columns in this row.

- See example: [Lengths.java](#)

The array can have variable length rows.

# Summing The Elements of a Two-Dimensional Array

```
int[][] numbers = { { 1, 2, 3, 4 },
 {5, 6, 7, 8},
 {9, 10, 11, 12} };

int total;
total = 0;
for (int row = 0; row < numbers.length; row++)
{
 for (int col = 0; col < numbers[row].length; col++)
 total += numbers[row][col];
}

System.out.println("The total is " + total);
```

# Summing The Rows of a Two-Dimensional Array

```
int[][] numbers = {{ 1, 2, 3, 4},
 {5, 6, 7, 8},
 {9, 10, 11, 12}};

int total;

for (int row = 0; row < numbers.length; row++)
{
 total = 0;
 for (int col = 0; col < numbers[row].length; col++)
 total += numbers[row][col];
 System.out.println("Total of row "
 + row + " is " + total);
}
```

# Summing The Columns of a Two-Dimensional Array

```
int[][] numbers = {{1, 2, 3, 4},
 {5, 6, 7, 8},
 {9, 10, 11, 12}};

int total;

for (int col = 0; col < numbers[0].length; col++)
{
 total = 0;
 for (int row = 0; row < numbers.length; row++)
 total += numbers[row][col];
 System.out.println("Total of column "
 + col + " is " + total);
}
```

# Passing and Returning Two-Dimensional Array References

- There is no difference between passing a single or two-dimensional array as an argument to a method.
- The method must accept a two-dimensional array as a parameter.
- See example: [Pass2Darray.java](#)

# Ragged Arrays

- When the rows of a two-dimensional array are of different lengths, the array is known as a *ragged array*.
- You can create a ragged array by creating a two-dimensional array with a specific number of rows, but no columns.

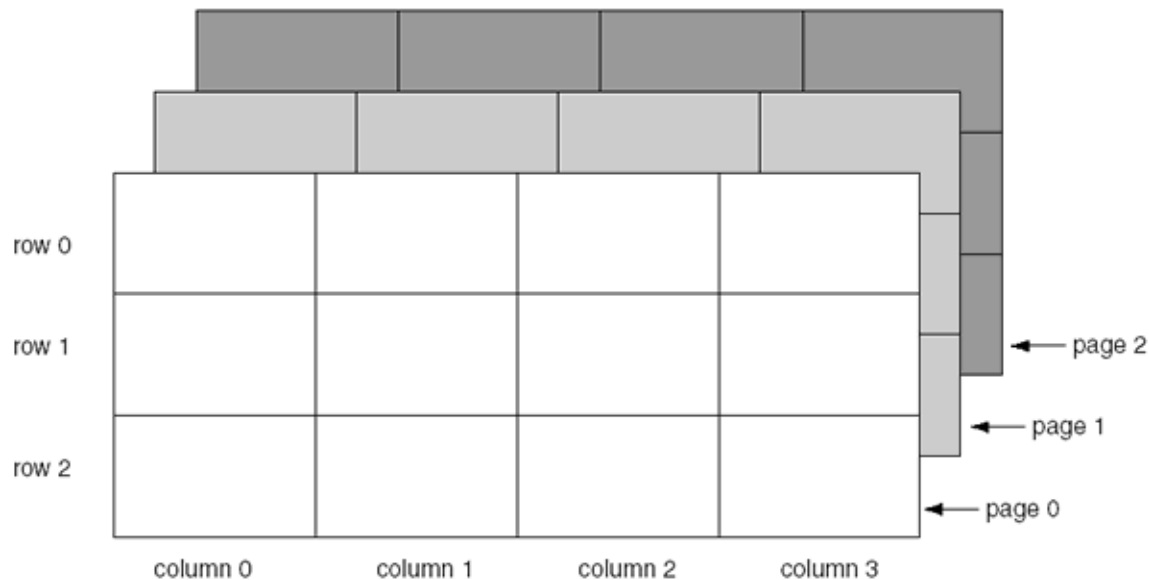
```
int [][] ragged = new int [4][];
```

- Then create the individual rows.

```
ragged[0] = new int [3];
ragged[1] = new int [4];
ragged[2] = new int [5];
ragged[3] = new int [6];
```

# More Than Two Dimensions

- Java does not limit the number of dimensions that an array may be.
- More than three dimensions is hard to visualize, but can be useful in some programming problems.



# Selection Sort

- In a selection sort:
  - The smallest value in the array is located and moved to element 0.
  - Then the next smallest value is located and moved to element 1.
  - This process continues until all of the elements have been placed in their proper order.
  - See example: [SelectionSortDemo.java](#)



# Binary Search

- A binary search:
  - requires an array sorted in ascending order.
  - starts with the element in the middle of the array.
  - If that element is the desired value, the search is over.
  - Otherwise, the value in the middle element is either greater or less than the desired value
  - If it is greater than the desired value, search in the first half of the array.
  - Otherwise, search the last half of the array.
  - Repeat as needed while adjusting start and end points of the search.
- See example: [BinarySearchDemo.java](#)

# Command-Line Arguments

- A Java program can receive arguments from the operating system command-line.
- The `main` method has a header that looks like this:

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

- The `main` method receives a `String` array as a parameter.
- The array that is passed into the `args` parameter comes from the operating system command-line.

# Command-Line Arguments

- To run the example:

```
java CommandLine How does this work?
```

```
args[0] is assigned "How"
```

```
args[1] is assigned "does"
```

```
args[2] is assigned "this"
```

```
args[3] is assigned "work?"
```

- Example: [CommandLine.java](#)
- It is not required that the name of `main`'s parameter array be `args`.

# Variable-Length Argument Lists

- Special type parameter – vararg...
  - Vararg parameters are actually arrays
  - Examples: [VarArgsDemo1.java](#), [VarargsDemo2.java](#)

```
public static int sum(int... numbers)
{
 int total = 0; // Accumulator
 // Add all the values in the numbers array.
 for (int val : numbers)
 total += val;
 // Return the total.
 return total;
}
```

# The ArrayList Class

- Similar to an array, an ArrayList allows object storage
- Unlike an array, an ArrayList object:
  - Automatically expands when a new item is added
  - Automatically shrinks when items are removed
- Requires:

```
import java.util.ArrayList;
```

# Creating an ArrayList

```
ArrayList<String> nameList = new ArrayList<String>();
```

Two red arrows originate from below the code line. One arrow points to the 'String' type inside the first angle bracket of 'ArrayList<String>'. The other arrow points to the 'String' type inside the second angle bracket of 'new ArrayList<String>()'.

Notice the word `String` written inside angled brackets `<>`

This specifies that the `ArrayList` can hold `String` objects.

If we try to store any other type of object in this `ArrayList`, an error will occur.

# Using an ArrayList

- To populate the ArrayList, use the add method:
  - `nameList.add("James");`
  - `nameList.add("Catherine");`
- To get the current size, call the size method
  - `nameList.size();` // returns 2

# Using an ArrayList

- To access items in an ArrayList, use the get method  
`nameList.get(1);`

In this statement 1 is the index of the item to get.

- Example: [ArrayListDemo1.java](#)



# Using an ArrayList

- The ArrayList class's toString method returns a string representing all items in the ArrayList  
`System.out.println(nameList);`  
This statement yields :  
[ James, Catherine ]
- The ArrayList class's remove method removes designated item from the ArrayList  
`nameList.remove(1);`  
This statement removes the second item.
- See example: [ArrayListDemo3.java](#)

# Using an ArrayList

- The `ArrayList` class's `add` method with one argument adds new items to the end of the `ArrayList`
- To insert items at a location of choice, use the `add` method with two arguments:

```
nameList.add(1, "Mary");
```

This statement inserts the `String` "Mary" at index 1

- To replace an existing item, use the `set` method:

```
nameList.set(1, "Becky");
```

This statement replaces "Mary" with "Becky"

- See example: [ArrayListDemo5.java](#)

# Using an ArrayList

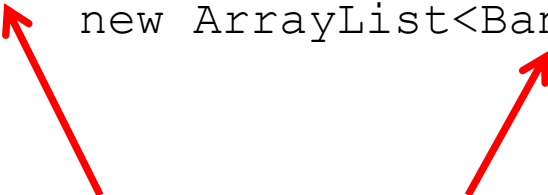
- An ArrayList has a capacity, which is the number of items it can hold without increasing its size.
- The default capacity of an ArrayList is 10 items.
- To designate a different capacity, use a parameterized constructor:

```
ArrayList<String> list = new ArrayList<String>(100);
```

# Using an ArrayList

- You can store any type of *object* in an ArrayList

```
ArrayList<BankAccount> accountList =
 new ArrayList<BankAccount>();
```



This creates an ArrayList that can hold  
BankAccount objects.

# Using an ArrayList

```
// Create an ArrayList to hold BankAccount objects.
ArrayList<BankAccount> list = new ArrayList<BankAccount>();

// Add three BankAccount objects to the ArrayList.
list.add(new BankAccount(100.0));
list.add(new BankAccount(500.0));
list.add(new BankAccount(1500.0));

// Display each item.
for (int index = 0; index < list.size(); index++)
{
 BankAccount account = list.get(index);
 System.out.println("Account at index " + index +
 "\nBalance: " + account.getBalance());
}
```

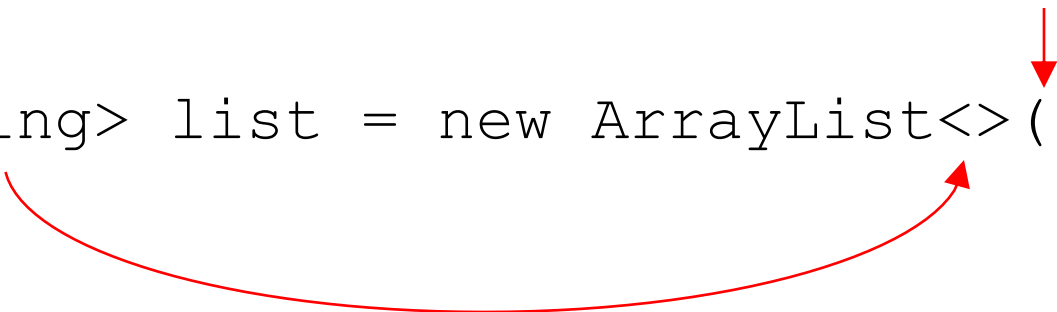
See: [ArrayListDemo6.java](#)

# Using an ArrayList

- The diamond operator
  - Beginning in Java 7, you can use the <> operator for simpler ArrayList declarations:

No need to specify the data type here.

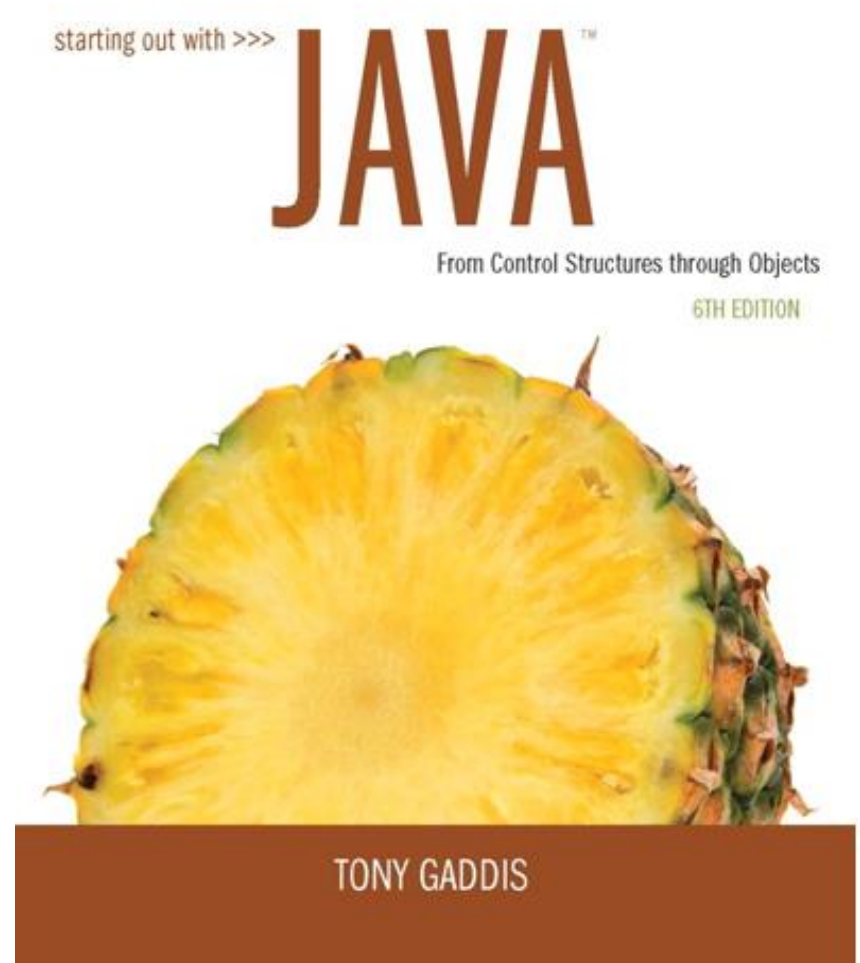
```
ArrayList<String> list = new ArrayList<>();
```



Java infers the type of the ArrayList object from the variable declaration.

# CHAPTER 8

## A Second Look at Classes and Objects



# Chapter Topics

Chapter 8 discusses the following main topics:

- Static Class Members
- Passing Objects as Arguments to Methods
- Returning Objects from Methods
- The `toString` method
- Writing an `equals` Method
- Methods that Copy Objects



# Chapter Topics

Chapter 8 discusses the following main topics:

- Aggregation
- The `this` Reference Variable
- Enumerated Types
- Garbage Collection
- Focus on Object-Oriented Design: Class Collaboration

# Review of Instance Fields and Methods

- Each instance of a class has its own copy of instance variables.
  - Example:
    - The `Rectangle` class defines a `length` and a `width` field.
    - Each instance of the `Rectangle` class can have different values stored in its `length` and `width` fields.
- Instance methods require that an instance of a class be created in order to be used.
- Instance methods typically interact with instance fields or calculate values based on those fields.

# Static Class Members

- *Static fields* and *static methods* do not belong to a single instance of a class.
- To invoke a static method or use a static field, the class name, rather than the instance name, is used.
- Example:

```
double val = Math.sqrt(25.0);
```



Class name

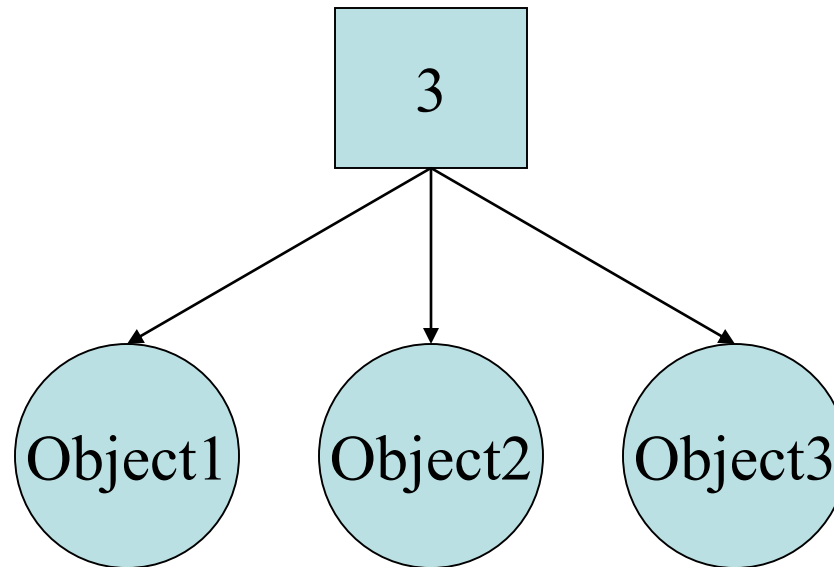
Static method

# Static Fields

- Class fields are declared using the `static` keyword between the access specifier and the field type.  
**`private static int instanceCount = 0;`**
- The field is initialized to 0 only once, regardless of the number of times the class is instantiated.
  - Primitive static fields are initialized to 0 if no initialization is performed.
- Examples: [Countable.java](#), [StaticDemo.java](#)

# Static Fields

instanceCount field  
(static)



# Static Methods

- Methods can also be declared static by placing the `static` keyword between the access modifier and the return type of the method.

```
public static double milesToKilometers(double miles)
{...}
```

- When a class contains a static method, it is not necessary to create an instance of the class in order to use the method.

```
double kilosPerMile = Metric.milesToKilometers(1.0);
```

- Examples: [Metric.java](#), [MetricDemo.java](#)

# Static Methods

- Static methods are convenient because they may be called at the class level.
- They are typically used to create utility classes, such as the `Math` class in the Java Standard Library.
- Static methods may not communicate with instance fields, only static fields.

# Passing Objects as Arguments

- Objects can be passed to methods as arguments.
- Java passes all arguments *by value*.
- When an object is passed as an argument, the value of the reference variable is passed.
- The value of the reference variable is an address or reference to the object in memory.
- A *copy* of the object is *not passed*, just a pointer to the object.
- When a method receives a reference variable as an argument, it is possible for the method to modify the contents of the object referenced by the variable.



# Passing Objects as Arguments

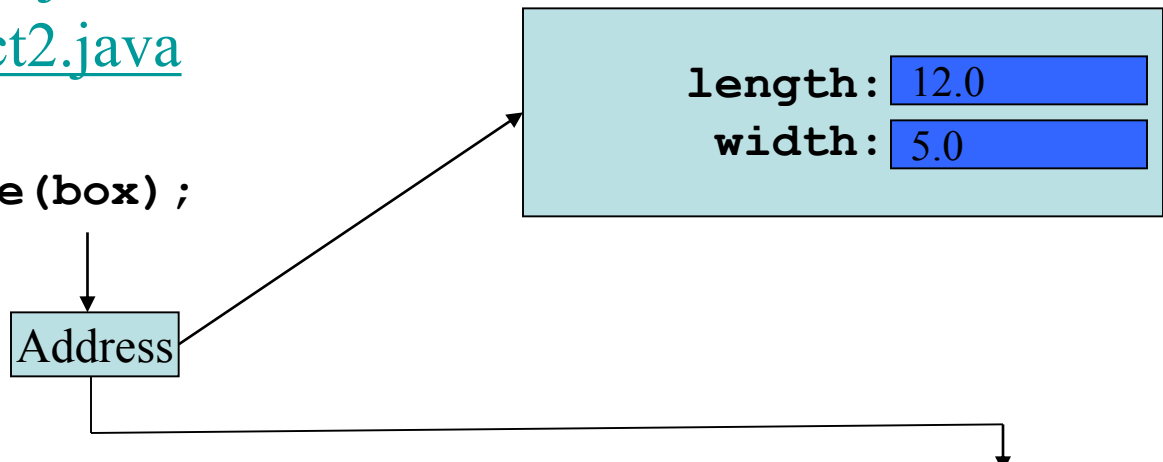
Examples:

[PassObject.java](#)

[PassObject2.java](#)

```
displayRectangle(box);
```

Address



A Rectangle object

length: 12.0

width: 5.0

```
public static void displayRectangle(Rectangle r)
{
 // Display the length and width.
 System.out.println("Length: " + r.getLength() +
 " Width: " + r.getWidth());
}
```

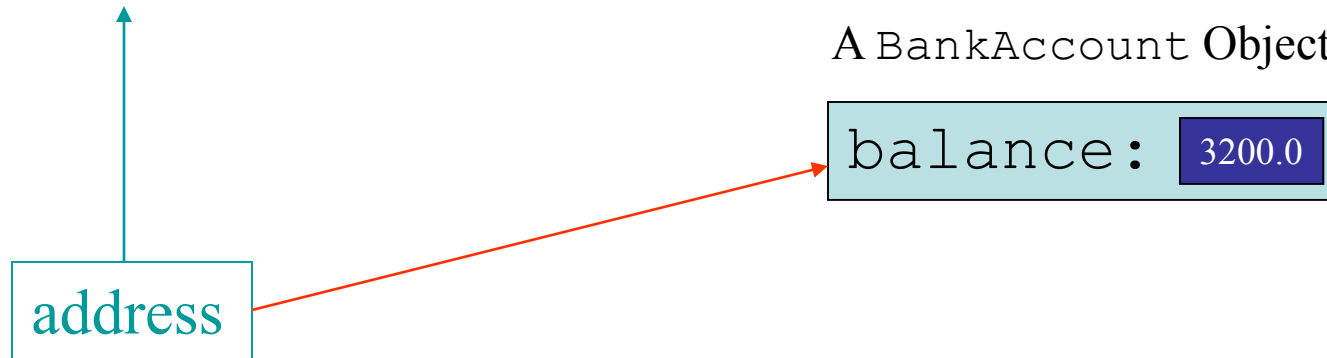
# Returning Objects From Methods

- Methods are not limited to returning the primitive data types.
- Methods can return references to objects as well.
- Just as with passing arguments, a copy of the object is **not** returned, only its address.
- See example: [ReturnObject.java](#)
- Method return type:

```
public static BankAccount getAccount()
{
 ...
 return new BankAccount(balance);
}
```

# Returning Objects from Methods

```
account = getAccount();
```



```
public static BankAccount getAccount()
{
 ...
 return new BankAccount(balance);
}
```

# The toString Method

- The toString method of a class can be called *explicitly*:

```
Stock xyzCompany = new Stock ("XYZ", 9.62);
System.out.println(xyzCompany.toString());
```

- However, the toString method does not have to be called explicitly but is called implicitly whenever you pass an object of the class to println or print.

```
Stock xyzCompany = new Stock ("XYZ", 9.62);
System.out.println(xyzCompany);
```

# The toString method

- The toString method is also called implicitly whenever you concatenate an object of the class with a string.

```
Stock xyzCompany = new Stock ("XYZ", 9.62);
System.out.println("The stock data is:\n" +
 xyzCompany);
```

# The toString Method

- All objects have a toString method that returns the class name and a hash of the memory address of the object.
- We can override the default method with our own to print out more useful information.
- Examples: [Stock.java](#), [StockDemo1.java](#)

# The `equals` Method

- When the `==` operator is used with reference variables, the memory address of the objects are compared.
- The contents of the objects are not compared.
- All objects have an `equals` method.
- The default operation of the `equals` method is to compare memory addresses of the objects (just like the `==` operator).

# The equals Method

- The Stock class has an equals method.
- If we try the following:

```
Stock stock1 = new Stock("GMX", 55.3);
Stock stock2 = new Stock("GMX", 55.3);
if (stock1 == stock2) // This is a mistake.
 System.out.println("The objects are the same.");
else
 System.out.println("The objects are not the same.");
```

only the addresses of the objects are compared.



# The equals Method

- Instead of using the `==` operator to compare two `Stock` objects, we should use the `equals` method.

```
public boolean equals(Stock object2)
{
 boolean status;

 if(symbol.equals(Object2.symbol && sharePrice == Object2.sharePrice)
 status = true;
 else
 status = false;
 return status;
}
```

- Now, objects can be compared by their contents rather than by their memory addresses.
- See example: [StockCompare.java](#)

# Methods That Copy Objects

- There are two ways to copy an object.
  - You cannot use the assignment operator to copy reference types
  - Reference only copy
    - This is simply copying the address of an object into another reference variable.
  - Deep copy (correct)
    - This involves creating a new instance of the class and copying the values from one object into the new object.
  - Example: [ObjectCopy.java](#)

# Copy Constructors

- A copy constructor accepts an existing object of the same class and clones it

```
public Stock(Stock object 2)
{
 symbol = object2.symbol;
 sharePrice = object2.sharePrice;
}

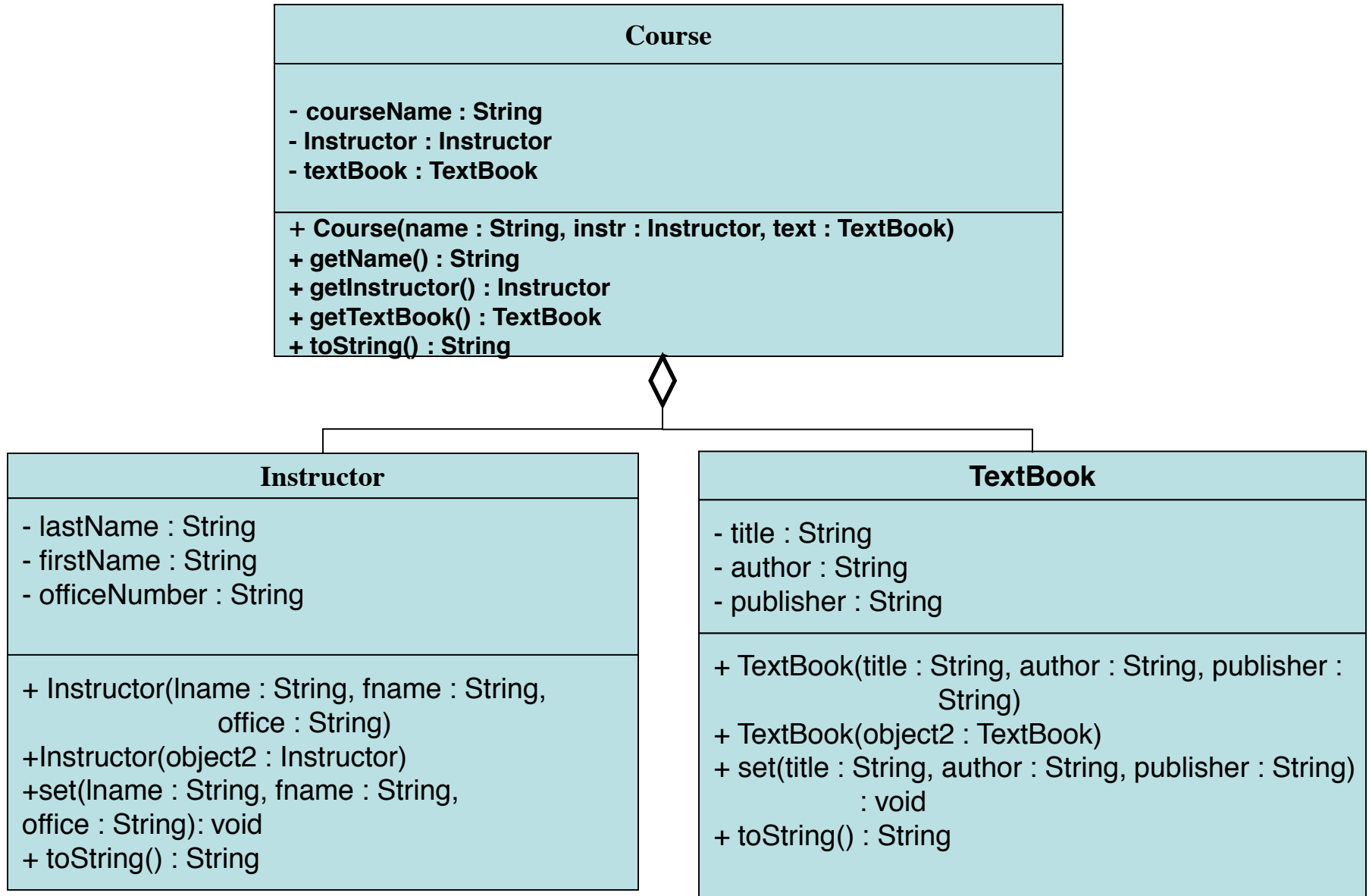
// Create a Stock object
Stock company1 = new Stock("XYZ", 9.62);

//Create company2, a copy of company1
Stock company2 = new Stock(company1);
```

# Aggregation

- Creating an instance of one class as a reference in another class is called *object aggregation*.
- Aggregation creates a “has a” relationship between objects.
- Examples:
  - [Instructor.java](#), [Textbook.java](#), [Course.java](#),  
[CourseDemo.java](#)

# Aggregation in UML Diagrams



# Returning References to Private Fields

- Avoid returning references to private data elements.
- Returning references to private variables will allow any object that receives the reference to modify the variable.

# Null References

- A *null reference* is a reference variable that points to nothing.
- If a reference is null, then no operations can be performed on it.
- References can be tested to see if they point to null prior to being used.

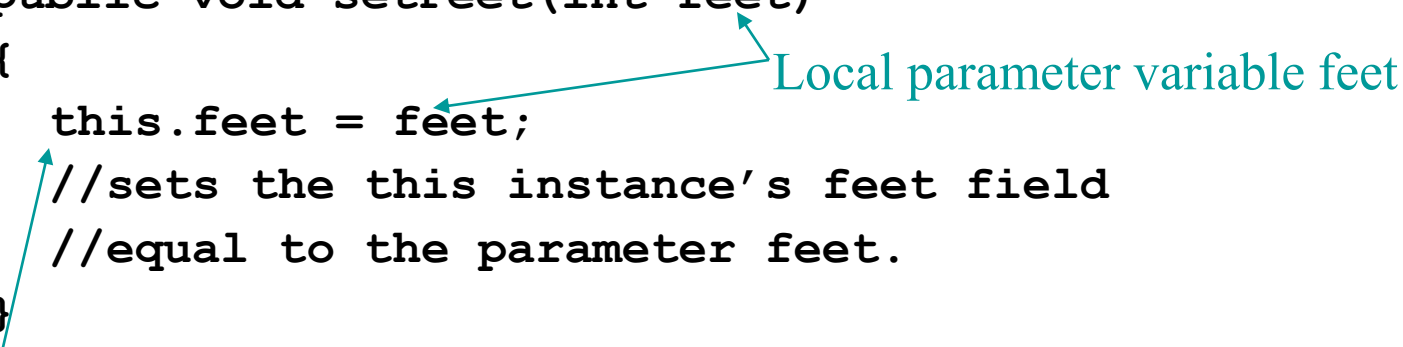
```
if(name != null)
{
 System.out.println("Name is: "
 + name.toUpperCase());
}
```

- Examples: [FullName.java](#), [NameTester.java](#)

# The `this` Reference

- The `this` reference is simply a name that an object can use to refer to itself.
- The `this` reference can be used to overcome shadowing and allow a parameter to have the same name as an instance field.

```
public void setFeet(int feet)
{
 this.feet = feet;
 //sets the this instance's feet field
 //equal to the parameter feet.
}
```

A diagram illustrating variable shadowing. A teal arrow points from the text 'Local parameter variable feet' to the parameter 'feet' in the method signature. Another teal arrow points from the text 'Shadowed instance variable' to the 'this' keyword in the assignment statement 'this.feet = feet;'.

Local parameter variable feet

Shadowed instance variable



# The `this` Reference

- The `this` reference can be used to call a constructor from another constructor.

```
public Stock(String sym)
{
 this(sym, 0.0);
}
```

- This constructor would allow an instance of the `Stock` class to be created using only the symbol name as a parameter.
- It calls the constructor that takes the symbol and the price, using *sym* as the symbol argument and 0 as the price argument.
- Elaborate constructor chaining can be created using this technique.
- If `this` is used in a constructor, it must be the first statement in the constructor.

# Enumerated Types

- Known as an enum, requires declaration and definition like a class

- Syntax:

```
enum typeName { one or more enum constants }
```

- Definition:

```
enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
 FRIDAY, SATURDAY }
```

- Declaration:

```
Day WorkDay; // creates a Day enum
```

- Assignment:

```
Day WorkDay = Day.WEDNESDAY;
```

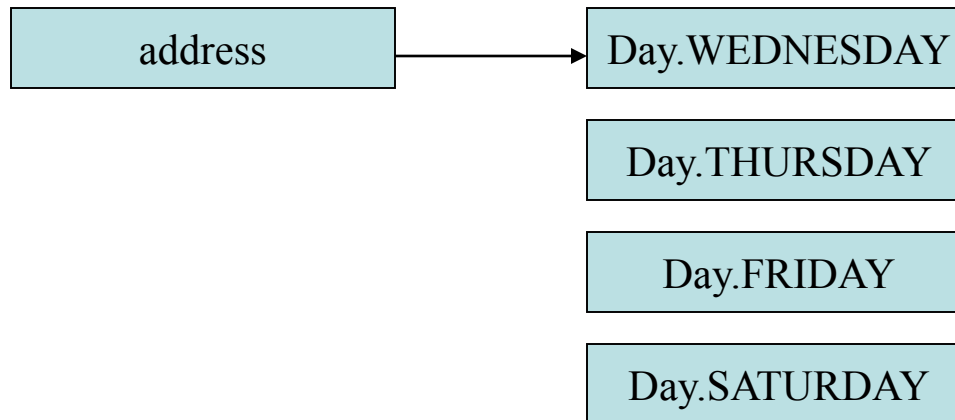
# Enumerated Types

- An enum is a specialized class

Each are objects of type `Day`, a specialized class

```
Day workDay = Day.WEDNESDAY;
```

The `workDay` variable holds the address of the `Day.WEDNESDAY` object



# Enumerated Types - Methods

- `toString` – returns name of calling constant
- `ordinal` – returns the zero-based position of the constant in the enum. For example the ordinal for `Day.THURSDAY` is 4
- `equals` – accepts an object as an argument and returns true if the argument is equal to the calling enum constant
- `compareTo` - accepts an object as an argument and returns a negative integer if the calling constant's ordinal < than the argument's ordinal, a positive integer if the calling constant's ordinal > than the argument's ordinal and zero if the calling constant's ordinal == the argument's ordinal.
- Examples: [EnumDemo.java](#), [CarType.java](#), [SportsCar.java](#), [SportsCarDemo.java](#)

# Enumerated Types - Switching

- Java allows you to test an enum constant with a `switch` statement.

Example: [SportsCarDemo2.java](#)

# Garbage Collection

- When objects are no longer needed they should be destroyed.
- This frees up the memory that they consumed.
- Java handles all of the memory operations for you.
- Simply set the reference to *null* and Java will reclaim the memory.

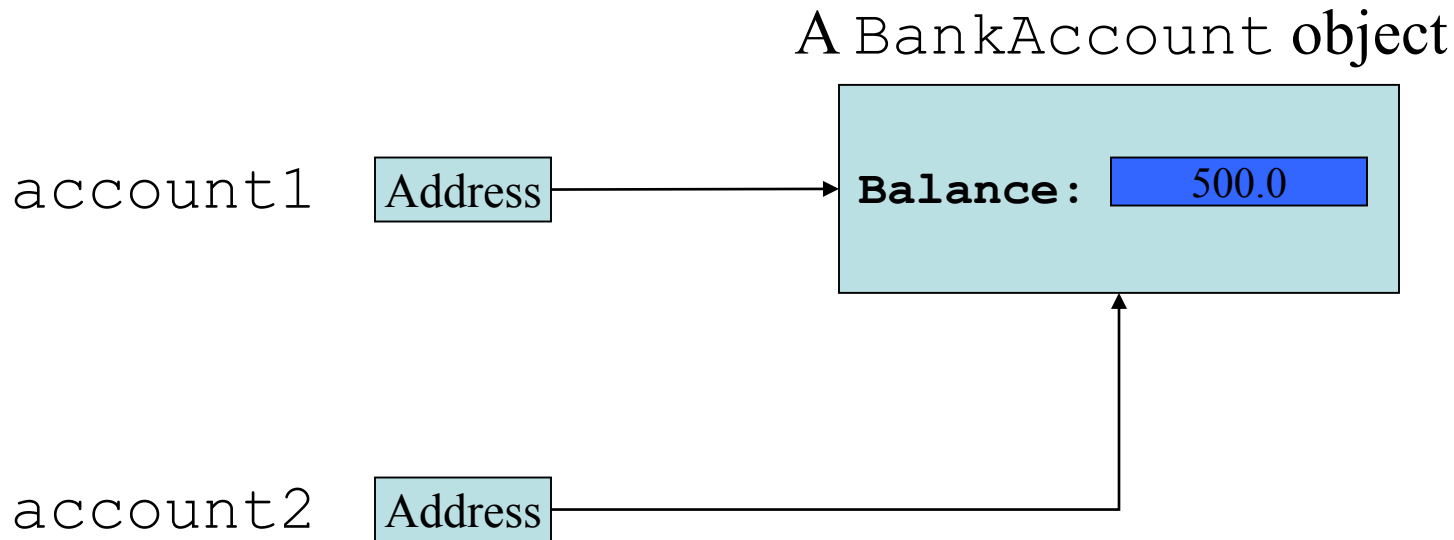
# Garbage Collection

- The Java Virtual Machine has a process that runs in the background that reclaims memory from released objects.
- The *garbage collector* will reclaim memory from any object that no longer has a valid reference pointing to it.

```
BankAccount account1 = new BankAccount(500.0);
BankAccount account2 = account1;
```

- This sets `account1` and `account2` to point to the same object.

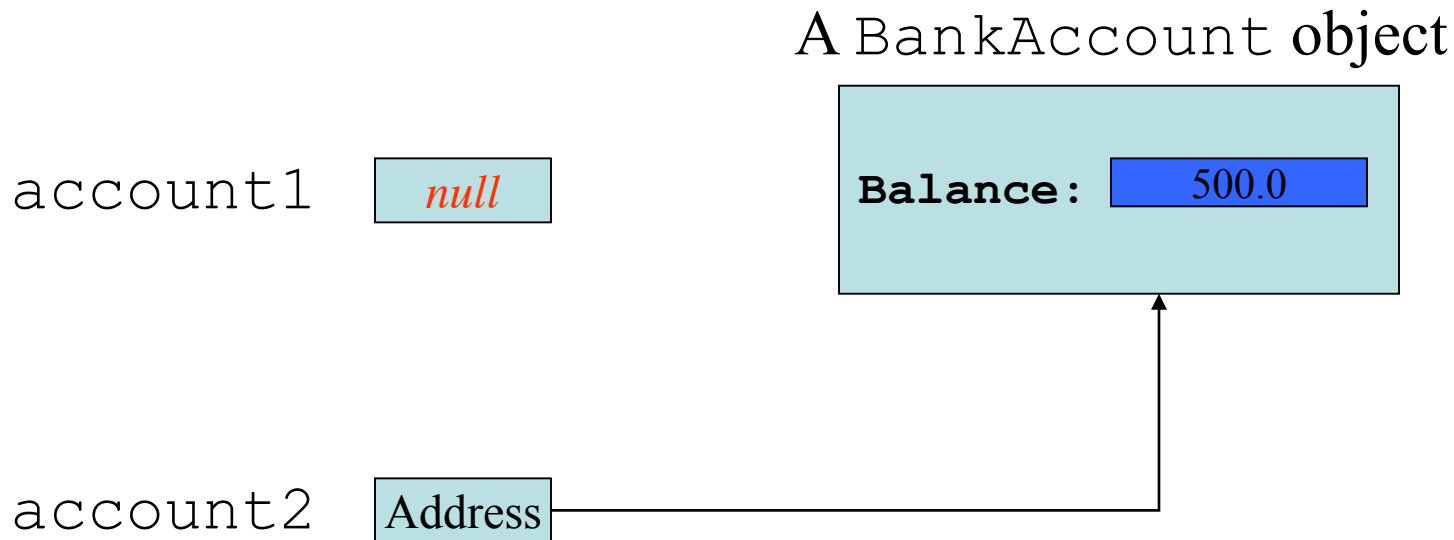
# Garbage Collection



Here, both `account1` and `account2` point to the same instance of the `BankAccount` class.

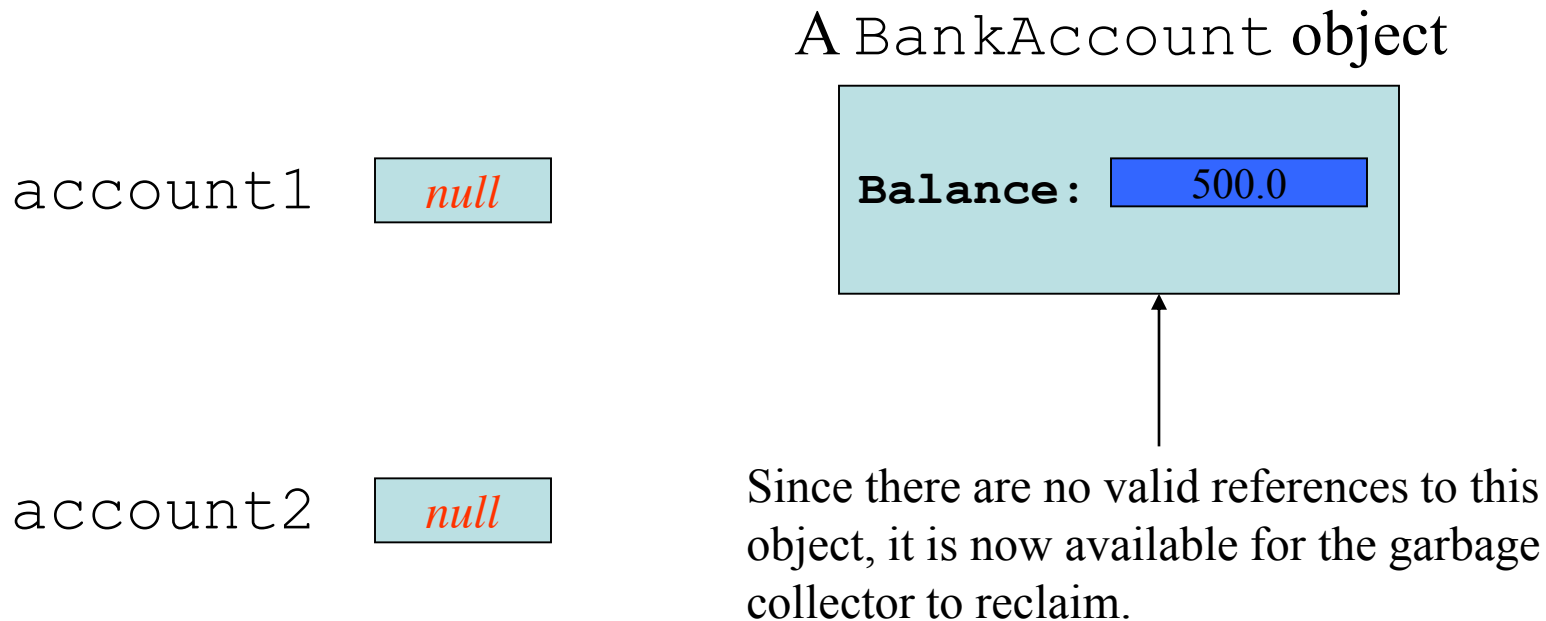


# Garbage Collection



However, by running the statement: **account1 = null;**  
only `account2` will be pointing to the object.

# Garbage Collection



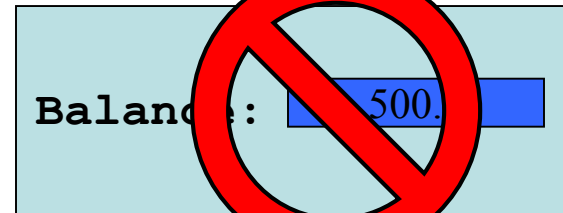
If we now run the statement: **account2 = null;**  
neither account1 or account2 will be pointing to the object.

# Garbage Collection

account1 *null*

account2 *null*

A BankAccount object



The garbage collector reclaims the memory the next time it runs in the background.

# The `finalize` Method

- If a method with the signature:  
**`public void finalize() {...}`**  
is included in a class, it will run just prior to the garbage collector reclaiming its memory.
- The garbage collector is a background thread that runs periodically.
- It cannot be determined when the `finalize` method will actually be run.

# Class Collaboration

- Collaboration – two classes interact with each other
- If an object is to collaborate with another object, it must know something about the second object's methods and how to call them
- If we design a class `StockPurchase` that collaborates with the `Stock` class (previously defined), we define it to create and manipulate a `Stock` object

See examples: [StockPurchase.java](#), [StockTrader.java](#)

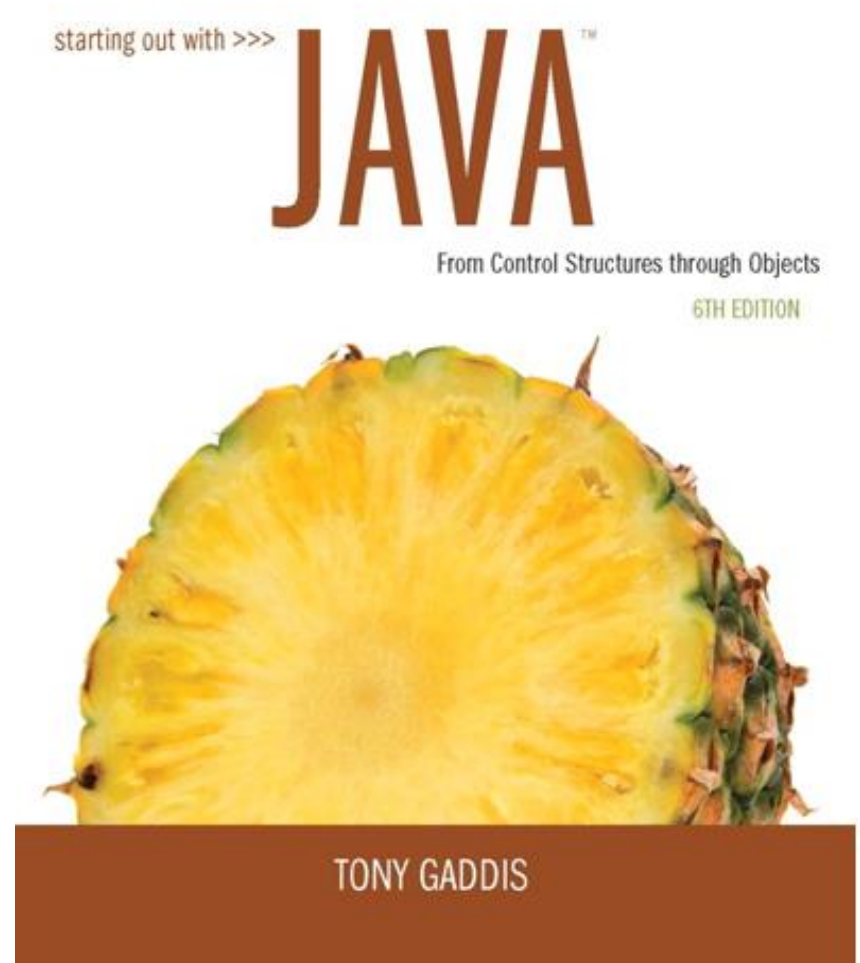
# CRC Cards

- Class, Responsibilities and Collaborations (CRC) cards are useful for determining and documenting a class's responsibilities
  - The things a class is responsible for knowing
  - The actions a class is responsible for doing
- CRC Card Layout (Example for class Stock)

| Stock                      |                    |
|----------------------------|--------------------|
| Know stock to purchase     | Stock class        |
| Know number of shares      | None               |
| Calculate cost of purchase | Stock class        |
| Etc.                       | None or class name |

# CHAPTER 9

## Text Processing and More about Wrapper Classes



# Chapter Topics

Chapter 9 discusses the following main topics:

- Introduction to Wrapper Classes
- Character Testing and Conversion with the Character Class
- More `String` Methods
- The `StringBuilder` Class
- Tokenizing Strings
- Wrapper Classes for the Numeric Data Types
- Focus on Problem Solving: The `TestScoreReader` Class



# Introduction to Wrapper Classes

- Java provides 8 primitive data types.
- They are called “primitive” because they are not created from classes.
- Java provides wrapper classes for all of the primitive data types.
- A *wrapper class* is a class that is “wrapped around” a primitive data type.
- The wrapper classes are part of `java.lang` so to use them, there is no `import` statement required.

# Wrapper Classes

- Wrapper classes allow you to create objects to represent a primitive.
- Wrapper classes are immutable, which means that once you create an object, you cannot change the object's value.
- To get the value stored in an object you must call a method.
- Wrapper classes provide static methods that are very useful

# Character Testing and Conversion With The Character Class

- The `Character` class allows a `char` data type to be *wrapped* in an object.
- The `Character` class provides methods that allow easy testing, processing, and conversion of character data.

# The Character Class

| Method                                                             | Description                                                                                                                          |
|--------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| <code>boolean isDigit(<br/>                char ch)</code>         | Returns true if the argument passed into <i>ch</i> is a digit from 0 through 9. Otherwise returns false.                             |
| <code>boolean isLetter(<br/>                char ch)</code>        | Returns true if the argument passed into <i>ch</i> is an alphabetic letter. Otherwise returns false.                                 |
| <code>boolean isLetterOrDigit(<br/>                char ch)</code> | Returns true if the character passed into <i>ch</i> contains a digit (0 through 9) or an alphabetic letter. Otherwise returns false. |
| <code>boolean isLowerCase(<br/>                char ch)</code>     | Returns true if the argument passed into <i>ch</i> is a lowercase letter. Otherwise returns false.                                   |
| <code>boolean isUpperCase(<br/>                char ch)</code>     | Returns true if the argument passed into <i>ch</i> is an uppercase letter. Otherwise returns false.                                  |
| <code>boolean isSpaceChar(<br/>                char ch)</code>     | Returns true if the argument passed into <i>ch</i> is a space character. Otherwise returns false.                                    |

# Character Testing and Conversion With The `Character` Class

- Example:

[CharacterTest.java](#)

[CustomerNumber.java](#)

- The `Character` class provides two methods that will change the case of a character.

| Method                                                        | Description                                                            |
|---------------------------------------------------------------|------------------------------------------------------------------------|
| <code>char toLowerCase(<br/>                  char ch)</code> | Returns the lowercase equivalent of the argument passed to <i>ch</i> . |
| <code>char toUpperCase(<br/>                  char ch)</code> | Returns the uppercase equivalent of the argument passed to <i>ch</i> . |

See example: [CircleArea.java](#)

# Substrings

- The `String` class provides several methods that search for a string inside of a string.
- A *substring* is a string that is part of another string.
- Some of the substring searching methods provided by the `String` class:

```
boolean startsWith(String str)
```

```
boolean endsWith(String str)
```

```
boolean regionMatches(int start, String str, int start2,
 int n)
```

```
boolean regionMatches(boolean ignoreCase, int start,
 String str, int start2, int n)
```

# Searching Strings

- The `startsWith` method determines whether a string begins with a specified substring.

```
String str = "Four score and seven years ago";
if (str.startsWith("Four"))
 System.out.println("The string starts with Four.");
else
 System.out.println("The string does not start with Four.");
```

- `str.startsWith("Four")` returns true because `str` does begin with “Four”.
- `startsWith` is a case sensitive comparison.

# Searching Strings

- The `endsWith` method determines whether a string ends with a specified substring.

```
String str = "Four score and seven years ago";
if (str.endsWith("ago"))
 System.out.println("The string ends with ago.");
else
 System.out.println("The string does not end with ago.");
```

- The `endsWith` method also performs a case sensitive comparison.
- Example: [PersonSearch.java](#)



# Searching Strings

- The `String` class provides methods that will if specified regions of two strings match.
  - `regionMatches(int start, String str, int start2, int n)`
    - returns true if the specified regions match or false if they don't
    - Case sensitive comparison
  - `regionMatches(boolean ignoreCase, int start, String str, int start2, int n)`
    - If `ignoreCase` is true, it performs case insensitive comparison

# Searching Strings

- The `String` class also provides methods that will locate the position of a substring.
  - `indexOf`
    - returns the first location of a substring or character in the calling `String` Object.
  - `lastIndexOf`
    - returns the last location of a substring or character in the calling `String` Object.

# Searching Strings

```
String str = "Four score and seven years ago";
int first, last;
first = str.indexOf('r');
last = str.lastIndexOf('r');
System.out.println("The letter r first appears at "
 + "position " + first);
System.out.println("The letter r last appears at "
 + "position " + last);
```

---

```
String str = "and a one and a two and a three";
int position;
System.out.println("The word and appears at the "
 + "following locations.");
```

```
position = str.indexOf("and");
while (position != -1)
{
 System.out.println(position);
 position = str.indexOf("and", position + 1);
}
```

# String Methods For Getting Character Or Substring Location

**See Table 9-4**

| Method                                          | Description                                                                                                                                                                                                                                                                                                                   |
|-------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>int indexOf(char ch)</code>               | Searches the calling <code>String</code> object for the character passed into <code>ch</code> . If the character is found, the position of its first occurrence is returned. Otherwise, <code>-1</code> is returned.                                                                                                          |
| <code>int indexOf(char ch, int start)</code>    | Searches the calling <code>String</code> object for the character passed into <code>ch</code> , beginning at the position passed into <code>start</code> and going to the end of the string. If the character is found, the position of its first occurrence is returned. Otherwise, <code>-1</code> is returned.             |
| <code>int indexOf(String str)</code>            | Searches the calling <code>String</code> object for the string passed into <code>str</code> . If the string is found, the beginning position of its first occurrence is returned. Otherwise, <code>-1</code> is returned.                                                                                                     |
| <code>int indexOf(String str, int start)</code> | Searches the calling <code>String</code> object for the string passed into <code>str</code> . The search begins at the position passed into <code>start</code> and goes to the end of the string. If the string is found, the beginning position of its first occurrence is returned. Otherwise, <code>-1</code> is returned. |

# String Methods For Getting Character Or Substring Location

**See Table 9-4**

`int lastIndexOf(char ch)`

Searches the calling `String` object for the character passed into `ch`. If the character is found, the position of its last occurrence is returned. Otherwise, `-1` is returned.

`int lastIndexOf(char ch, int start)`

Searches the calling `String` object for the character passed into `ch`, beginning at the position passed into `start`. The search is conducted backward through the string, to position 0. If the character is found, the position of its last occurrence is returned. Otherwise, `-1` is returned.

`int lastIndexOf(String str)`

Searches the calling `String` object for the string passed into `str`. If the string is found, the beginning position of its last occurrence is returned. Otherwise, `-1` is returned.

`int lastIndexOf(String str,  
int start)`

Searches the calling `String` object for the string passed into `str`, beginning at the position passed into `start`. The search is conducted backward through the string, to position 0. If the string is found, the beginning position of its last occurrence is returned. Otherwise, `-1` is returned.

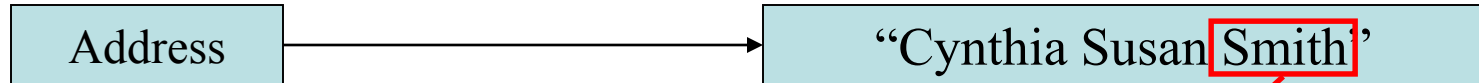
# Extracting Substrings

- The `String` class provides methods to extract substrings in a `String` object.
  - The `substring` method returns a substring beginning at a start location and an optional ending location.

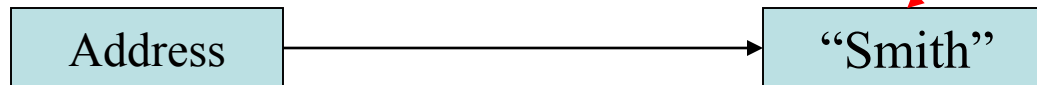
```
String fullName = "Cynthia Susan Smith";
String lastName = fullName.substring(14);
System.out.println("The full name is "
 + fullName);
System.out.println("The last name is "
 + lastName);
```

# Extracting Substrings

The `fullName` variable holds  
the address of a `String` object.



The `lastName` variable holds  
the address of a `String` object.



# Extracting Characters to Arrays

- The `String` class provides methods to extract substrings in a `String` object and store them in `char` arrays.
  - `getChars`
    - Stores a substring in a `char` array
  - `toCharArray`
    - Returns the `String` object's contents in an array of `char` values.
- Example: [StringAnalyzer.java](#)



# Returning Modified Strings

- The `String` class provides methods to return modified `String` objects.
  - `concat`
    - Returns a `String` object that is the concatenation of two `String` objects.
  - `replace`
    - Returns a `String` object with all occurrences of one character being replaced by another character.
  - `trim`
    - Returns a `String` object with all leading and trailing whitespace characters removed.

# The `valueOf` Methods

- The `String` class provides several overloaded `valueOf` methods.
- They return a `String` object representation of
  - a primitive value or
  - a character array.

`String.valueOf(true)` will return `"true"`.

`String.valueOf(5.0)` will return `"5.0"`.

`String.valueOf('C')` will return `"C"`.

# The `valueOf` Methods

```
boolean b = true;
char [] letters = { 'a', 'b', 'c', 'd', 'e' };
double d = 2.4981567;
int i = 7;
System.out.println(String.valueOf(b));
System.out.println(String.valueOf(letters));
System.out.println(String.valueOf(letters, 1, 3));
System.out.println(String.valueOf(d));
System.out.println(String.valueOf(i));
```

- Produces the following output:

```
true
abcde
bcd
2.4981567
7
```

# The `StringBuilder` Class

- The `StringBuilder` class is similar to the `String` class.
- However, you may change the contents of `StringBuilder` objects.
  - You can change specific characters,
  - insert characters,
  - delete characters, and
  - perform other operations.
- A `StringBuilder` object will grow or shrink in size, as needed, to accommodate the changes.

# StringBuilder Constructors

- `StringBuilder()`
  - This constructor gives the object enough storage space to hold 16 characters.
- `StringBuilder(int length)`
  - This constructor gives the object enough storage space to hold *length* characters.
- `StringBuilder(String str)`
  - This constructor initializes the object with the string in *str*.
  - The object will have at least enough storage space to hold the string in *str*.

# Other `StringBuilder` Methods

- The `String` and `StringBuilder` also have common methods:

```
char charAt(int position)
void getChars(int start, int end,
 char[] array, int arrayStart)
int indexOf(String str)
int indexOf(String str, int start)
int lastIndexOf(String str)
int lastIndexOf(String str, int start)
int length()
String substring(int start)
String substring(int start, int end)
```

# Appending to a `StringBuilder` Object

- The `StringBuilder` class has several overloaded versions of a method named `append`.
- They append a string representation of their argument to the calling object's current contents.
- The general form of the `append` method is:  
`object.append(item) ;`
  - where `object` is an instance of the `StringBuilder` class and `item` is:
    - a primitive literal or variable.
    - a `char` array, or
    - a `String` literal or object.

# Appending to a `StringBuilder` Object

- After the `append` method is called, a string representation of *item* will be appended to *object*'s contents.

```
StringBuilder str = new StringBuilder();
```

```
str.append("We sold ");
```

```
str.append(12);
```

```
str.append(" doughnuts for $");
```

```
str.append(15.95);
```

```
System.out.println(str);
```

- This code will produce the following output:  
`We sold 12 doughnuts for $15.95`



# Appending to a `StringBuilder` Object

- The `StringBuilder` class also has several overloaded versions of a method named `insert`
- These methods accept two arguments:
  - an `int` that specifies the position to begin insertion, and
  - the value to be inserted.
- The value to be inserted may be
  - a primitive literal or variable.
  - a `char` array, or
  - a `String` literal or object.

# Appending to a `StringBuilder` Object

- The general form of a typical call to the `insert` method.
  - `object.insert(start, item);`
    - where `object` is an instance of the `StringBuilder` class, `start` is the insertion location, and `item` is:
      - a primitive literal or variable.
      - a `char` array, or
      - a `String` literal or object.
- Example:
  - [Telephone.java](#)
  - [TelephoneTester.java](#)

# Replacing a Substring in a `StringBuilder` Object

- The `StringBuilder` class has a `replace` method that replaces a specified substring with a string.
- The general form of a call to the method:
  - `object.replace(start, end, str);`
    - `start` is an `int` that specifies the starting position of a substring in the calling object, and
    - `end` is an `int` that specifies the ending position of the substring. (The starting position is included in the substring, but the ending position is not.)
    - The `str` parameter is a `String` object.
  - After the method executes, the substring will be replaced with `str`.

## Replacing a Substring in a `StringBuilder` Object

- The `replace` method in this code replaces the word “Chicago” with “New York”.

```
StringBuilder str = new StringBuilder(
 "We moved from Chicago to Atlanta.");
str.replace(14, 21, "New York");
System.out.println(str);
```

- The code will produce the following output:  
`We moved from New York to Atlanta.`

# Other StringBuilder Methods

- The `StringBuilder` class also provides methods to set and delete characters in an object.

```
StringBuilder str = new StringBuilder(
 "I ate 100 blueberries!");
// Display the StringBuilder object.
System.out.println(str);
// Delete the '0'.
str.deleteCharAt(8);
// Delete "blue".
str.delete(9, 13);
// Display the StringBuilder object.
System.out.println(str);
// Change the '1' to '5'
str.setCharAt(6, '5');
// Display the StringBuilder object.
System.out.println(str);
```

# Other StringBuilder Methods

- The `toString` method
  - You can call a `StringBuilder`'s `toString` method to convert that `StringBuilder` object to a regular `String`

```
StringBuilder strb = new StringBuilder("This is a test.");
String str = strb.toString();
```

# Tokenizing Strings

- Use the `String` class's `split` method
- Tokenizes a `String` object and returns an array of `String` objects
- Each array element is one token.

```
// Create a String to tokenize.
String str = "one two three four";
// Get the tokens from the string.
String[] tokens = str.split(" ");
// Display each token.
for (String s : tokens)
 System.out.println(s);
```

- This code will produce the following output:

```
one
two
three
four
```

# Numeric Data Type Wrappers

- Java provides wrapper classes for all of the primitive data types.
- The numeric primitive wrapper classes are:

| Wrapper Class | Numeric Primitive Type It Applies To |
|---------------|--------------------------------------|
| Byte          | byte                                 |
| Double        | double                               |
| Float         | float                                |
| Integer       | int                                  |
| Long          | long                                 |
| Short         | short                                |



# Creating a Wrapper Object

- To create objects from these wrapper classes, you can pass a value to the constructor:

```
Integer number = new Integer(7) ;
```

- You can also assign a primitive value to a wrapper class object:

```
Integer number ;
```

```
number = 7 ;
```

# The Parse Methods

- Recall from Chapter 2, we converted `String` input (from `JOptionPane`) into numbers. Any `String` containing a number, such as “127.89”, can be converted to a numeric data type.
- Each of the numeric wrapper classes has a static method that converts a string to a number.
  - The `Integer` class has a method that converts a `String` to an `int`,
  - The `Double` class has a method that converts a `String` to a `double`,
  - etc.
- These methods are known as *parse methods* because their names begin with the word “parse.”

# The Parse Methods

```
// Store 1 in bVar.
byte bVar = Byte.parseByte("1");
// Store 2599 in iVar.
int iVar = Integer.parseInt("2599");
// Store 10 in sVar.
short sVar = Short.parseShort("10");
// Store 15908 in lVar.
long lVar = Long.parseLong("15908");
// Store 12.3 in fVar.
float fVar = Float.parseFloat("12.3");
// Store 7945.6 in dVar.
double dVar = Double.parseDouble("7945.6");
```

- The parse methods all throw a `NumberFormatException` if the `String` object does not represent a numeric value.

# The toString Methods

- Each of the numeric wrapper classes has a static `toString` method that converts a number to a string.
- The method accepts the number as its argument and returns a string representation of that number.

```
int i = 12;
double d = 14.95;
String str1 = Integer.toString(i) ;
String str2 = Double.toString(d) ;
```

# The `toBinaryString`, `toHexString`, and `toOctalString` Methods

- The `Integer` and `Long` classes have three additional methods:
  - `toBinaryString`, `toHexString`, and `toOctalString`

```
int number = 14;
System.out.println(Integer.toBinaryString(number));
System.out.println(Integer.toHexString(number));
System.out.println(Integer.toOctalString(number));
```

- This code will produce the following output:  
1110  
e  
16

# MIN\_VALUE and MAX\_VALUE

- The numeric wrapper classes each have a set of static final variables
  - MIN\_VALUE and
  - MAX\_VALUE.
- These variables hold the minimum and maximum values for a particular data type.

```
System.out.println("The minimum value for an "
 + "int is "
 + Integer.MIN_VALUE) ;
System.out.println("The maximum value for an "
 + "int is "
 + Integer.MAX_VALUE) ;
```

# Autoboxing and Unboxing

- You can declare a wrapper class variable and assign a value:

```
Integer number;
number = 7;
```

- You may think this is an error, but because `number` is a wrapper class variable, *autoboxing* occurs.
- *Unboxing* does the opposite with wrapper class variables:  

```
Integer myInt = 5; // Autoboxes the value 5
int primitiveNumber;
primitiveNumber = myInt; // unboxing
```

# Autoboxing and Unboxing

- You rarely need to declare numeric wrapper class objects, but they can be useful when you need to work with primitives in a context where primitives are not permitted
- Recall the `ArrayList` class, which works only with objects.

```
ArrayList<int> list =
 new ArrayList<int>(); // Error!
ArrayList<Integer> list =
 new ArrayList<Integer>(); // OK!
```

- Autoboxing and unboxing allow you to conveniently use `ArrayLists` with primitives.



# Problem Solving

- Dr. Harrison keeps student scores in an Excel file. This can be exported as a comma separated text file. Each student's data will be on one line. We want to write a Java program that will find the average for each student. (The number of students changes each year.)
- Solution: [TestScoreReader.java](#), [TestAverages.java](#)

## CHAPTER 10

# Inheritance

starting out with >>>

# JAVA™

From Control Structures through Objects

6TH EDITION



TONY GADDIS

# Chapter Topics

Chapter 10 discusses the following main topics:

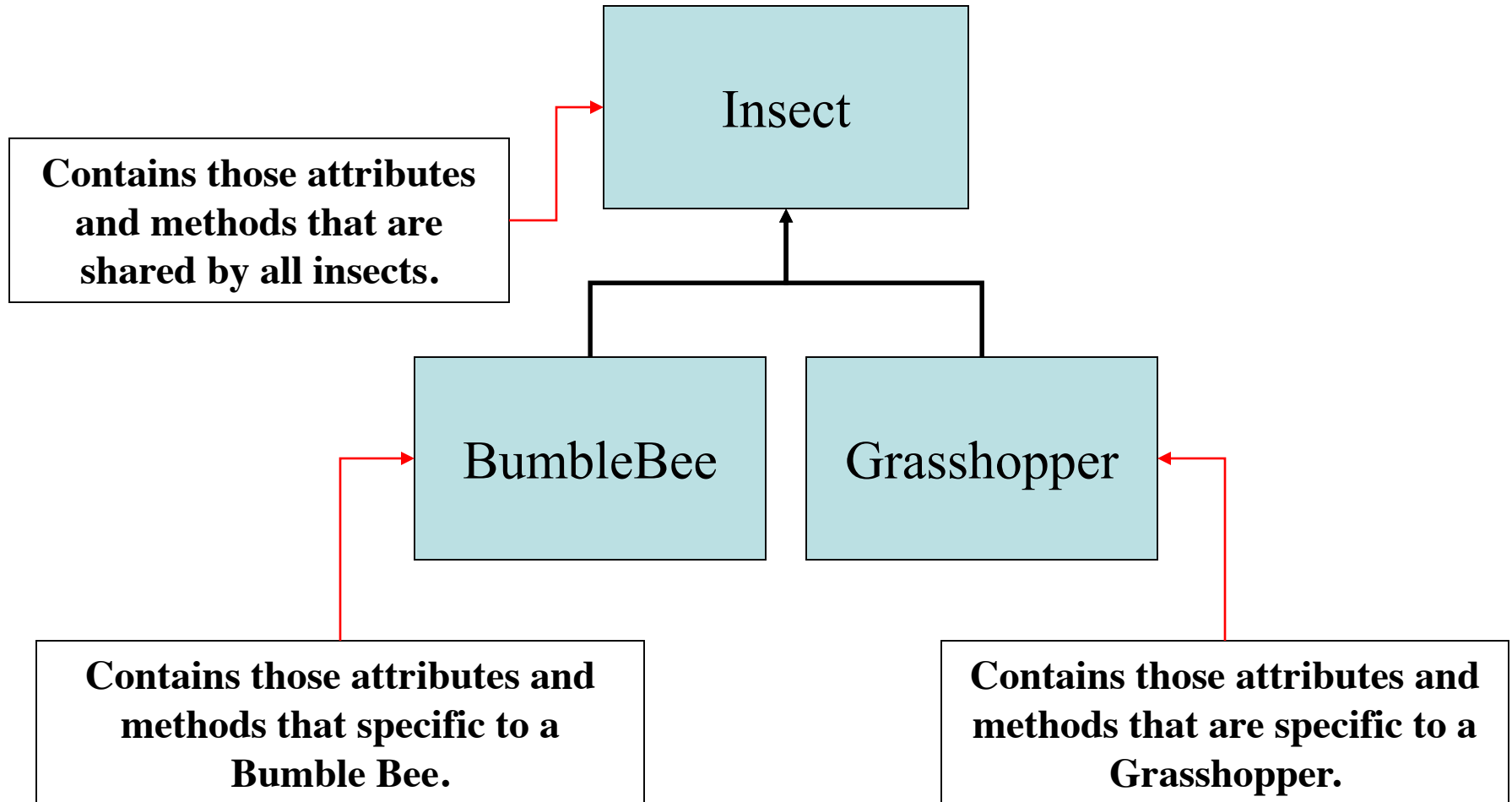
- What Is Inheritance?
- Calling the Superclass Constructor
- Overriding Superclass Methods
- Protected Members
- Chains of Inheritance
- The `Object` Class
- Polymorphism
- Abstract Classes and Abstract Methods
- Interfaces
- Anonymous Classes
- Functional Interfaces and Lambda Expressions

# What is Inheritance?

## Generalization vs. Specialization

- Real-life objects are typically specialized versions of other more general objects.
- The term “insect” describes a very general type of creature with numerous characteristics.
- Grasshoppers and bumblebees are insects
  - They share the general characteristics of an insect.
  - However, they have special characteristics of their own.
    - grasshoppers have a jumping ability, and
    - bumblebees have a stinger.
- Grasshoppers and bumblebees are specialized versions of an insect.

# Inheritance



# The “is a” Relationship

- The relationship between a superclass and an inherited class is called an “is a” relationship.
  - A grasshopper “is a” insect.
  - A poodle “is a” dog.
  - A car “is a” vehicle.
- A specialized object has:
  - all of the characteristics of the general object, plus
  - additional characteristics that make it special.
- In object-oriented programming, *inheritance* is used to create an “is a” relationship among classes.

# The “is a” Relationship

- We can *extend* the capabilities of a class.
- Inheritance involves a superclass and a subclass.
  - The *superclass* is the general class and
  - the *subclass* is the specialized class.
- The subclass is based on, or extended from, the superclass.
  - Superclasses are also called *base classes*, and
  - subclasses are also called *derived classes*.
- The relationship of classes can be thought of as *parent classes* and *child classes*.

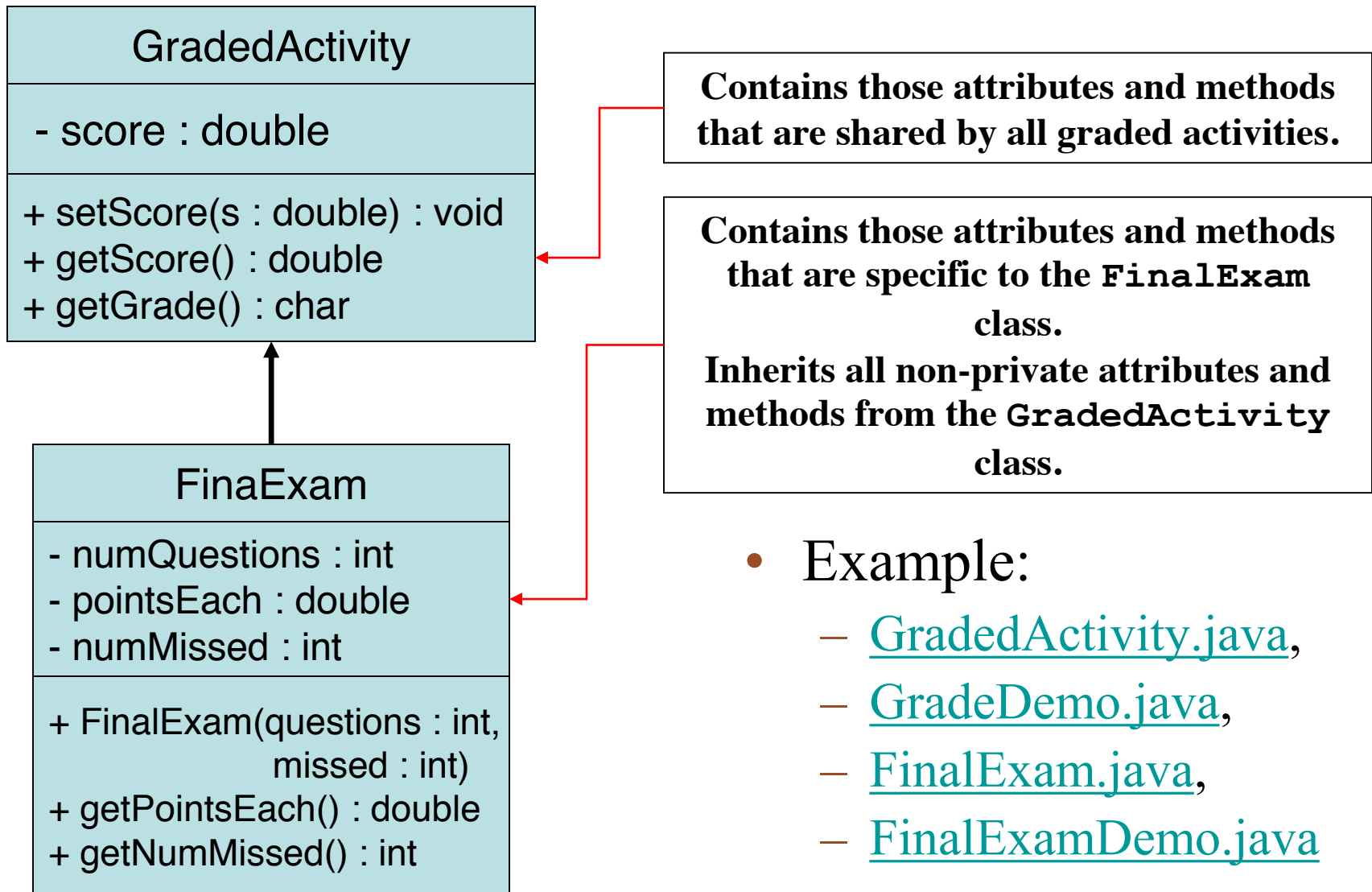
# Inheritance

- The subclass inherits fields and methods from the superclass without any of them being rewritten.
- New fields and methods may be added to the subclass.
- The Java keyword, *extends*, is used on the class header to define the subclass.

```
public class FinalExam extends GradedActivity
```



# The GradedActivity Example



# Inheritance, Fields and Methods

- Members of the superclass that are marked *private*:
  - are not inherited by the subclass,
  - exist in memory when the object of the subclass is created
  - may only be accessed from the subclass by public methods of the superclass.
- Members of the superclass that are marked *public*:
  - are inherited by the subclass, and
  - may be directly accessed from the subclass.

# Inheritance, Fields and Methods

- When an instance of the subclass is created, the non-private methods of the superclass are available through the subclass object.

```
FinalExam exam = new FinalExam();
exam.setScore(85.0);
System.out.println("Score = "
 + exam.getScore());
```

- Non-private methods and fields of the superclass are available in the subclass.

```
setScore(newScore);
```

# Inheritance and Constructors

- Constructors are not inherited.
- When a subclass is instantiated, the superclass default constructor is executed first.
- Example:
  - [SuperClass1.java](#)
  - [SubClass1.java](#)
  - [ConstructorDemo1.java](#)

# The Superclass's Constructor

- The `super` keyword refers to an object's superclass.
- The superclass constructor can be explicitly called from the subclass by using the `super` keyword.
- Example:
  - [SuperClass2.java](#), [SubClass2.java](#), [ConstructorDemo2.java](#)
  - [Rectangle.java](#), [Cube.java](#), [CubeDemo.java](#)

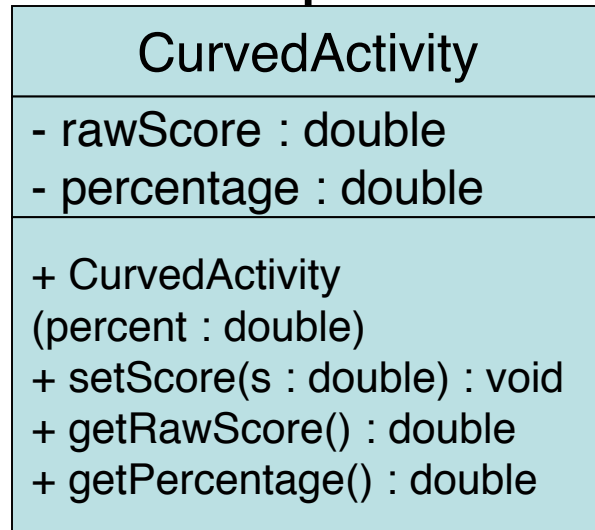
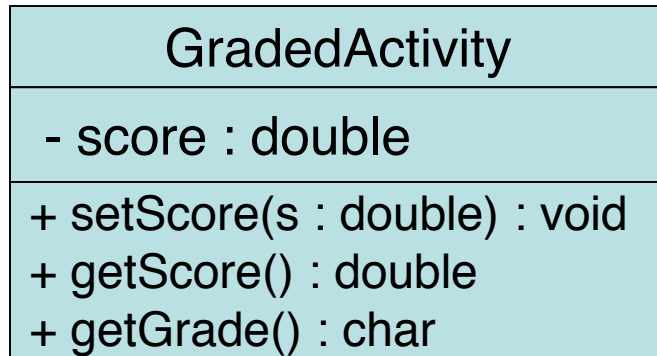
# Calling The Superclass Constructor

- If a parameterized constructor is defined in the superclass,
  - the superclass must provide a no-arg constructor, or
    - subclasses must provide a constructor, and
    - subclasses must call a superclass constructor.
- Calls to a superclass constructor must be the first java statement in the subclass constructors.

# Overriding Superclass Methods

- A subclass may have a method with the same signature as a superclass method.
- The subclass method overrides the superclass method.
- This is known as *method overriding*.
- Example:
  - [GradedActivity.java](#), [CurvedActivity.java](#),  
[CurvedActivityDemo.java](#)

# Overriding Superclass Methods



**This method is a more specialized version of the `setScore` method in the superclass, `GradedActivity`.**





# Overriding Superclass Methods

- Recall that a method's *signature* consists of:
  - the method's name
  - the data types method's parameters in the order that they appear.
- A subclass method that overrides a superclass method must have the same signature as the superclass method.
- An object of the subclass invokes the subclass's version of the method, not the superclass's.
- The `@Override` annotation should be used just before the subclass method declaration.
  - This causes the compiler to display a error message if the method fails to correctly override a method in the superclass.

# Overriding Superclass Methods

- An subclass method can call the overridden superclass method via the `super` keyword.

```
super.setScore(rawScore * percentage) ;
```

- There is a distinction between overloading a method and overriding a method.
- Overloading is when a method has the same name as one or more other methods, but with a different signature.
- When a method overrides another method, however, they both have the same signature.

# Overriding Superclass Methods

- Both overloading and overriding can take place in an inheritance relationship.
- Overriding can only take place in an inheritance relationship.
- Example:
  - [SuperClass3.java](#),
  - [SubClass3.java](#),
  - [ShowValueDemo.java](#)

# Preventing a Method from Being Overridden

- The `final` modifier will prevent the overriding of a superclass method in a subclass.

```
public final void message()
```

- If a subclass attempts to override a final method, the compiler generates an error.
- This ensures that a particular superclass method is used by subclasses rather than a modified version of it.

# Protected Members

- Protected members of class:
  - may be accessed by methods in a subclass, and
  - by methods in the same package as the class.
- Java provides a third access specification, `protected`.
- A *protected* member's access is somewhere between *private* and *public*.
- Example:
  - [GradedActivity2.java](#)
  - [FinalExam2.java](#)
  - [ProtectedDemo.java](#)

# Protected Members

- Using `protected` instead of `private` makes some tasks easier.
- However, any class that is derived from the class, or is in the same package, has unrestricted access to the protected member.
- It is always better to make all fields `private` and then provide `public` methods for accessing those fields.
- If no access specifier for a class member is provided, the class member is given *package access* by default.
- Any method in the same package may access the member.

# Access Specifiers

| <b>Access Modifier</b>   | <b>Accessible to a subclass inside the same package?</b> | <b>Accessible to all other classes inside the same package?</b> |
|--------------------------|----------------------------------------------------------|-----------------------------------------------------------------|
| default<br>(no modifier) | Yes                                                      | Yes                                                             |
| Public                   | Yes                                                      | Yes                                                             |
| Protected                | Yes                                                      | Yes                                                             |
| Private                  | No                                                       | No                                                              |

| <b>Access Modifier</b>   | <b>Accessible to a subclass outside the package?</b> | <b>Accessible to all other classes outside the package?</b> |
|--------------------------|------------------------------------------------------|-------------------------------------------------------------|
| default<br>(no modifier) | No                                                   | No                                                          |
| Public                   | Yes                                                  | Yes                                                         |
| Protected                | Yes                                                  | No                                                          |
| Private                  | No                                                   | No                                                          |

# Chains of Inheritance

- A superclass can also be derived from another class.

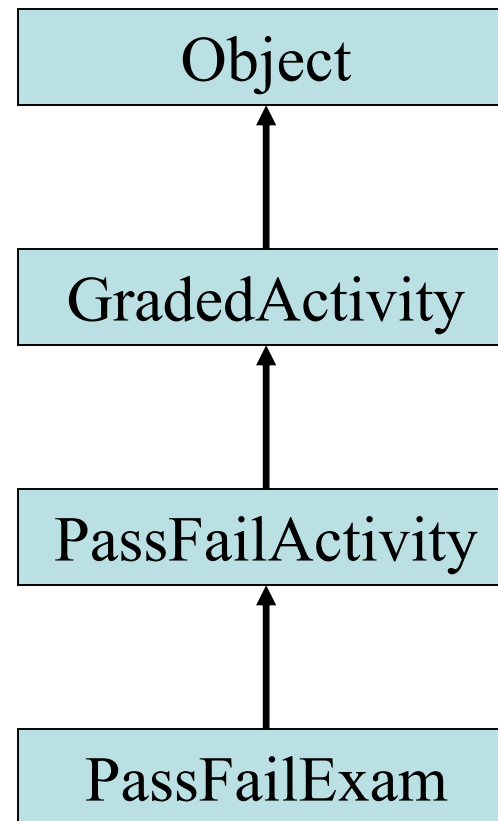
Example:

[GradedActivity.java](#)

[PassFailActivity.java](#)

[PassFailExam.java](#)

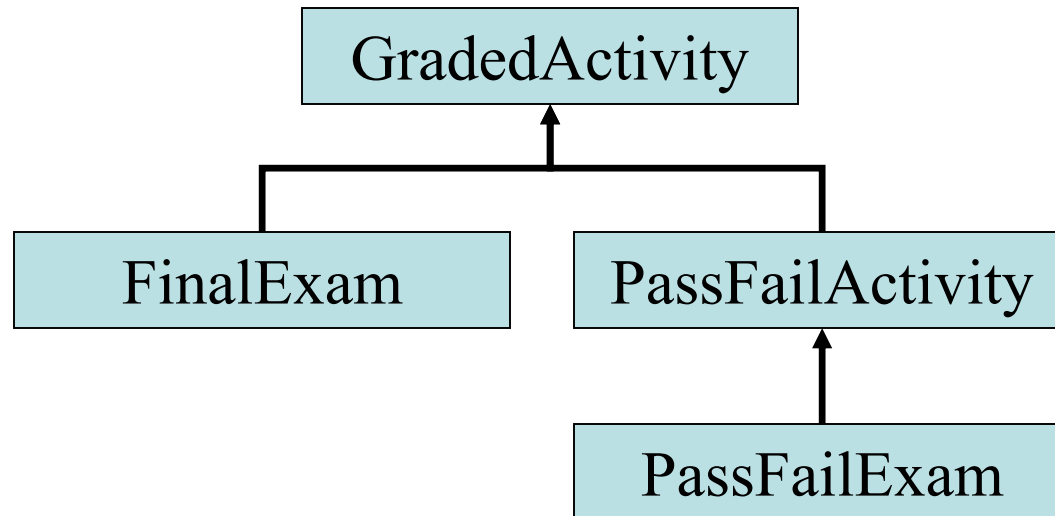
[PassFailExamDemo.java](#)





# Chains of Inheritance

- Classes often are depicted graphically in a *class hierarchy*.
- A class hierarchy shows the inheritance relationships between classes.



# The Object Class

- All Java classes are directly or indirectly derived from a class named `Object`.
- `Object` is in the `java.lang` package.
- Any class that does not specify the `extends` keyword is automatically derived from the `Object` class.

```
public class MyClass
{
 // This class is derived from Object.
}
```

- Ultimately, every class is derived from the `Object` class.

# The Object Class

- Because every class is directly or indirectly derived from the `Object` class:
  - every class inherits the `Object` class's members.
    - example: `toString` and `equals`.
- In the `Object` class, the `toString` method returns a string containing the object's class name and a hash of its memory address.
- The `equals` method accepts the address of an object as its argument and returns `true` if it is the same as the calling object's address.
- Example: [ObjectMethods.java](#)

# Polymorphism

- A reference variable can reference objects of classes that are derived from the variable's class.

```
GradedActivity exam;
```

- We can use the exam variable to reference a `GradedActivity` object.

```
exam = new GradedActivity();
```

- The `GradedActivity` class is also used as the superclass for the `FinalExam` class.
- An object of the `FinalExam` class *is a* `GradedActivity` object.

# Polymorphism

- A `GradedActivity` variable can be used to reference a `FinalExam` object.

```
GradedActivity exam = new FinalExam(50, 7);
```

- This statement creates a `FinalExam` object and stores the object's address in the `exam` variable.
- This is an example of polymorphism.
- The term *polymorphism* means the ability to take many forms.
- In Java, a reference variable is *polymorphic* because it can reference objects of types different from its own, as long as those types are subclasses of its type.

# Polymorphism

- Other legal polymorphic references:

```
GradedActivity exam1 = new FinalExam(50, 7);
GradedActivity exam2 = new PassFailActivity(70);
GradedActivity exam3 = new PassFailExam(100, 10, 70);
```

- The `GradedActivity` class has three methods: `setScore`, `getScore`, and `getGrade`.
- A `GradedActivity` variable can be used to call only those three methods.

```
GradedActivity exam = new PassFailExam(100, 10, 70);
System.out.println(exam.getScore()); // This works.
System.out.println(exam.getGrade()); // This works.
System.out.println(exam.getPointsEach()); // ERROR!
```

# Polymorphism and Dynamic Binding

- If the object of the subclass has overridden a method in the superclass:
    - If the variable makes a call to that method the subclass's version of the method will be run.
- ```
GradedActivity exam = new PassFailActivity(60);  
exam.setScore(70);  
System.out.println(exam.getGrade());
```
- Java performs *dynamic binding* or *late binding* when a variable contains a polymorphic reference.
 - The Java Virtual Machine determines at runtime which method to call, depending on the type of object that the variable references.

Polymorphism

- It is the object's type, rather than the reference type, that determines which method is called.
- Example:
 - [Polymorphic.java](#)
- You cannot assign a superclass object to a subclass reference variable.

Abstract Classes

- An abstract class cannot be instantiated, but other classes are derived from it.
- An *Abstract class* serves as a superclass for other classes.
- The abstract class represents the generic or abstract form of all the classes that are derived from it.
- A class becomes abstract when you place the abstract key word in the class definition.

```
public abstract class ClassName
```

Abstract Methods

- An abstract method has no body and must be overridden in a subclass.
- An *abstract method* is a method that appears in a superclass, but expects to be overridden in a subclass.
- An abstract method has only a header and no body.

`AccessSpecifier abstract ReturnType MethodName(ParameterList) ;`

- Example:
 - [Student.java](#), [CompSciStudent.java](#), [CompSciStudentDemo.java](#)

Abstract Methods

- Notice that the key word `abstract` appears in the header, and that the header ends with a semicolon.

```
public abstract void setValue(int value);
```

- Any class that contains an abstract method is automatically abstract.
- If a subclass fails to override an abstract method, a compiler error will result.
- Abstract methods are used to ensure that a subclass implements the method.

Interfaces

- An *interface* is similar to an abstract class that has all abstract methods.
 - It cannot be instantiated, and
 - all of the methods listed in an interface must be written elsewhere.
- The purpose of an interface is to specify behavior for other classes.
- It is often said that an interface is like a “contract,” and when a class implements an interface it must adhere to the contract.
- An interface looks similar to a class, except:
 - the keyword `interface` is used instead of the keyword `class`, and
 - the methods that are specified in an interface have no bodies, only headers that are terminated by semicolons.

Interfaces

- The general format of an interface definition:

```
public interface InterfaceName
{
    (Method headers...)
}
```

- All methods specified by an interface are public by default.
- A class can implement one or more interfaces.

Interfaces

- If a class implements an interface, it uses the `implements` keyword in the class header.

```
public class FinalExam3 extends GradedActivity  
    implements Relatable
```

- Example:
 - [GradedActivity.java](#)
 - [Relatable.java](#)
 - [FinalExam3.java](#)
 - [InterfaceDemo.java](#)

Fields in Interfaces

- An interface can contain field declarations:
 - all fields in an interface are treated as `final` and `static`.
- Because they automatically become `final`, you must provide an initialization value.

```
public interface Doable
{
    int FIELD1 = 1, FIELD2 = 2;
    (Method headers...)
}
```

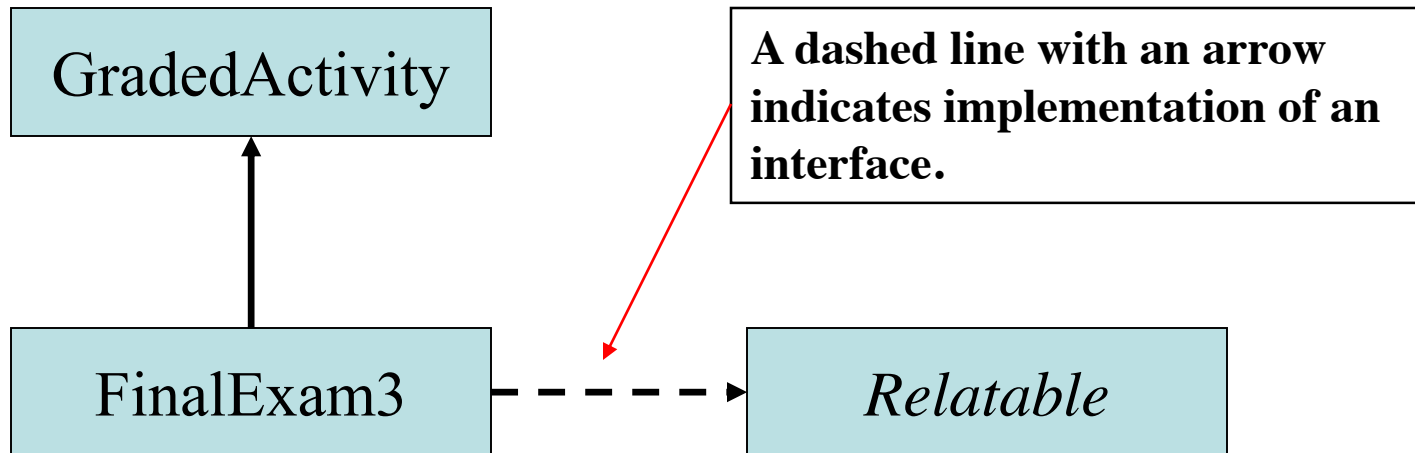
- In this interface, `FIELD1` and `FIELD2` are `final static int` variables.
- Any class that implements this interface has access to these variables.

Implementing Multiple Interfaces

- A class can be derived from only one superclass.
- Java allows a class to implement multiple interfaces.
- When a class implements multiple interfaces, it must provide the methods specified by all of them.
- To specify multiple interfaces in a class definition, simply list the names of the interfaces, separated by commas, after the implements key word.

```
public class MyClass implements Interface1,  
                                Interface2,  
                                Interface3
```


Interfaces in UML



Polymorphism with Interfaces

- Java allows you to create reference variables of an interface type.
- An interface reference variable can reference any object that implements that interface, regardless of its class type.
- This is another example of polymorphism.
- Example:
 - [RetailItem.java](#)
 - [CompactDisc.java](#)
 - [DvdMovie.java](#)
 - [PolymorphicInterfaceDemo.java](#)

Polymorphism with Interfaces

- In the example code, two `RetailItem` reference variables, `item1` and `item2`, are declared.
- The `item1` variable references a `CompactDisc` object and the `item2` variable references a `DvdMovie` object.
- When a class implements an interface, an inheritance relationship known as *interface inheritance* is established.
 - a `CompactDisc` object *is a* `RetailItem`, and
 - a `DvdMovie` object *is a* `RetailItem`.

Polymorphism with Interfaces

- A reference to an interface can point to any class that implements that interface.
- You cannot create an instance of an interface.

```
RetailItem item = new RetailItem(); // ERROR!
```

- When an interface variable references an object:
 - only the methods declared in the interface are available,
 - explicit type casting is required to access the other methods of an object referenced by an interface reference.

Default Methods

- Beginning in Java 8, interfaces may have *default methods*.
- A default method is an interface method that has a body.
- You can add new methods to an existing interface without causing errors in the classes that already implement the interface.
- Example:
 - [Displayable.java](#)
 - [Person.java](#)
 - [InterfaceDemoDefaultMethod.java](#)

Anonymous Inner Classes

- An inner class is a class that is defined inside another class.
- An anonymous inner class is an inner class that has no name.
- An anonymous inner class must implement an interface, or extend another class.
- Useful when you need a class that is simple, and to be instantiated only once in your code.
- Example:
 - [IntCalculator.java](#)
 - [AnonymousClassDemo.java](#)

Functional Interfaces and Lambda Expressions

- A functional interface is an interface that has one abstract method.
- A lambda expression can be used to create an object that implements the interface, and overrides its abstract method.
- In Java 8, these features work together to simplify code, particularly in situations where you might use anonymous inner classes.
- Example:
 - [LambdaDemo.java](#)
 - [LambdaDemo2.java](#)

CHAPTER 11

Exceptions and Advanced File I/O

starting out with >>>

JAVA™

From Control Structures through Objects

6TH EDITION



TONY GADDIS

Chapter Topics

Chapter 11 discusses the following main topics:

- Handling Exceptions
- Throwing Exceptions
- More about Input/Output Streams
- Advanced Topics:
 - Binary Files,
 - Random Access Files, and
 - Object Serialization

Handling Exceptions

- An exception is an object that is generated as the result of an error or an unexpected event.
- Exception are said to have been “thrown.”
- It is the programmers responsibility to write code that detects and handles exceptions.
- Unhandled exceptions will crash a program.
- Example: [BadArray.java](#)
- Java allows you to create exception handlers.

Handling Exceptions

- An *exception handler* is a section of code that gracefully responds to exceptions.
- The process of intercepting and responding to exceptions is called *exception handling*.
- The *default exception handler* deals with unhandled exceptions.
- The default exception handler prints an error message and crashes the program.

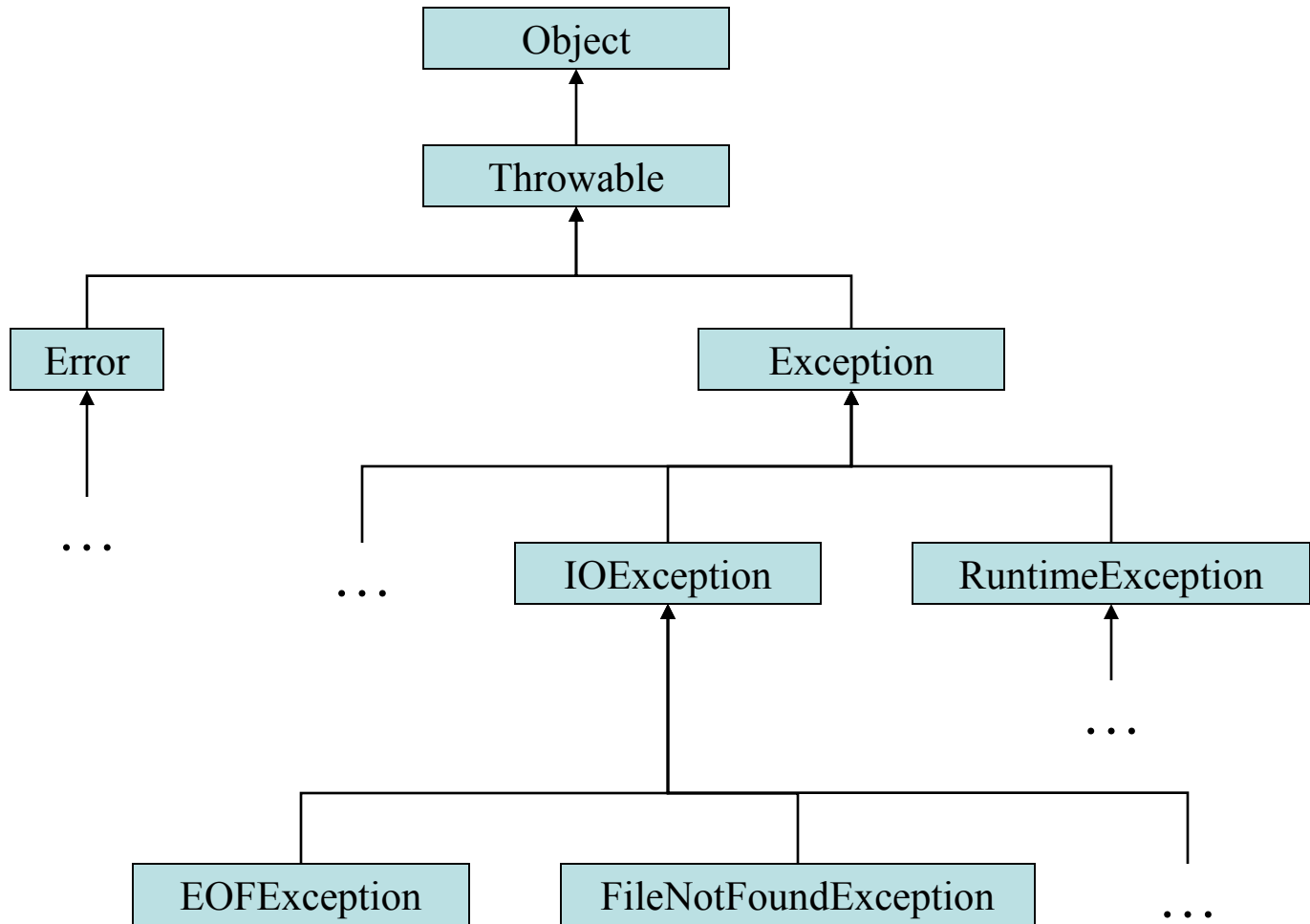
Exception Classes

- An exception is an object.
- Exception objects are created from classes in the Java API hierarchy of exception classes.
- All of the exception classes in the hierarchy are derived from the `Throwable` class.
- `Error` and `Exception` are derived from the `Throwable` class.

Exception Classes

- Classes that are derived from `Error`:
 - are for exceptions that are thrown when critical errors occur. (i.e.)
 - an internal error in the Java Virtual Machine, or
 - running out of memory.
- Applications should not try to handle these errors because they are the result of a serious condition.
- Programmers should handle the exceptions that are instances of classes that are derived from the `Exception` class.

Exception Classes



Handling Exceptions

- To handle an exception, you use a *try* statement.

```
try
{
    (try block statements...)
}
catch (ExceptionType ParameterName)
{
    (catch block statements...)
}
```

- First the keyword `try` indicates a block of code will be attempted (the curly braces are required).
- This block of code is known as a *try block*.

Handling Exceptions

- A *try block* is:
 - one or more statements that are executed, and
 - can potentially throw an exception.
- The application will not halt if the try block throws an exception.
- After the try block, a `catch` clause appears.

Handling Exceptions

- A catch clause begins with the key word `catch`:

`catch (ExceptionType ParameterName)`

- *ExceptionType* is the name of an exception class and
 - *ParameterName* is a variable name which will reference the exception object if the code in the try block throws an exception.
- The code that immediately follows the catch clause is known as a *catch block* (the curly braces are required).
- The code in the catch block is executed if the try block throws an exception.

Handling Exceptions

- This code is designed to handle a `FileNotFoundException` if it is thrown.

```
try
{
    File file = new File ("MyFile.txt");
    Scanner inputFile = new Scanner(file);
}
catch (FileNotFoundException e)
{
    System.out.println("File not found.");
}
```

- The Java Virtual Machine searches for a `catch` clause that can deal with the exception.
- Example: [OpenFile.java](#)

Handling Exceptions

- The parameter must be of a type that is compatible with the thrown exception's type.
- After an exception, the program will continue execution at the point just past the catch block.

Handling Exceptions

- Each exception object has a method named `getMessage` that can be used to retrieve the default error message for the exception.
- Example:
 - [ExceptionMessage.java](#)
 - [ParseIntError.java](#)

Polymorphic References To Exceptions

- When handling exceptions, you can use a polymorphic reference as a parameter in the `catch` clause.
- Most exceptions are derived from the `Exception` class.
- A `catch` clause that uses a parameter variable of the `Exception` type is capable of catching any exception that is derived from the `Exception` class.

Polymorphic References To Exceptions

```
try
{
    number = Integer.parseInt(str);
}
catch (Exception e)
{
    System.out.println("The following error occurred: "
                       + e.getMessage());
}
```

- The Integer class's parseInt method throws a NumberFormatException object.
- The NumberFormatException class is derived from the Exception class.

Handling Multiple Exceptions

- The code in the try block may be capable of throwing more than one type of exception.
- A `catch` clause needs to be written for each type of exception that could potentially be thrown.
- The JVM will run the first compatible `catch` clause found.
- The `catch` clauses must be listed from most specific to most general.
- Example: [SalesReport.java](#), [SalesReport2.java](#)

Exception Handlers

- There can be many polymorphic catch clauses.
- A try statement may have only one catch clause for each specific type of exception.

```
try
{
    number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
    System.out.println("Bad number format.");
}
catch (NumberFormatException e) // ERROR!!!
{
    System.out.println(str + " is not a number.");
}
```


Exception Handlers

- The `NumberFormatException` class is derived from the `IllegalArgumentException` class.

```
try
{
    number = Integer.parseInt(str);
}
catch (IllegalArgumentException e)
{
    System.out.println("Bad number format.");
}
catch (NumberFormatException e) // ERROR!!!
{
    System.out.println(str + " is not a number.");
}
```

Exception Handlers

- The previous code could be rewritten to work, as follows, with no errors:

```
try
{
    number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
    System.out.println(str +
                        " is not a number.");
}
catch (IllegalArgumentException e) //OK
{
    System.out.println("Bad number format.");
}
```

The `finally` Clause

- The try statement may have an optional `finally` clause.
- If present, the `finally` clause must appear after all of the catch clauses.

```
try
{
    (try block statements...)
}
catch (ExceptionType ParameterName)
{
    (catch block statements...)
}
finally
{
    (finally block statements...)
}
```

The *finally* Clause

- The *finally block* is one or more statements,
 - that are always executed after the try block has executed and
 - after any catch blocks have executed if an exception was thrown.
- The statements in the finally block execute whether an exception occurs or not.

The Stack Trace

- The *call stack* is an internal list of all the methods that are currently executing.
- A *stack trace* is a list of all the methods in the call stack.
- It indicates:
 - the method that was executing when an exception occurred and
 - all of the methods that were called in order to execute that method.
- Example: [StackTrace.java](#)

Multi-Catch (Java 7)

- Beginning in Java 7, you can specify more than one exception in a catch clause:

```
try
{
}
catch (NumberFormatException | InputMismatchException ex)
{
}
```



Separate the exceptions with
the | character.

Uncaught Exceptions

- When an exception is thrown, it cannot be ignored.
- It must be handled by the program, or by the default exception handler.
- When the code in a method throws an exception:
 - normal execution of that method stops, and
 - the JVM searches for a compatible exception handler inside the method.

Uncaught Exceptions

- If there is no exception handler inside the method:
 - control of the program is passed to the previous method in the call stack.
 - If that method has no exception handler, then control is passed again, up the call stack, to the previous method.
- If control reaches the `main` method:
 - the main method must either handle the exception, or
 - the program is halted and the default exception handler handles the exception.

Checked and Unchecked Exceptions

- There are two categories of exceptions:
 - unchecked
 - checked.
- *Unchecked exceptions* are those that are derived from the `Error` class or the `RuntimeException` class.
- Exceptions derived from `Error` are thrown when a critical error occurs, and should not be handled.
- `RuntimeException` serves as a superclass for exceptions that result from programming errors.

Checked and Unchecked Exceptions

- These exceptions can be avoided with properly written code.
- Unchecked exceptions, in most cases, should not be handled.
- All exceptions that are *not* derived from `Error` or `RuntimeException` are *checked exceptions*.

Checked and Unchecked Exceptions

- If the code in a method can throw a checked exception, the method:
 - must handle the exception, or
 - it must have a `throws` clause listed in the method header.
- The `throws` clause informs the compiler what exceptions can be thrown from a method.

Checked and Unchecked Exceptions

```
// This method will not compile!
public void displayFile(String name)
{
    // Open the file.
    File file = new File(name);
    Scanner inputFile = new Scanner(file);
    // Read and display the file's contents.
    while (inputFile.hasNext())
    {
        System.out.println(inputFile.nextLine());
    }
    // Close the file.
    inputFile.close();
}
```

Checked and Unchecked Exceptions

- The code in this method is capable of throwing checked exceptions.
- The keyword `throws` can be written at the end of the method header, followed by a list of the types of exceptions that the method can throw.

```
public void displayFile(String name)  
    throws FileNotFoundException
```

Throwing Exceptions

- You can write code that:
 - throws one of the standard Java exceptions, or
 - an instance of a custom exception class that you have designed.
- The `throw` statement is used to manually throw an exception.

```
throw new ExceptionType (MessageString) ;
```

- The `throw` statement causes an exception object to be created and thrown.

Throwing Exceptions

- The *MessageString* argument contains a custom error message that can be retrieved from the exception object's `getMessage` method.
- If you do not pass a message to the constructor, the exception will have a null message.

```
throw new Exception("Out of fuel");
```

— *Note: Don't confuse the `throw` statement with the `throws` clause.*

- Example: [DieExceptionDemo.java](#)

Creating Exception Classes

- You can create your own exception classes by deriving them from the `Exception` class or one of its derived classes.
- Example:
 - [BankAccount.java](#)
 - [NegativeStartingBalance.java](#)
 - [AccountTest.java](#)

Creating Exception Classes

- Some examples of exceptions that can affect a bank account:
 - A negative starting balance is passed to the constructor.
 - A negative interest rate is passed to the constructor.
 - A negative number is passed to the deposit method.
 - A negative number is passed to the withdraw method.
 - The amount passed to the withdraw method exceeds the account's balance.
- We can create exceptions that represent each of these error conditions.

@exception Tag in Documentation Comments

- General format

`@exception` *ExceptionName Description*

- The following rules apply

- The `@exception` tag in a method's documentation comment must appear after the general description of the method.
- The description can span several lines. It ends at the end of the documentation comment (the `*/` symbol) or at the beginning of another tag.

Binary Files

- The way data is stored in memory is sometimes called the *raw binary format*.
- Data can be stored in a file in its raw binary format.
- A file that contains binary data is often called a *binary file*.
- Storing data in its binary format is more efficient than storing it as text.
- There are some types of data that should only be stored in its raw binary format.

Binary Files

- Binary files cannot be opened in a text editor such as Notepad.
- To write data to a binary file you must create objects from the following classes:
 - **FileOutputStream** - allows you to open a file for writing binary data. It provides only basic functionality for writing bytes to the file.
 - **DataOutputStream** - allows you to write data of any primitive type or String objects to a binary file. Cannot directly access a file. It is used in conjunction with a `FileOutputStream` object that has a connection to a file.

Binary Files

- A `DataOutputStream` object is wrapped around a `FileOutputStream` object to write data to a binary file.

```
FileOutputStream fstream = new  
    FileOutputStream("MyInfo.dat");  
DataOutputStream outputFile = new  
    DataOutputStream(fstream);
```

- If the file that you are opening with the `FileOutputStream` object already exists, it will be erased and an empty file by the same name will be created.

Binary Files

- These statements can be combined into one.

```
DataOutputStream outputFile = new  
    OutputStream(new  
        FileOutputStream("MyInfo.dat"));
```

- Once the `DataOutputStream` object has been created, you can use it to write binary data to the file.
- Example: [WriteBinaryFile.java](#)

Binary Files

- To open a binary file for input, you wrap a `DataInputStream` object around a `FileInputStream` object.

```
FileInputStream fstream = new
    FileInputStream("MyInfo.dat");
DataInputStream inputFile = new
    DataInputStream(fstream);
```

- These two statements can be combined into one.

```
DataInputStream inputFile = new
    DataInputStream(new
        FileInputStream("MyInfo.dat"));
```

Binary Files

- The `FileInputStream` constructor will throw a `FileNotFoundException` if the file named by the string argument cannot be found.
- Once the `DataInputStream` object has been created, you can use it to read binary data from the file.
- Example:
 - [ReadBinaryFile.java](#)

Writing and Reading Strings

- To write a string to a binary file, use the `DataOutputStream` class's `writeUTF` method.
- This method writes its `String` argument in a format known as *UTF-8 encoding*.
 - Just before writing the string, this method writes a two-byte integer indicating the number of bytes that the string occupies.
 - Then, it writes the string's characters in Unicode. (UTF stands for Unicode Text Format.)
- The `DataInputStream` class's `readUTF` method reads from the file.

Writing and Reading Strings

- To write a string to a file:
`String name = "Chloe";`
`outputFile.writeUTF(name);`
- To read a string from a file:
`String name = inputFile.readUTF();`
- The `readUTF` method will correctly read a string only when the string was written with the `writeUTF` method.
- Example:
 - [WriteUTF.java](#)
 - [ReadUTF.java](#)

Appending Data to Binary Files

- The `FileOutputStream` constructor takes an optional second argument which must be a `boolean` value.
- If the argument is `true`, the file will not be erased if it exists; new data will be written to the end of the file.
- If the argument is `false`, the file will be erased if it already exists.

```
FileOutputStream fstream = new  
    FileOutputStream("MyInfo.dat", true);  
DataOutputStream outputFile = new  
    DataOutputStream(fstream);
```

Random Access Files

- Text files and the binary files previously shown use *sequential file access*.
- With sequential access:
 - The first time data is read from the file, the data will be read from its beginning.
 - As the reading continues, the file's read position advances sequentially through the file's contents.
- Sequential file access is useful in many circumstances.
- If the file is very large, locating data buried deep inside it can take a long time.

Random Access Files

- Java allows a program to perform *random file access*.
- In random file access, a program may immediately jump to any location in the file.
- To create and work with random access files in Java, you use the `RandomAccessFile` class.

`RandomAccessFile(String filename, String mode)`

- *filename*: the name of the file.
- *mode*: a string indicating the mode in which you wish to use the file.
 - "r" = reading
 - "rw" = for reading and writing.

Random Access Files

```
// Open a file for random reading.  
RandomAccessFile randomFile = new  
    RandomAccessFile("MyData.dat", "r");  
// Open a file for random reading and writing.  
RandomAccessFile randomFile = new  
    RandomAccessFile("MyData.dat", "rw");
```

- When opening a file in "r" mode where the file does not exist, a `FileNotFoundException` will be thrown.
- Opening a file in "r" mode and trying to write to it will throw an `IOException`.
- If you open an existing file in "rw" mode, it will not be deleted and the file's existing content will be preserved.

Random Access Files

- Items in a sequential access file are accessed one after the other.
- Items in a random access file are accessed in any order.
- If you open a file in "rw" mode and the file does not exist, it will be created.
- A file that is opened or created with the `RandomAccessFile` class is treated as a binary file.

Random Access Files

- The `RandomAccessFile` class has:
 - the same methods as the `DataOutputStream` class for writing data, and
 - the same methods as the `DataInputStream` class for reading data.
- The `RandomAccessFile` class can be used to sequentially process a binary file.
- Example: [WriteLetters.java](#)

The File Pointer

- The `RandomAccessFile` class treats a file as a stream of bytes.
- The bytes are numbered:
 - the first byte is byte 0.
 - The last byte's number is one less than the number of bytes in the file.
- These byte numbers are similar to an array's subscripts, and are used to identify locations in the file.
- Internally, the `RandomAccessFile` class keeps a long integer value known as the *file pointer*.

The File Pointer

- The *file pointer* holds the byte number of a location in the file.
- When a file is first opened, the file pointer is set to 0.
- When an item is read from the file, it is read from the byte that the file pointer points to.
- Reading also causes the file pointer to advance to the byte just beyond the item that was read.
- If another item is immediately read, the reading will begin at that point in the file.

The File Pointer

- An `EOFException` is thrown when a read causes the file pointer to go beyond the size of the file.
- Writing also takes place at the location pointed to by the file pointer.
- If the file pointer points to the end of the file, data will be written to the end of the file.
- If the file pointer holds the number of a byte within the file, at a location where data is already stored, a write will overwrite the data at that point.

The File Pointer

- The `RandomAccessFile` class lets you move the file pointer.
- This allows data to be read and written at any byte location in the file.
- The `seek` method is used to move the file pointer.

`rndFile.seek(long position) ;`

- The argument is the number of the byte that you want to move the file pointer to.

The File Pointer

```
RandomAccessFile file = new  
    RandomAccessFile("MyInfo.dat", "r");  
file.seek(99);  
byte b = file.readByte();
```

- Example: [ReadRandomLetters.java](#)

Object Serialization

- If an object contains other types of objects as fields, saving its contents can be complicated.
- Java allows you to *serialize* objects, which is a simpler way of saving objects to a file.
- When an object is serialized, it is converted into a series of bytes that contain the object's data.
- If the object is set up properly, even the other objects that it might contain as fields are automatically serialized.
- The resulting set of bytes can be saved to a file for later retrieval.

Object Serialization

- For an object to be serialized, its class must implement the `Serializable` interface.
- The `Serializable` interface has no methods or fields.
- It is used only to let the Java compiler know that objects of the class might be serialized.
- If a class contains objects of other classes as fields, those classes must also implement the `Serializable` interface, in order to be serialized.
- Example: [BankAccount2.java](#)

Object Serialization

- The `String` class, as many others in the Java API, implements the `Serializable` interface.
- To write a serialized object to a file, you use an `ObjectOutputStream` object.
- The `ObjectOutputStream` class is designed to perform the serialization process.
- To write the bytes to a file, an output stream object is needed.

```
FileOutputStream outStream = new  
    FileOutputStream("Objects.dat");  
ObjectOutputStream objectOutputFile = new  
    ObjectOutputStream(outStream);
```


Object Serialization

- To serialize an object and write it to the file, the `ObjectOutputStream` class's `writeObject` method is used.

```
BankAccount2 account = new  
    BankAccount(25000.0) ;  
objectOutputStream.writeObject(account) ;
```

- The `writeObject` method throws an `IOException` if an error occurs.
- The process of reading a serialized object's bytes and constructing an object from them is known as *deserialization*.

Object Serialization

- To deserialize an object an `ObjectInputStream` object is used in conjunction with a `FileInputStream` object.

```
FileInputStream inStream = new  
    FileInputStream("Objects.dat");  
ObjectInputStream objectInputFile = new  
    ObjectInputStream(inStream);
```

- To read a serialized object from the file, the `ObjectInputStream` class's `readObject` method is used.

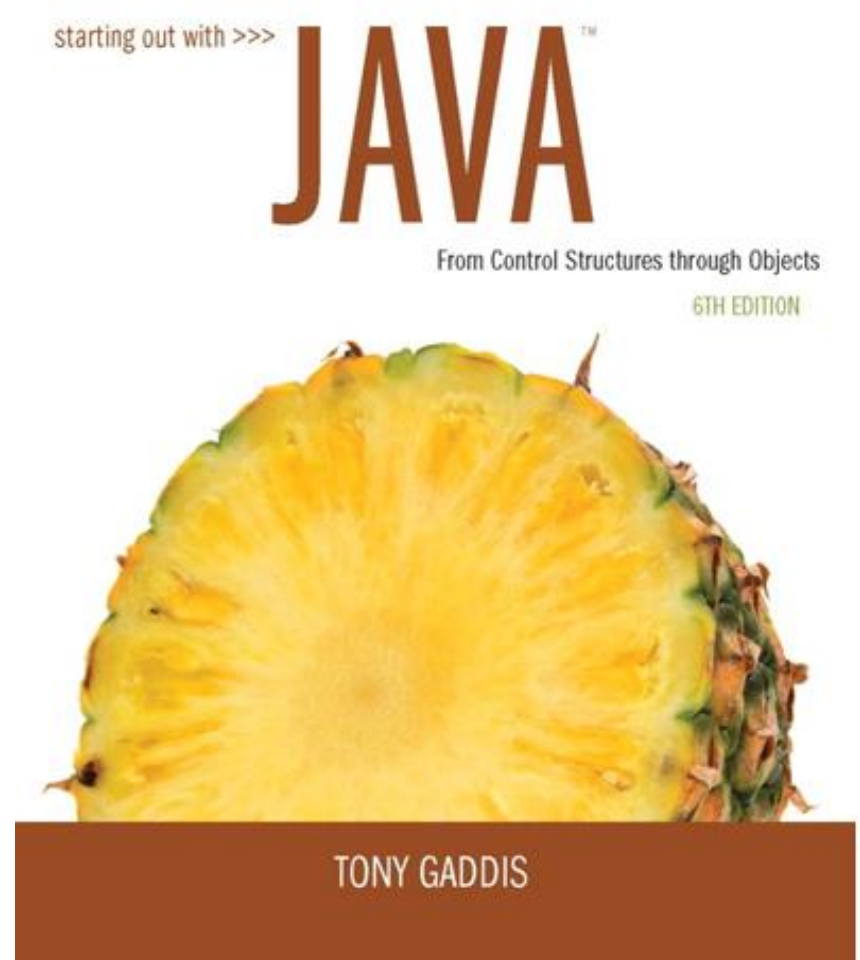
```
BankAccount2 account;  
account = (BankAccount2)  
    objectInputFile.readObject();
```

Object Serialization

- The `readObject` method returns the deserialized object.
 - *Notice that you must cast the return value to the desired class type.*
- The `readObject` method throws a number of different exceptions if an error occurs.
- Examples:
 - [SerializeObjects.java](#)
 - [DeserializeObjects.java](#)

CHAPTER 12

A First Look at GUI Applications



Chapter Topics

Chapter 12 discusses the following main topics:

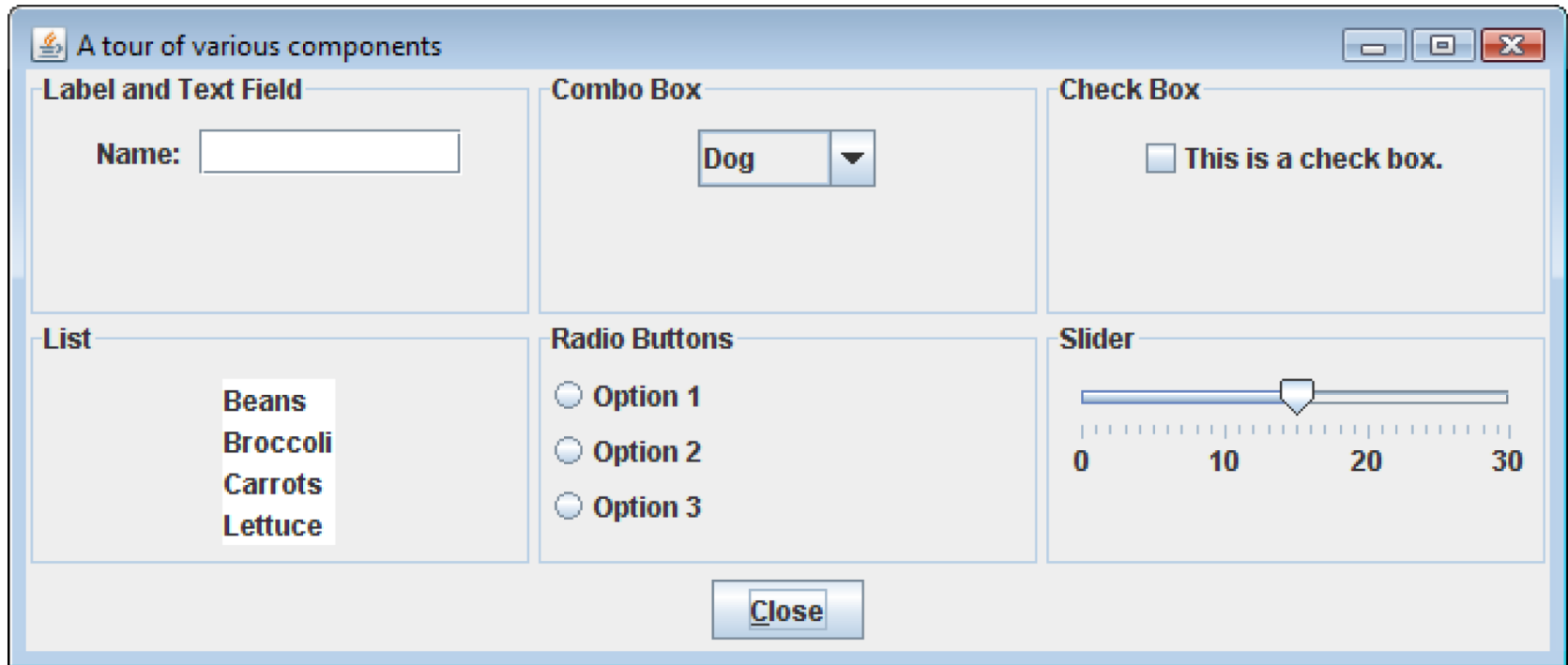
- Introduction
- Creating Windows
- Equipping GUI Classes with a main method
- Layout Managers
- Radio Buttons and Check Boxes
- Borders
- Focus on Problem Solving: Extending Classes from `JPanel`

Introduction

- Many Java application use a *graphical user interface* or *GUI* (pronounced “gooey”).
- A GUI is a graphical window or windows that provide interaction with the user.
- GUI’s accept input from:
 - the keyboard
 - a mouse.
- A window in a GUI consists of *components* that:
 - present data to the user
 - allow interaction with the application.

Introduction

- Some common GUI components are:
 - buttons, labels, text fields, check boxes, radio buttons, combo boxes, and sliders.



JFC, AWT, Swing

- Java programmers use the *Java Foundation Classes (JFC)* to create GUI applications.
- The JFC consists of several sets of classes, many of which are beyond the scope of this book.
- The two sets of JFC classes that we focus on are AWT and Swing classes.
- Java is equipped with a set of classes for drawing graphics and creating graphical user interfaces.
- These classes are part of the *Abstract Windowing Toolkit (AWT)*.

JFC, AWT, Swing

- The AWT allows creation of applications and applets with GUI components.
- The AWT does not actually draw user interface components on the screen.
- The AWT communicates with a layer of software, *peer classes*.
- Each version of Java for a particular operating system has its own set of peer classes.

JFC, AWT, Swing

- Java programs using the AWT:
 - look consistent with other applications on the same system.
 - can offer only components that are common to all the operating systems that support Java.
- The behavior of components across various operating systems can differ.
- Programmers cannot easily extend the AWT components.
- AWT components are commonly called *heavyweight components*.

JFC, AWT, Swing

- Swing was introduced with the release of Java 2.
- *Swing* is a library of classes that provide an improved alternative for creating GUI applications and applets.
- Very few Swing classes rely on peer classes, so they are referred to called *lightweight components*.
- Swing draws most of its own components.
- Swing components have a consistent look and predictable behavior on any operating system.
- Swing components can be easily extended.

Event Driven Programming

- Programs that operate in a GUI environment must be *event-driven*.
- An *event* is an action that takes place within a program, such as the clicking of a button.
- Part of writing a GUI application is creating event listeners.
- An *event listener* is an object that automatically executes one of its methods when a specific event occurs.

javax.swing and java.awt

- In an application that uses Swing classes, it is necessary to use the following statement:

```
import javax.swing.*;
```

- Note the letter `x` that appears after the word `java`.
- Some of the AWT classes are used to determine when events, such as the clicking of a mouse, take place in applications.
- In an application that uses an AWT class, it is necessary to use the following statement.

```
import java.awt.*;
```

Note that there is no `x` after `java` in this package name.

Creating Windows

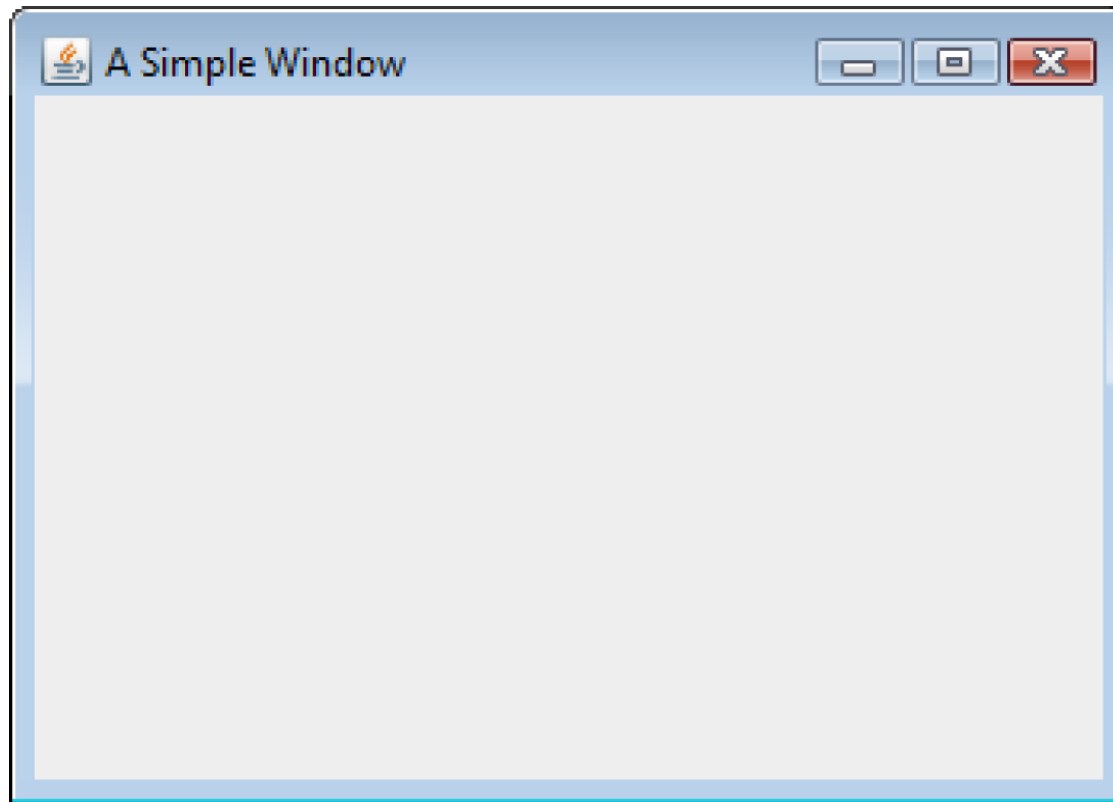
- Often, applications need one or more windows with various components.
- A window is a *container*, which is simply a component that holds other components.
- A container that can be displayed as a window is a *frame*.
- In a Swing application, you create a frame from the `JFrame` class.

Creating Windows

- A frame is a basic window that has:
 - a border around it,
 - a title bar, and
 - a set of buttons for:
 - minimizing,
 - maximizing, and
 - closing the window.
- These standard features are sometimes referred to as window *decorations*.

Creating Windows

- See example: [ShowWindow.java](#)



Creating Windows

- The following `import` statement is needed to use the swing components:

```
import javax.swing.*;
```
- In the `main` method, two constants are declared:

```
final int WINDOW_WIDTH = 350;  
final int WINDOW_HEIGHT = 250;
```
- We use these constants later in the program to set the size of the window.
- The window's size is measured in pixels.
- A *pixel* (*picture element*) is one of the small dots that make up a screen display.

Creating Windows

- An instance of the `JFrame` class needs to be created:
`JFrame window = new JFrame();`
- This statement:
 - creates a `JFrame` object in memory and
 - assigns its address to the `window` variable.
- The string that is passed to the `setTitle` method will appear in the window's title bar when it is displayed.
`window.setTitle("A Simple Window");`
- A `JFrame` is initially invisible.

Creating Windows

- To set the size of the window:

```
window.setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
```

- To specify the action to take place when the user clicks on the close button.

```
window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

- The `setDefaultCloseOperation` method takes an `int` argument which specifies the action.
 - `JFrame.HIDE_ON_CLOSE` - causes the window to be hidden from view, but the application does not end.
 - The default action is `JFrame.HIDE_ON_CLOSE`.

Creating Windows

- The following code displays the window:
`window.setVisible(true);`
- The `setVisible` method takes a boolean argument.
 - `true` - display the window.
 - `false` - hide the window.

Extending JFrame

- We usually use inheritance to create a new class that extends the JFrame class.
- When a new class extends an existing class, it inherits many of the existing class's members just as if they were part of the new class.
- These members act just as if they were written into the new class declaration.
- New fields and methods can be declared in the new class declaration.
- This allows specialized methods and fields to be added to your window.
- Examples: [SimpleWindow.java](#), [SimpleWindowDemo.java](#)

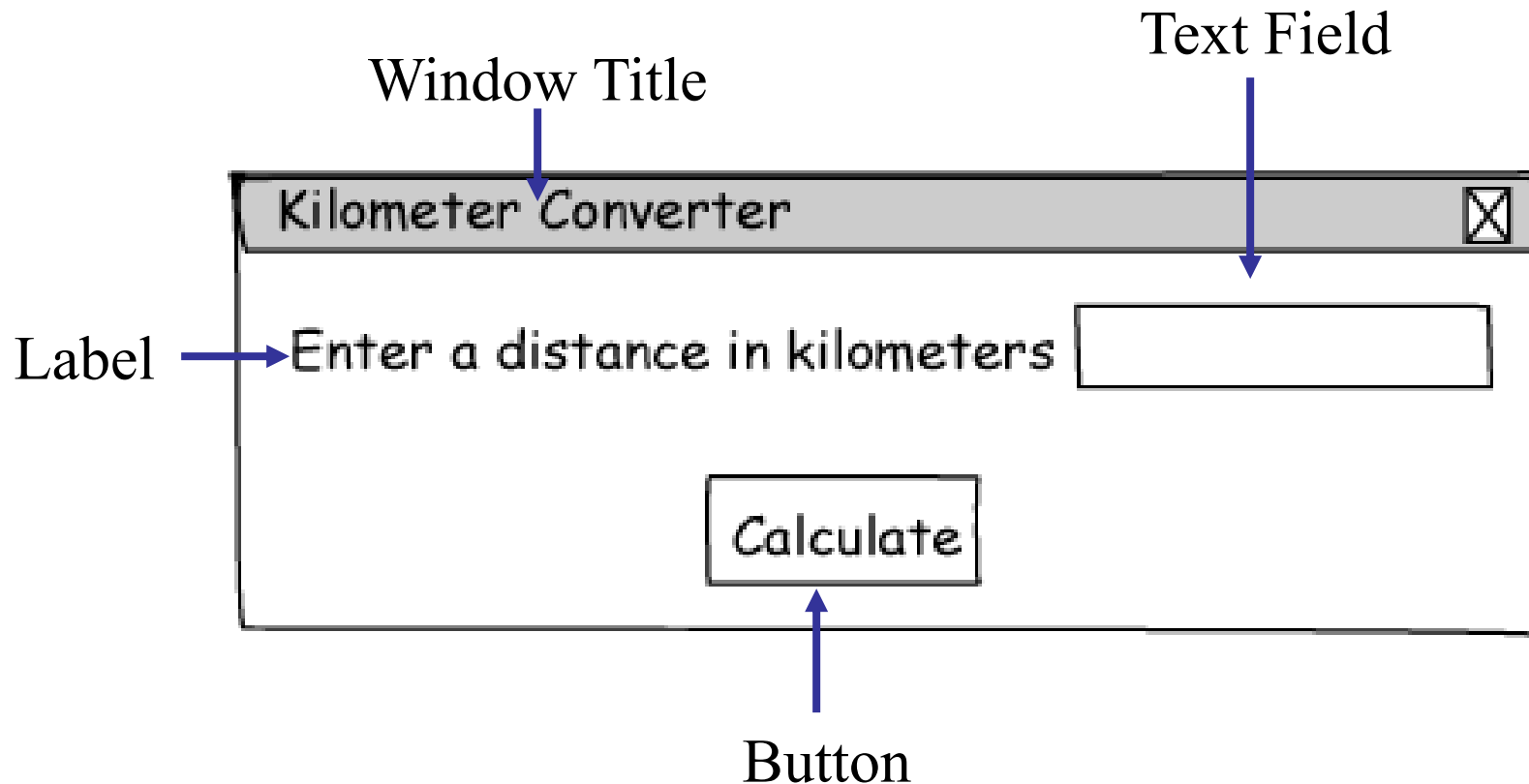
Equipping GUI Classes with a `main` Method

- Java applications always starts execution with a method named `main`.
- The previous example used two separate files:
 - `SimpleWindow.java` -- the class that defines the GUI window
 - `SimpleWindowDemo.java` – contains the `main` method that creates an instance of the `SimpleWindow` class.
- Applications can also be written with the `main` method directly written into the GUI class.
- See example: [EmbeddedMain.java](#)

Adding Components

- Swing provides numerous components that can be added to a window.
- Three fundamental components are:
 - `JLabel` : An area that can display text.
 - `JTextField` : An area in which the user may type a single line of input from the keyboard.
 - `JButton` : A button that can cause an action to occur when it is clicked.

Sketch of Kilometer Converter Graphical User Interface



Adding Components

```
private JLabel message;  
private JTextField kilometers;  
private JButton calcButton;  
...  
message = new JLabel(  
    "Enter a distance in kilometers");  
kilometers = new JTextField(10);  
calcButton = new JButton("Calculate");
```

- This code declares and instantiates three Swing components.

Adding Components

- A *content pane* is a container that is part of every `JFrame` object.
- Every component added to a `JFrame` must be added to its content pane. You do this with the `JFrame` class's `add` method.
- The content pane is not visible and it does not have a border.
- A *panel* is also a container that can hold GUI components.

Adding Components

- Panels cannot be displayed by themselves.
- Panels are commonly used to hold and organize collections of related components.
- Create panels with the `JPanel` class.

```
private JPanel panel;  
...  
panel = new JPanel();  
panel.add(message);  
panel.add(kilometers);  
panel.add(calcButton);
```

Adding Components

- Components are typically placed on a panel and then the panel is added to the `JFrame`'s content pane.

```
add (panel) ;
```

- Examples: [KiloConverter.java](#)

Handling Action Events

- An *event* is an action that takes place within a program, such as the clicking of a button.
- When an event takes place, the component that is responsible for the event creates an *event object* in memory.
- The event object contains information about the event.
- The component that generated the event object is known as the *event source*.
- It is possible that the source component is connected to one or more event listeners.

Handling Action Events

- An *event listener* is an object that responds to events.
- The source component *fires* an event which is passed to a method in the event listener.
- Event listener classes are specific to each application.
- Event listener classes are commonly written as private inner classes in an application.

Writing Event Listener Classes as Private Inner Classes

A class that is defined inside of another class is known as an inner class

```
public class Outer  
{
```

Fields and methods of the Outer class appear here.

```
    private class Inner  
    {
```

Fields and methods of the Inner class appear here.

```
    }  
}
```

Event Listeners Must Implement an Interface

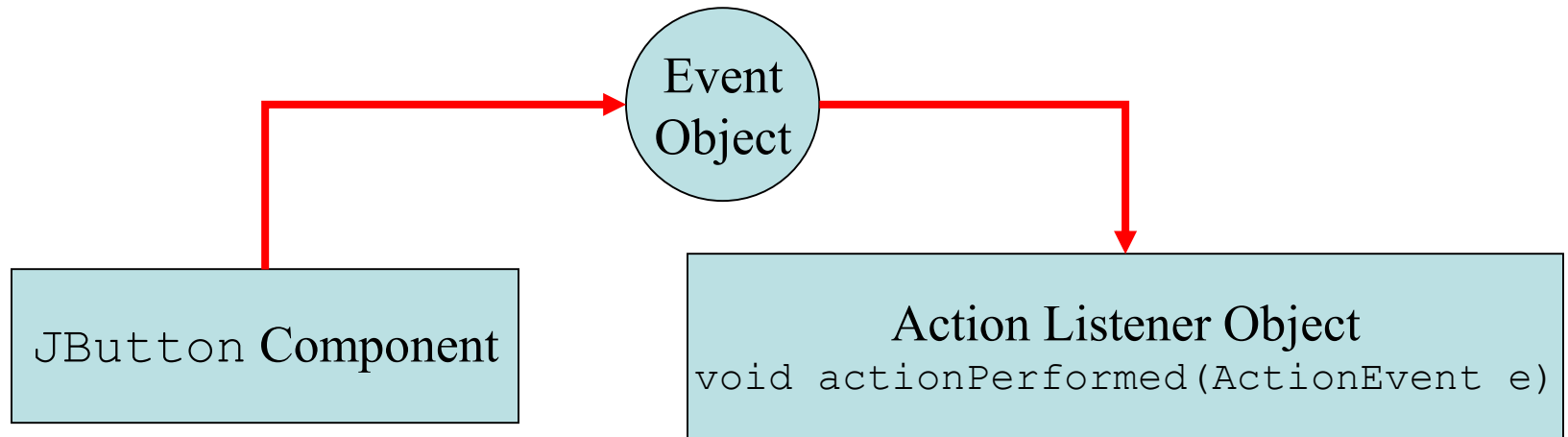
- All event listener classes must *implement an interface*.
- An interface is something like a class containing one or more method headers.
- When you write a class that implements an interface, you are agreeing that the class will have all of the methods that are specified in the interface.

Handling Action Events

- JButton components generate *action events*, which require an *action listener* class.
- Action listener classes must meet the following requirements:
 - It must implement the ActionListener interface.
 - It must have a method named actionPerformed.
- The actionPerformed method takes an argument of the(ActionEvent) type.

```
public void actionPerformed(ActionEvent e)
{
    Code to be executed when button is pressed goes here .
}
```

Handling Action Events



When the button is pressed ...

The `JButton` component generates an event object and passes it to the action listener object's `actionPerformed` method.

Example:

[KiloConverter.java](#)

Registering A Listener

- The process of connecting an event listener object to a component is called *registering* the event listener.
- JButton components have a method named `addActionListener`.

```
calcButton.addActionListener(  
    new CalcButtonListener() );
```

- When the user clicks on the source button, the action listener object's `actionPerformed` method will be executed.

Background and Foreground Colors

- Many of the Swing component classes have methods named `setBackground` and `setForeground`.
- `setBackground` is used to change the color of the component itself.
- `setForeground` is used to change the color of the text displayed on the component.
- Each method takes a color constant as an argument.

Color Constants

- There are predefined constants that you can use for colors.

`Color.BLACK`

`Color.CYAN`

`Color.GRAY`

`Color.LIGHT_GRAY`

`Color.ORANGE`

`Color.RED`

`Color.YELLOW`

`Color.BLUE`

`Color.DARK_GRAY`

`Color.GREEN`

`Color.MAGENTA`

`Color.PINK`

`Color.WHITE`

- Examples: [ColorWindow.java](#)

The `ActionEvent` Object

- Event objects contain certain information about the event.
- This information can be obtained by calling one of the event object's methods.
- Two of these methods are:
 - `getSource` - returns a reference to the object that generated this event.
 - `getActionCommand` - returns the action command for this event as a `String`.
- Example:
 - [EventObject.java](#)

Layout Managers

- An important part of designing a GUI application is determining the layout of the components.
- The term *layout* refers to the positioning and sizing of components.
- In Java, you do not normally specify the exact location of a component within a window.
- A *layout manager* is an object that:
 - controls the positions and sizes of components, and
 - makes adjustments when necessary.

Layout Managers

- The layout manager object and the container work together.
- Java provides several layout managers:
 - `FlowLayout` - Arranges components in rows. This is the default for panels.
 - `BorderLayout` - Arranges components in five regions:
 - North, South, East, West, and Center.
 - This is the default layout manager for a `JFrame` object's content pane.
 - `GridLayout` - Arranges components in a grid with rows and columns.

Layout Managers

- The `Container` class is one of the base classes that many components are derived from.
- Any component that is derived from the `Container` class can have a layout manager added to it.
- You add a layout manager to a container by calling the `setLayout` method.

```
JPanel panel = new JPanel();  
panel.setLayout(new BorderLayout());
```

- In a `JFrame` constructor you might use:
`setLayout(new FlowLayout());`

FlowLayout Manager

- FlowLayout is the default layout manager for JPanel objects.
- Components appear horizontally, from left to right, in the order that they were added. When there is no more room in a row, the next components “flow” to the next row.
- See example: [FlowWindow.java](#)

FlowLayout Manager

- The FlowLayout manager allows you to align components:
 - in the center of each row
 - along the left or right edges of each row.
- An overloaded constructor allows you to pass:
 - `FlowLayout.CENTER`,
 - `FlowLayout.LEFT`, or
 - `FlowLayout.RIGHT`.
- Example:

```
setLayout(new FlowLayout(FlowLayout.LEFT) );
```

FlowLayout Manager

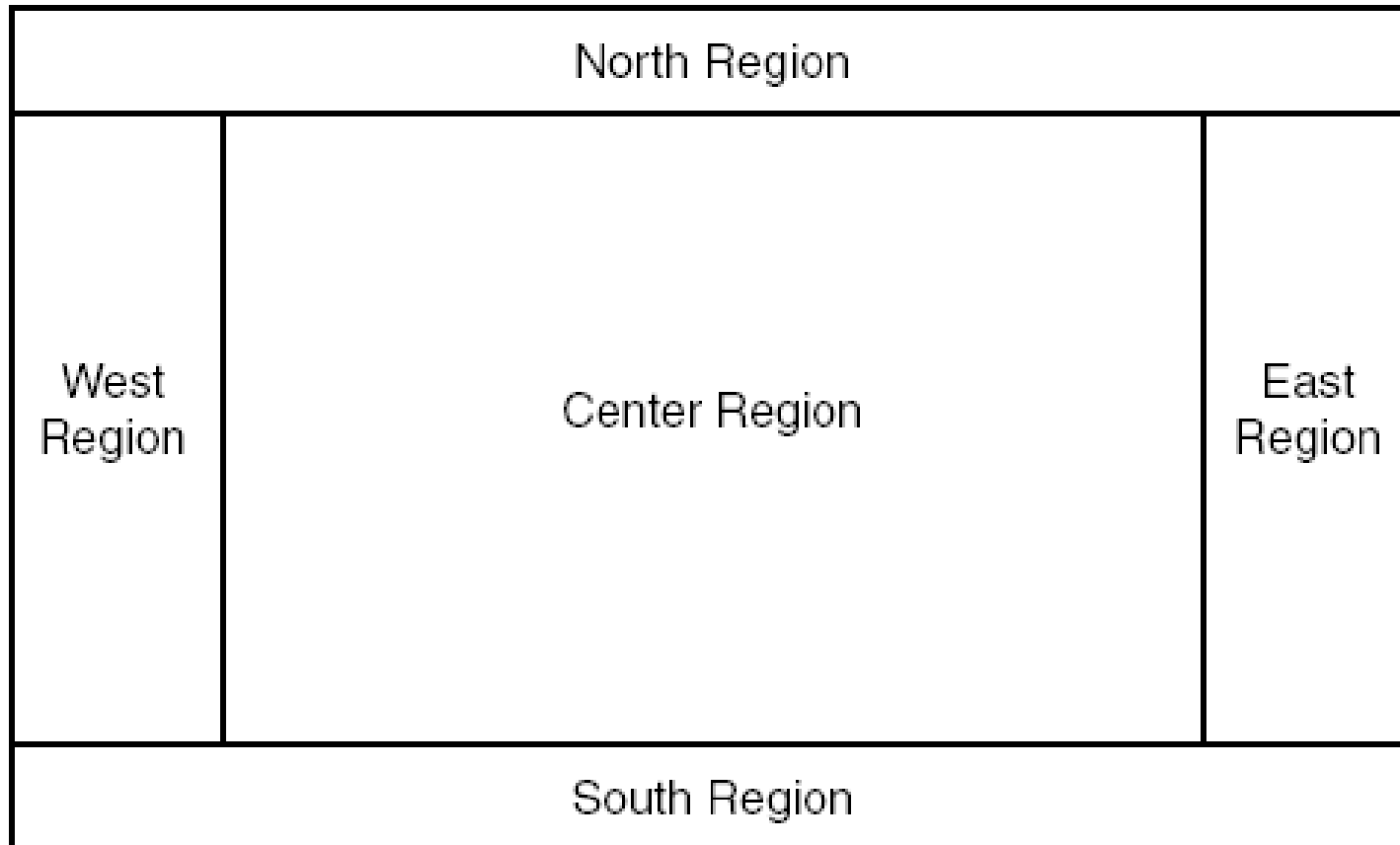
- FlowLayout inserts a gap of five pixels between components, horizontally and vertically.
- An overloaded FlowLayout constructor allows these to be adjusted.
- The constructor has the following format:

```
FlowLayout(int alignment,  
            int horizontalGap,  
            int verticalGap)
```

- Example:
`setLayout(new FlowLayout(FlowLayout.LEFT, 10, 7));`

BorderLayout Manager

BorderLayout manages five regions where components can be placed.



BorderLayout Manager

- See example: [BorderWindow.java](#)
- A component placed into a container that is managed by a BorderLayout must be placed into one of five regions:
 - `BorderLayout.NORTH`
 - `BorderLayout.SOUTH`
 - `BorderLayout.EAST`
 - `BorderLayout.WEST`
 - `BorderLayout.CENTER`

BorderLayout Manager

- Each region can hold only one component at a time.
- When a component is added to a region, it is stretched so it fills up the entire region.
- BorderLayout is the default manager for JFrame objects.

`add(button, BorderLayout.NORTH) ;`

- If you do not pass a second argument to the add method, the component will be added to the center region.

BorderLayout Manager

- Normally the size of a button is just large enough to accommodate the text that it displays
- The buttons displayed in BorderLayout region will not retain their normal size.
- The components are stretched to fill all of the space in their regions.

BorderLayout Manager

- If the user resizes the window, the sizes of the components will be changed as well.
- BorderLayout manager resizes components:
 - placed in the north or south regions may be resized horizontally so it fills up the entire region,
 - placed in the east or west regions may be resized vertically so it fills up the entire region.
 - A component that is placed in the center region may be resized both horizontally and vertically so it fills up the entire region.

BorderLayout Manager

- By default there is no gap between the regions.
- An overloaded `BorderLayout` constructor allows horizontal and vertical gaps to be specified (in pixels).
- The constructor has the following format

```
BorderLayout(int horizontalGap, int verticalGap)
```

- **Example:**

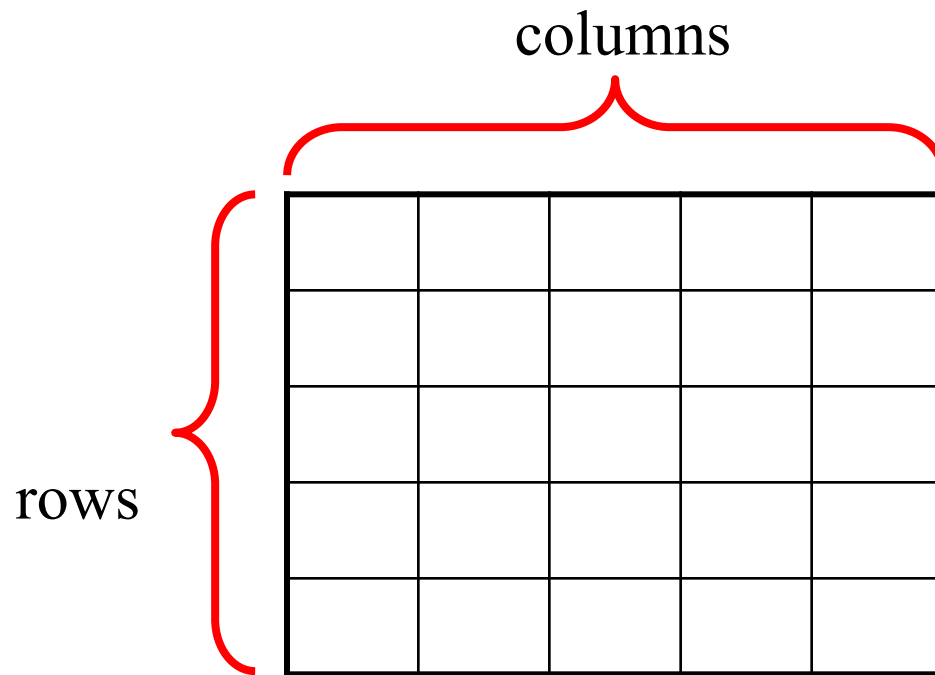
```
setLayout(new BorderLayout(5,10));
```

Nesting Components in a Layout

- Adding components to panels and then nesting the panels inside the regions can overcome the single component limitation of layout regions.
- By adding buttons to a `JPanel` and then adding the `JPanel` object to a region, sophisticated layouts can be achieved.
- See example: [BorderPanelWindow.java](#)

GridLayout Manager

GridLayout creates a grid with rows and columns, much like a spreadsheet. A container that is managed by a GridLayout object is divided into equally sized cells.



GridLayout Manager

- GridLayout manager follows some simple rules:
 - Each cell can hold only one component.
 - All of the cells are the size of the largest component placed within the layout.
 - A component that is placed in a cell is automatically resized to fill up any extra space.
- You pass the number of rows and columns as arguments to the GridLayout constructor.

GridLayout Manager

- The general format of the constructor:

```
GridLayout(int rows, int columns)
```

- Example

```
setLayout(new GridLayout(2, 3));
```

- A zero (0) can be passed for one of the arguments but not both.
 - passing 0 for both arguments will cause an `IllegalArgumentException` to be thrown.

GridLayout Manager

- Components are added to a `GridLayout` in the following order (for a 5×5 grid):

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Example:

[GridWindow.java](#)

`GridLayout` also accepts
nested components:

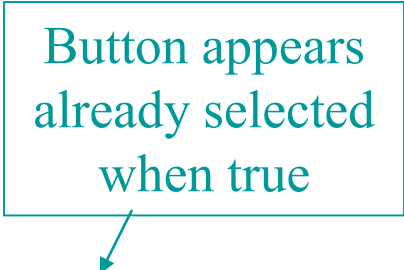
Example:

[GridPanelWindow.java](#)

Radio Buttons

- *Radio buttons* allow the user to select one choice from several possible options.
- The `JRadioButton` class is used to create radio buttons.
- `JRadioButton` constructors:
 - `JRadioButton(String text)`
 - `JRadioButton(String text, boolean selected)`
- Example:

Button appears
already selected
when true



```
JRadioButton radio1 = new JRadioButton("Choice 1");
```

or

```
JRadioButton radio1 = new JRadioButton(  
                                "Choice 1", true);
```


Button Groups

- Radio buttons normally are grouped together.
- In a radio button group only one of the radio buttons in the group may be selected at any time.
- Clicking on a radio button selects it and automatically deselects any other radio button in the same group.
- An instance of the `ButtonGroup` class is used to group radio buttons

Button Groups

- The `ButtonGroup` object creates the *mutually exclusive* relationship between the radio buttons that it contains.

```
JRadioButton radio1 = new JRadioButton("Choice 1",  
                                         true);  
  
JRadioButton radio2 = new JRadioButton("Choice 2");  
JRadioButton radio3 = new JRadioButton("Choice 3");  
ButtonGroup group = new ButtonGroup();  
group.add(radio1);  
group.add(radio2);  
group.add(radio3);
```

Button Groups

- `ButtonGroup` objects are not containers like `JPanel` objects, or content frames.
- If you wish to add the radio buttons to a panel or a content frame, you must add them individually.

```
panel.add (radio1) ;  
panel.add (radio2) ;  
panel.add (radio3) ;
```

Radio Button Events

- `JRadioButton` objects generate an action event when they are clicked.
- To respond to an action event, you must write an action listener class, just like a `JButton` event handler.
- See example: [MetricConverter.java](#)

Determining Selected Radio Buttons

- The `JRadioButton` class's `isSelected` method returns a `boolean` value indicating if the radio button is selected.

```
if (radio.isSelected())  
{  
    // Code here executes if the radio  
    // button is selected.  
}
```

Selecting a Radio Button in Code

- It is also possible to select a radio button in code with the `JRadioButton` class's `doClick` method.
- When the method is called, the radio button is selected just as if the user had clicked on it.
- As a result, an action event is generated.

```
radio.doClick();
```

Check Boxes

- A *check box* appears as a small box with a label appearing next to it.
- Like radio buttons, check boxes may be selected or deselected at run time.
- When a check box is selected, a small check mark appears inside the box.
- Check boxes are often displayed in groups but they are not usually grouped in a `ButtonGroup`.

Check Boxes

- The user is allowed to select any or all of the check boxes that are displayed in a group.
- The `JCheckBox` class is used to create check boxes.

- Two `JCheckBox` constructors :

```
JCheckBox(String text)
```

```
JCheckBox(String text, boolean selected)
```

Check appears
in box if true



- Example:

```
JCheckBox check1 = new JCheckBox("Macaroni");
```

or

```
JCheckBox check1 = new JCheckBox("Macaroni",  
                                true);
```


Check Box Events

- When a `JCheckBox` object is selected or deselected, it generates an *item event*.
- Handling item events is similar to handling action events.
- Write an *item listener* class, which must meet the following requirements:
 - It must implement the `ItemListener` interface.
 - It must have a method named `itemStateChanged`.
 - This method must take an argument of the `ItemEvent` type.

Check Box Events

- Create an object of the class
- Register the item listener object with the `JCheckBox` component.
- On an event, the `itemStateChanged` method of the item listener object is automatically run
 - The event object is passed in as an argument.

Determining Selected Check Boxes

- The `isSelected` method will determine whether a `JCheckBox` component is selected.
- The method returns a `boolean` value.

```
if (checkBox.isSelected())  
{  
    // Code here executes if the check  
    // box is selected.  
}
```

- See example: [ColorCheckBoxWindow.java](#)

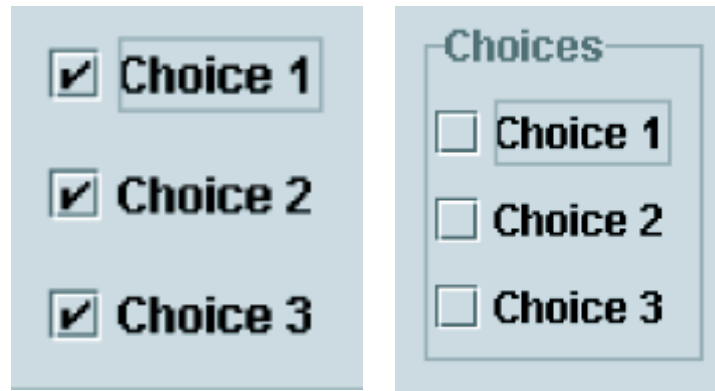
Selecting Check Boxes in Code

- It is possible to select check boxes in code with the `JCheckBox` class's `doClick` method.
- When the method is called, the check box is selected just as if the user had clicked on it.
- As a result, an item event is generated.

```
checkBox.doClick();
```

Borders

- Windows have a more organized look if related components are grouped inside borders.



- You can add a border to any component that is derived from the `JComponent` class.
 - Any component derived from `JComponent` inherits a method named `setBorder`

Borders

- The `setBorder` method is used to add a border to the component.
- The `setBorder` method accepts a `Border` object as its argument.
- A `Border` object contains detailed information describing the appearance of a border.
- The `BorderFactory` class, which is part of the `javax.swing` package, has static methods that return various types of borders.

Border	BorderFactory Method	Description
Compound border	<code>createCompoundBorder</code>	A border that has two parts: an inside edge and an outside edge. The inside and outside edges can be any of the other borders.
Empty border	<code>createEmptyBorder</code>	A border that contains only empty space.
Etched border	<code>createEtchedBorder</code>	A border with a 3D appearance that looks “etched” into the background.
Line border	<code>createLineBorder</code>	A border that appears as a line.
Lowered bevel border	<code>createLoweredBevelBorder</code>	A border that looks like beveled edges. It has a 3D appearance that gives the illusion of being sunken into the surrounding background.
Matte border	<code>createMatteBorder</code>	A line border that can have edges of different thicknesses.
Raised bevel border	<code>createRaisedBevelBorder</code>	A border that looks like beveled edges. It has a 3D appearance that gives the illusion of being raised above the surrounding background.
Titled border	<code>createTitledBorder</code>	An etched border with a title.

The Brandi's Bagel House Application

- A complex application that uses numerous components can be constructed from several specialized panel components, each containing other components and related code such as event listeners.
- Examples:

[GreetingPanel.java](#), [BagelPanel.java](#),
[ToppingPanel.java](#), [CoffeePanel.java](#),
[OrderCalculatorGUI.java](#)

Splash Screens

- A splash screen is a graphic image that is displayed while an application loads into memory and starts up.
- A splash screen keeps the user's attention while a large application loads and executes.
- Beginning with Java 6, you can display splash screens with your Java applications.

Splash Screens

- To display the splash screen you use the `java` command in the following way when you run the application:

```
java -splash:GraphicFileName ClassFileName
```

- *GraphicFileName* is the name of the file that contains the graphic image, and *ClassFileName* is the name of the *.class* file that you are running.
- The graphic file can be in the GIF, PNG, or JPEG formats.

Using Console Output to Debug a GUI

- Display variable values, etc. as your application executes to identify logic errors
 - Use `System.out.println()`

```
// For debugging, display the text entered, and
// its value converted to a double.
System.out.println("Reading " + str +
    " from the text field.");
System.out.println("Converted value: " +
    Double.parseDouble(str));
```

- See example: [KiloConverter.java](#)

CHAPTER 13

Advanced GUI Applications

starting out with >>>

JAVA™

From Control Structures through Objects

6TH EDITION



TONY GADDIS

Chapter Topics

Chapter 13 discusses the following main topics:

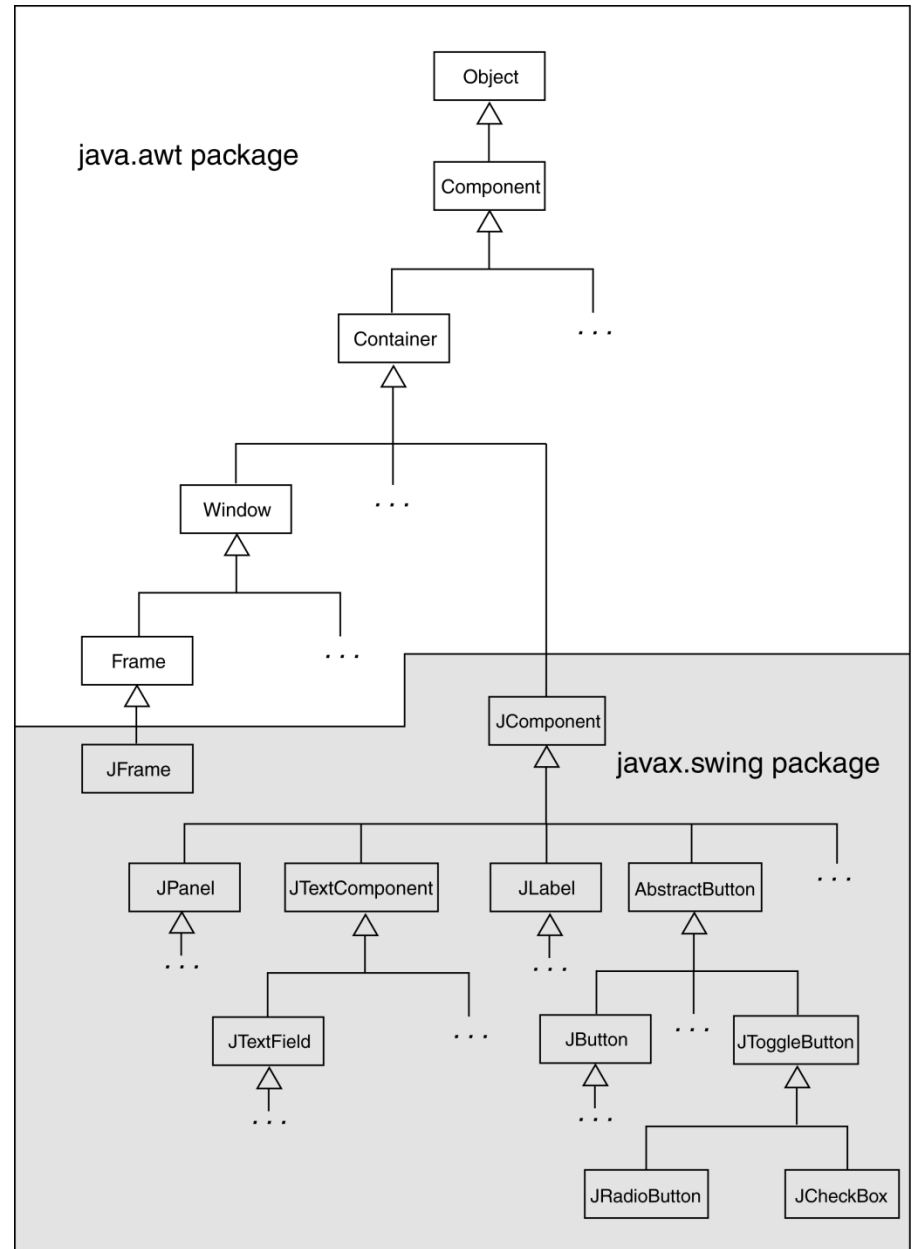
- The Swing and AWT Class Hierarchy
- Read-Only Text Fields
- Lists
- Combo Boxes
- Displaying Images in Labels and Buttons
- Mnemonics and Tool Tips

Chapter Topics

Chapter 13 discusses the following main topics:

- File Choosers and Color Choosers
- Menus
- More about Text Components: Text Areas and Fonts
- Sliders
- Look and Feel

The Swing and AWT Class Hierarchy



Read Only Text Fields

- Read only text fields are a different way to use the `JTextField` component.
- The `JTextField` component has a method named `setEditable`:

`setEditable(boolean editable)`

- By default a text field is editable.
- The `setEditable` method must be called and passed false to make the field read-only.

Lists

- A *list* is a component that displays a list of items and allows the user to select items from the list.
- The `JList` component is used for creating lists.
- When an instance of the `JList` class is created, an array of objects is passed to the constructor.
`JList (Object[] array)`
- The `JList` component uses the array to create the list of items.

```
String[] names = { "Bill", "Geri", "Greg", "Jean",  
    "Kirk", "Phillip", "Susan" };  
JList nameList = new JList(names);
```

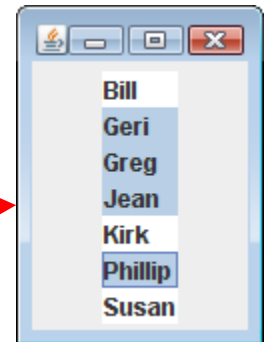
List Selection Modes

- The `JList` component can operate in any of the following selection modes:
 - **Single Selection Mode** - Only one item can be selected at a time.
 - **Single Interval Selection Mode** - Multiple items can be selected, but they must be in a single interval. An interval is a set of contiguous items.
 - **Multiple Interval Selection Mode** - In this mode multiple items may be selected with no restrictions.
 - This is the default selection mode.

List Selection Modes



Single selection mode allows only one item to be selected at a time.



Multiple interval selection mode allows multiple items to be selected with no restrictions.



Single interval selection mode allows a single interval of contiguous items to be selected.

List Selection Modes

- You change a `JList` component's selection mode with the `setSelectionMode` method.
- The method accepts an `int` argument that determines the selection mode:
 - `ListSelectionModel.SINGLE_SELECTION`
 - `ListSelectionModel.SINGLE_INTERVAL_SELECTION`
 - `ListSelectionModel.MULTIPLE_INTERVAL_SELECTION`
- Example:

```
nameList.setSelectionMode(  
    ListSelectionModel.SINGLE_SELECTION);
```

List Events

- When an item in a `JList` object is selected it generates a *list selection event*.
- The event is handled by an instance of a *list selection listener* class, which must meet the following requirements:
 - It must implement the `ListSelectionListener` interface.
 - It must have a method named `valueChanged`. This method must take an argument of the `ListSelectionEvent` type.
- Use the `addListSelectionListener` method of the `JList` class to register the instance of the list selection listener class with the list object.

List Events

- When the `JList` component generates an event:
 - it automatically executes the `valueChanged` method of the list selection listener object
 - It passes the event object as an argument.

Retrieving Selected Items

- You may use:
 - `getSelectedValue` or
 - `getSelectedIndex`
 - to determine which item in a list is currently selected.
- `getSelectedValue` returns a reference to the item that is currently selected.

```
String selectedName;  
selectedName = (String)nameList.getSelectedValue();
```
- The return value must be cast to `String` is required in order to store it in the `selectedName` variable.
- If no item in the list is selected, the method returns `null`.

Retrieving Selected Items

- The `getSelectedIndex` method returns the index of the selected item, or `-1` if no item is selected.
- Internally, the items that are stored in a list are numbered (similar to an array).
- Each item's number is called its *index*.
- The first item has the index 0.
- You can use the index of the selected item to retrieve the item from an array.

```
String[] names = { "Bill", "Geri", "Greg", "Jean",  
    "Kirk", "Phillip", "Susan" };  
JList nameList = new JList(names);
```


Retrieving Selected Items

- This code could be used to determine the selected item:

```
int index;  
String selectedName;  
index = nameList.getSelectedIndex();  
if (index != -1)  
    selectedName = names[index];
```

- Example: [ListWindow.java](#)

Bordered Lists

- The `setBorder` method can be used to draw a border around a `JList`.

```
monthList.setBorder(  
    BorderLayout.createLineBorder(Color.black,1));
```



Adding A Scroll Bar To a List

- By default, a list component is large enough to display all of the items it contains.
- Sometimes a list component contains too many items to be displayed at once.
- Most GUI applications display a scroll bar on list components that contain a large number of items.
- List components do not automatically display a scroll bar.

Adding A Scroll Bar To a List

- To display a scroll bar on a list component, follow these general steps.
 1. Set the number of visible rows for the list component.
 2. Create a scroll pane object and add the list component to it.
 3. Add the scroll pane object to any other containers, such as panels.
- For this list:

```
String[] names = { "Bill", "Geri", "Greg", "Jean",  
    "Kirk", "Phillip", "Susan" };  
JList nameList = new JList(names);
```

Adding A Scroll Bar To a List

- Establish the size of the list component.
`nameList.setVisibleRowCount(3);`
- Create a scroll pane object and add the list component to it.
- A *scroll pane object* is a container that displays scroll bars on any component it contains.
- The `JScrollPane` class to create a scroll pane object.
- We pass the object that we wish to add to the scroll pane as an argument to the `JScrollPane` constructor.

```
JScrollPane scrollPane = new  
    JScrollPane(nameList);
```

Adding A Scroll Bar To a List

- Add the scroll pane object to any other containers that are necessary for our GUI.

```
JPanel panel = new JPanel();  
panel.add(scrollPane);  
add(panel);
```

- When the list component is displayed, it will appear with:
 - Three items showing at a time and
 - scroll bars:

Adding A Scroll Bar To a List

- By default, `JList` components added to a `JScrollPane` object only display a scroll bar if there are more items in the list than there are visible rows.
- When a `JList` component is added to a `JScrollPane` object, a border will automatically appear around the list.
- Example: [ListWindowWithScroll.java](#)

Adding Items to an Existing List

- The `setListData` method allows the adding of items in an existing `JList` component.

```
void setListData(Object[] data)
```

- This replaces any items that are currently displayed in the component.
- This can be used to add items to an empty list.

Adding Items to an Existing List

- You can create an empty list by using the `JList` component's no-parameter constructor:

```
JList nameList = new JList();
```

- Items can be added to the list:

```
String[] names = { "Bill", "Geri", "Greg", "Jean",  
    "Kirk", "Phillip", "Susan" };  
nameList.setListData(names);
```

Single Interval Selection Mode

- A list is set to single interval selection mode by passing the constant
`ListSelectionModel.SINGLE_INTERVAL_SELECTION`
to the component's `setSelectionMode` method.
- An interval is a set of contiguous items.
- The user selects:
 - the first item in the interval by clicking on it
 - the last item by holding the Shift key while clicking on it.
- All of the items that appear in the list from the first item through the last item are selected.

Single Interval Selection Mode

- The `getSelectedValue` method returns the first item in the selected interval.
- The `getSelectedIndex` method returns the index of the first item in the selected interval.
- To get the entire selected interval, use the `getSelectedValues` method.
 - This method returns an array of objects, which are the items in the selected interval.
- The `getSelectedIndices` method returns an array of int values that are the indices of all the selected items in the list.

Multiple Interval Selection Mode

- Set multiple interval selection mode by passing the constant `ListSelectionModel.MULTIPLE_INTERVAL_SELECTION` to the component's `setSelectionMode` method.
- In multiple interval selection mode:
 - multiple items can be selected
 - the items do not have to be in the same interval.
- In multiple interval selection mode the user can select single items or intervals.

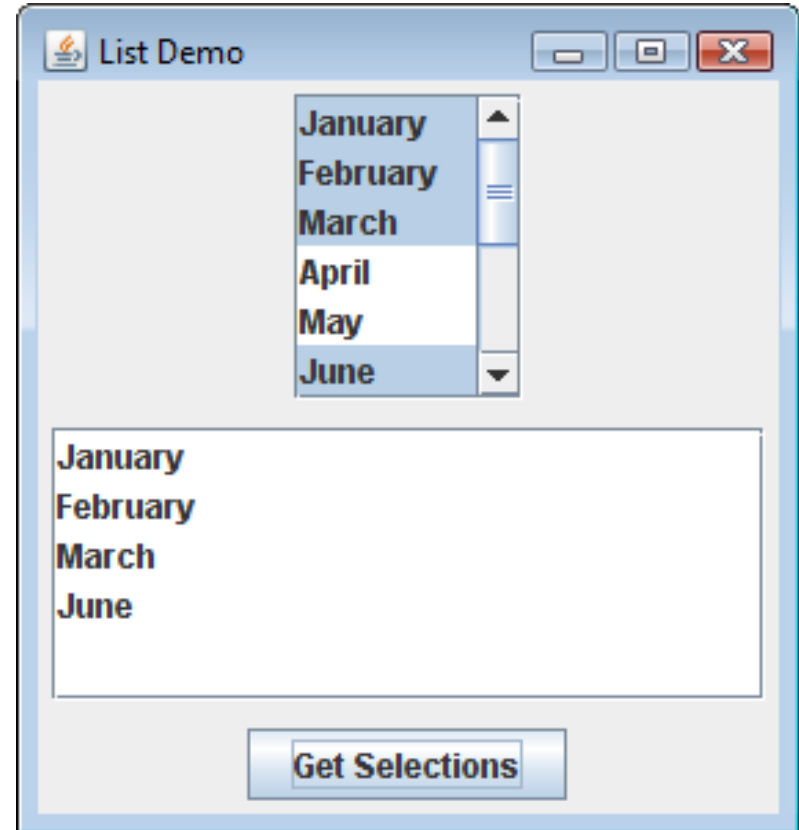
Multiple Interval Selection Mode

- The user holds down the Ctrl key while clicking on an item
 - it selects the item without deselecting other items.
- The `getSelectedValue` method returns the first selected item.
- The `getSelectedIndex` method returns the index of the first selected item.
- The `getSelectedValues` method returns an array of objects containing the items that are selected.
- The `getSelectedIndices` method returns an `int` array containing the indices of the selected items.

Multiple Interval Selection Mode

Example:

[MultipleIntervalSelection.java](#)



Combo Boxes

- A combo box presents a drop-down list of items that the user may select from.
- The `JComboBox` class is used to create a combo box.
- Pass an array of objects that are to be displayed as the items in the drop-down list to the constructor.

```
String[] names = { "Bill", "Geri", "Greg", "Jean",  
    "Kirk", "Phillip", "Susan" };  
JComboBox nameBox = new JComboBox(names);
```

- When displayed, the combo box created by this code will initially appear as the button:



Combo Boxes

- The button displays the item that is currently selected.
- The first item in the list is automatically selected when the combo box is displayed.
- When the user clicks on the button, the drop-down list appears and the user may select another item.



Combo Box Events

- When an item in a `JComboBox` object is selected, it generates an action event.
- Handle action events with an action event listener class, which must have an `actionPerformed` method.
- When the user selects an item in a combo box, the combo box executes its action event listener's `actionPerformed` method, passing an `ActionEvent` object as an argument.

Retrieving Selected Items

- There are two methods in the `JComboBox` class that can be used to determine which item in a list is currently selected:
 - `getSelectedItem`
 - `getSelectedIndex`
- The `getSelectedItem` method returns a reference to the item that is currently selected.

```
String selectedName;  
selectedName = (String) nameBox.getSelectedItem();
```
- `getSelectedItem` returns an `Object` reference so we cast the return value to a `String`.

Retrieving Selected Items

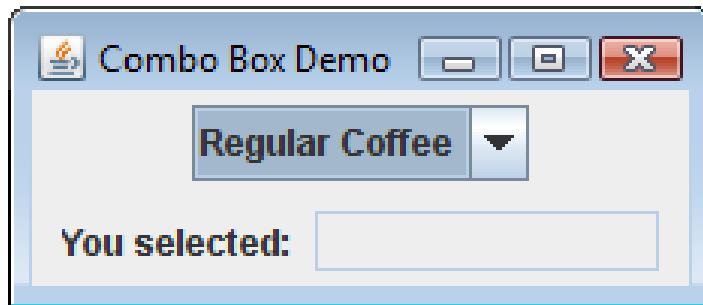
- The `getSelectedIndex` method returns the index of the selected item.

```
String[] names = { "Bill", "Geri", "Greg", "Jean",  
    "Kirk", "Phillip", "Susan" };  
JComboBox nameBox = new JComboBox(names);
```

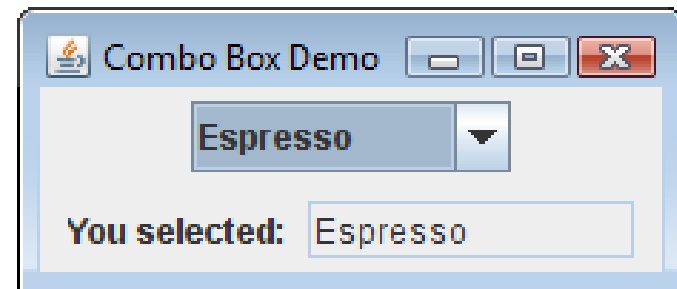
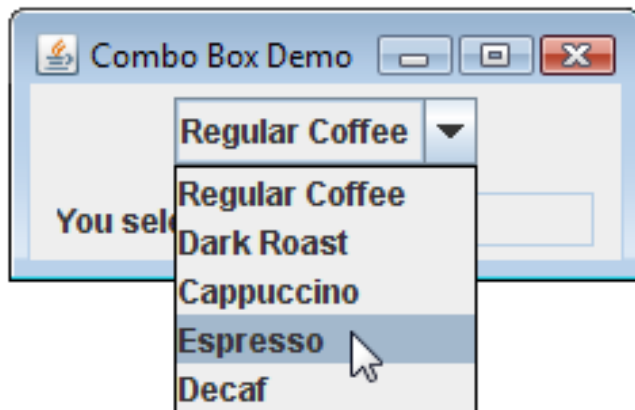
- Get the selected item from the names array:

```
int index;  
String selectedName;  
index = nameBox.getSelectedIndex();  
selectedName = names[index];
```

Retrieving Selected Items



- Example:
- [ComboBoxWindow.java](#)



Editable Combo Boxes

- There are two types of combo boxes:
 - uneditable – allows the user to only select items from its list.
 - editable – combines a text field and a list.
 - It allows the selection of items from the list
 - allows the user to type input into the text field
- The `setEditable` method sets the edit mode for the component.

```
String[] names = { "Bill", "Geri", "Greg", "Jean",  
    "Kirk", "Phillip", "Susan" };  
JComboBox nameBox = new JComboBox(names);  
nameBox.setEditable(true);
```

Editable Combo Boxes

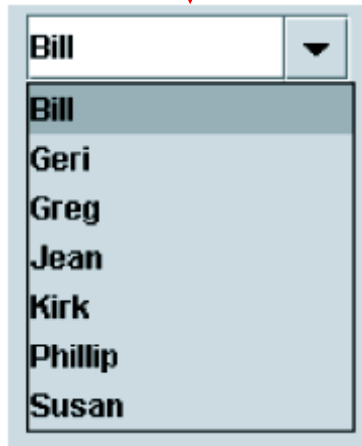
- An editable combo box appears as a text field with a small button displaying an arrow joining it.
- When the user clicks on the button, the drop-down list appears as shown in the center of the figure.
- The user may:
 - select an item from the list.
 - type a value into the text field.
- The user is not restricted to the values that appear in the list, and may type any input into the text field.

Editable Combo Boxes

Note that Sharon is not in the list.

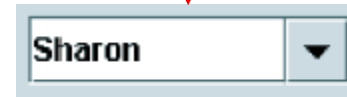


Bill



Bill

- Bill
- Geri
- Greg
- Jean
- Kirk
- Phillip
- Susan



Sharon

Displaying Images in Labels and Buttons

- Labels can display text, an image, or both.
- To display an image, create an instance of the `ImageIcon` class, which reads the image file.
- The constructor accepts the name of an image file.
- The supported file types are JPEG, GIF, and PNG.
- The name can also contain path information.

```
ImageIcon image = new ImageIcon("Smiley.gif");
```

or

```
ImageIcon image = new ImageIcon(  
    "C:\\Chapter 12\\Images\\Smiley.gif");
```


Displaying Images in Labels and Buttons

- Display the image in a label by passing the `ImageIcon` object as an argument to the `JLabel` constructor.

```
JLabel(Icon image)
```

- The argument passed can be an `ImageIcon` object or any object that implements the `Icon` interface.

```
ImageIcon image = new ImageIcon("Smiley.gif");  
JLabel label = new JLabel(image);
```

or

```
JLabel label = new JLabel("Have a nice day!");  
label.setIcon(image);
```

Displaying Images in Labels and Buttons

- Text is displayed to the right of images by default.
- Text alignment can be modified by passing one of the following to an overloaded constructor:
 - `SwingConstants.LEFT`
 - `SwingConstants.CENTER`
 - `SwingConstants.RIGHT`
- Example:

```
ImageIcon image = new ImageIcon("Smiley.gif");  
JLabel label = new JLabel("Have a nice day!",  
                           image,  
                           SwingConstants.RIGHT);
```

Displaying Images in Labels and Buttons

- Creating a button with an image is similar to that of creating a label with an image.

```
ImageIcon image = new ImageIcon("Smiley.gif");  
JButton button = new JButton(image);
```

- To create a button with an image and text:

```
ImageIcon image = new ImageIcon("Smiley.gif");  
JButton button = new JButton(  
    "Have a nice day!", image);  
button.setIcon(image);
```

Displaying Images in Labels and Buttons

- To add an image to an existing button:

```
JButton button = new JButton(  
    "Have a nice day!");  
ImageIcon image = new ImageIcon("Smiley.gif");  
button.setIcon(image);
```

- You are not limited to small graphical icons when placing images in labels or buttons.
- Example: [MyCatImage.java](#)

Mnemonics

- A *mnemonic* is a key that you press in combination with the Alt key to quickly access a component.
- These are sometimes referred to as hot keys.
- A hot key is assigned to a component through the component's `setMnemonic` method
- The argument passed to the method is an integer code that represents the key you wish to assign.

Mnemonics

- The key codes are predefined constants in the `KeyEvent` class (`java.awt.event` package).
- These constants take the form:
 - `KeyEvent.VK_x`, where *x* is a key on the keyboard.
 - The letters VK in the constants stand for “virtual key”.
 - To assign the A key as a mnemonic, use `KeyEvent.VK_A`.
- Example:

```
JButton exitButton = new JButton("Exit");  
exitButton.setMnemonic(KeyEvent.VK_X);
```

Mnemonics

- If the letter is in the component's text, the first occurrence of that letter will appear underlined.
- If the letter does not appear in the component's text, then no letter will appear underlined.

Mnemonics

- You can also assign mnemonics to radio buttons and check boxes:

```
JRadioButton rb1 = new  
    JRadioButton("Breakfast");  
rb1.setMnemonic(KeyEvent.VK_B);  
JRadioButton rb2 = new JRadioButton("Lunch");  
rb2.setMnemonic(KeyEvent.VK_L);  
JCheckBox cb1 = new JCheckBox("Monday");  
cb1.setMnemonic(KeyEvent.VK_M);  
JCheckBox cb2 = new JCheckBox("Wednesday");  
cb2.setMnemonic(KeyEvent.VK_W);
```


Tool Tips

- A *tool tip* is text that is displayed in a small box when the mouse is held over a component.
- The box usually gives a short description of what the component does.
- Most GUI applications use tool tips as concise help to the user.

Tool Tips

- Assign a tool tip to a component with the `setToolTipText` method.

```
JButton exitButton = new JButton("Exit");  
exitButton.setMnemonic(KeyEvent.VK_X);  
exitButton.setToolTipText(  
    "Click here to exit.");
```

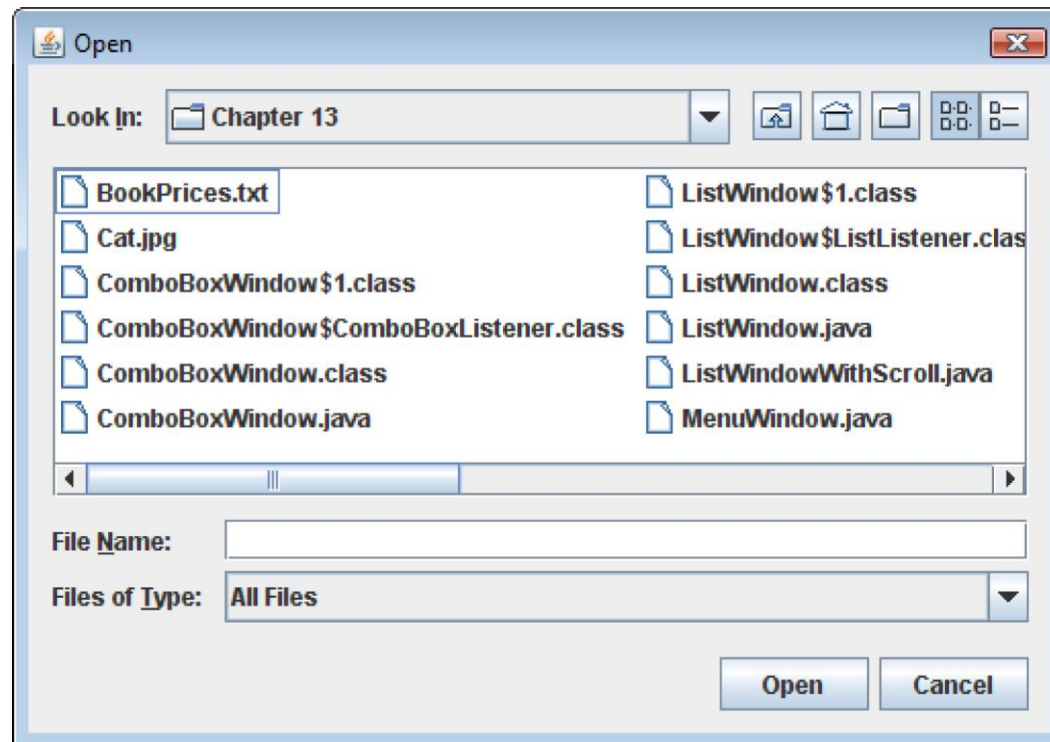


Note the mnemonic x.

Tool tip

File Choosers

- A file chooser is a specialized dialog box that allows the user to browse for a file and select it.



File Choosers

- Create an instance of the `JFileChooser` class to display a file chooser dialog box.
- Two of the constructors have the form:
`JFileChooser()`
`JFileChooser(String path)`
- The first constructor shown takes no arguments and uses the default directory as the starting point for all of its dialog boxes.
- The second constructor takes a `String` argument containing a valid path. This path will be the starting point for the object's dialog boxes.

File Choosers

- A `JFileChooser` object can display two types of predefined dialog boxes:
 - open file dialog box – lets the user browse for an existing file to open.
 - a save file dialog box – lets the user browse to a location to save a file.

File Choosers

- To display an open file dialog box, use the `showOpenDialog` method.
- General format:
`int showOpenDialog(Component parent)`
- The argument can be null or a reference to a component.
- If null is passed, the dialog box is normally centered in the screen.
- If you pass a reference to a component the dialog box is displayed over the component.

File Choosers

- To display a save file dialog box, use the `showSaveDialog` method.
- General format:
`int showSaveDialog(Component parent)`
- The argument can be either null or a reference to a component.
- Both methods return an integer that indicates the action taken by the user to close the dialog box.

File Choosers

- You can compare the return value to one of the following constants:
 - `JFileChooser.CANCEL_OPTION` – indicates that the user clicked on the Cancel button.
 - `JFileChooser.APPROVE_OPTION` – indicates that the user clicked on the OK button.
 - `JFileChooser.ERROR_OPTION` – indicates that an error occurred, or the user clicked on the standard close button on the window to dismiss it.
- If the user selected a file, use the `getSelectedFile` method to determine the file that was selected.
- The `getSelectedFile` method returns a `File` object, which contains data about the selected file.

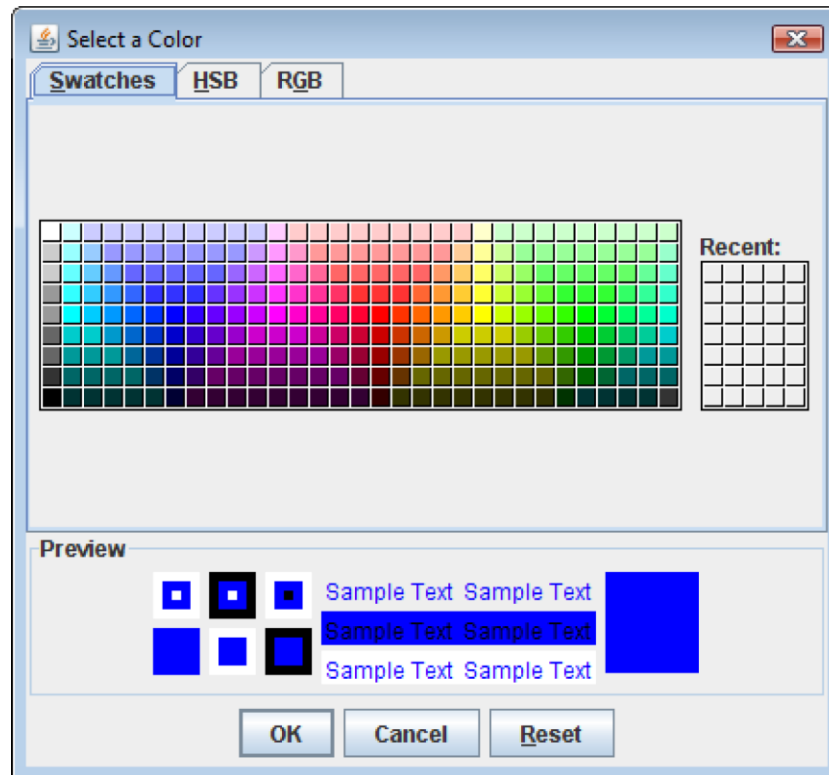
File Choosers

- Use the `File` object's `getPath` method to get the path and file name as a `String`.

```
JFileChooser fileChooser = new JFileChooser();
int status = fileChooser.showOpenDialog(null);
if (status == JFileChooser.APPROVE_OPTION)
{
    File selectedFile =
        fileChooser.getSelectedFile();
    String filename = selectedFile.getPath();
    JOptionPane.showMessageDialog(null,
        "You selected " + filename);
}
```

Color Choosers

- A color chooser is a specialized dialog box that allows the user to select a color from a predefined palette of colors.



Color Choosers

- By clicking the HSB tab you can select a color by specifying its hue, saturation, and brightness.
- By clicking the RGB tab you can select a color by specifying its red, green, and blue components.
- The `JColorChooser` class has a static method named `showDialog`, with the following general format:

```
Color showDialog(Component parent,  
                  String title, Color initial)
```

Color Choosers

- If the first argument is `null`, the dialog box is normally centered in the screen.
- If it is a reference to a component the dialog box is displayed over the component.
- The second argument is the dialog title.
- The third argument indicates the color that appears initially selected in the dialog box.
- This method returns the color selected by the user.

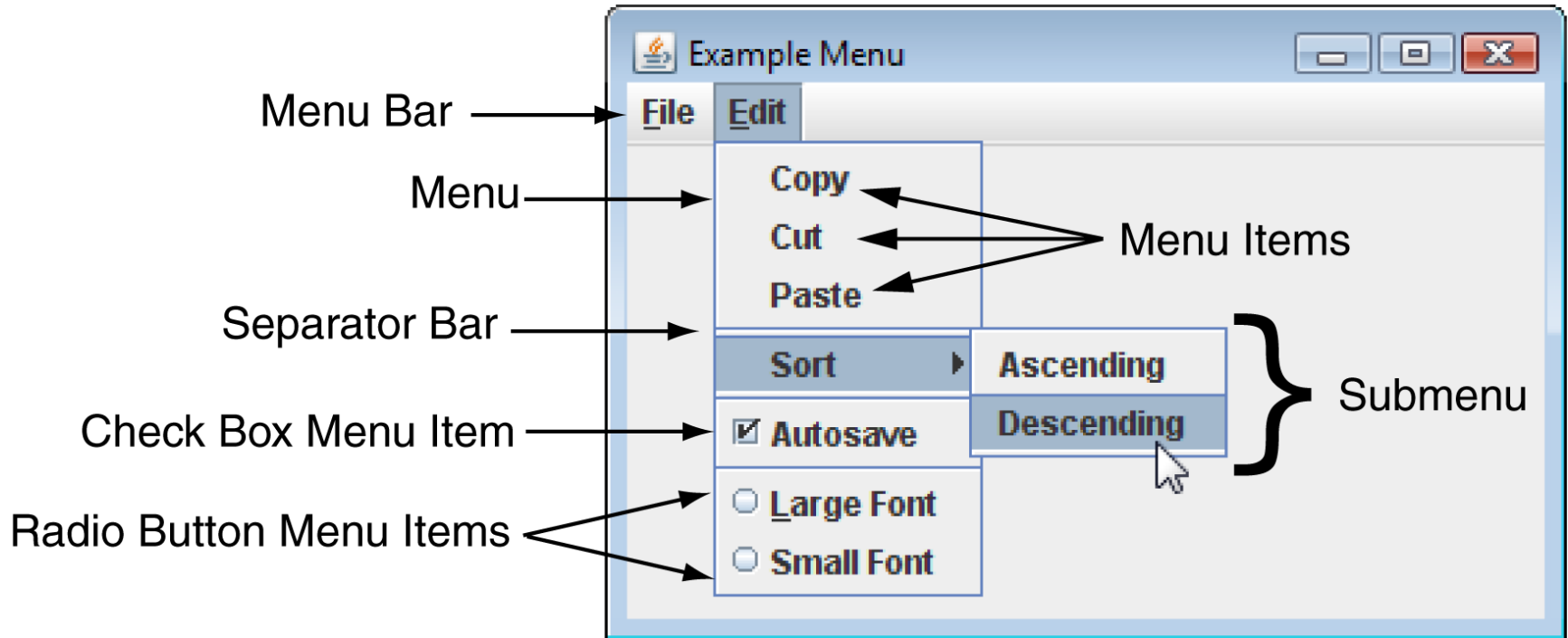
Color Choosers

- Example:

```
JPanel panel = new JPanel();  
Color selectedColor =  
    JColorChooser.showDialog(null,  
        "Select a Background Color",  
        Color.BLUE);  
panel.setBackground(selectedColor);
```

Menus

- A *menu system* is a collection of commands organized in one or more drop-down menus.



Components of A Menu System

- A menu system commonly consists of:
 - **Menu Bar** – A *menu bar* lists the names of one or menus.
 - **Menu** – A *menu* is a drop-down list of menu items.
 - **Menu Item** – A *menu item* can be selected by the user.
 - **Check box menu item** – A *check box menu item* appears with a small box beside it.
 - The item may be selected or deselected.
 - **Radio button menu item** – A *radio button menu item* may be selected or deselected.
 - **Submenu** – A menu within a menu is called a *submenu*.
 - **Separator bar** – A separator bar is a horizontal bar used to separate groups of items on a menu.

Menu Classes

- A menu system is constructed with the following classes:
 - JMenuBar – Used to create a menu bar.
 - A JMenuBar object can contain JMenu components.
 - JMenu – Used to create a menu. A JMenu component can contain:
 - JMenuItem, JCheckBoxMenuItem, and JRadioButtonMenuItem components,
 - as well as other JMenu components.
 - A submenu is a JMenu component that is inside another JMenu component.
 - JMenuItem – Used to create a regular menu item.
 - A JMenuItem component generates an action event when selected.

Menu Classes

- `JCheckBoxMenuItem` – Used to create a check box menu item.
 - The class's `isSelected` method returns true if the item is selected, or false otherwise.
 - A `JCheckBoxMenuItem` component generates an action event when selected.
- `JRadioButtonMenuItem` – Used to create a radio button menu item.
 - `JRadioButtonMenuItem` components can be grouped together in a `ButtonGroup` object so that only one of them can be selected at a time.
 - The class's `isSelected` method returns true if the item is selected, or false otherwise.
 - A `JRadioButtonMenuItem` component generates an action event when selected.

Menu Example

- Menu Example: [MenuWindow.java](#)

Text Areas

- The `JTextField` class is used to create text fields.
- A text field is a component that allows the user to enter a single line of text.
- A text area is like a text field that can accept multiple lines of input.
- You use the `JTextArea` class to create a text area.
- The general format of two of the class's constructors:
`JTextArea(int rows, int columns)`
`JTextArea(String text, int rows, int columns)`

Text Areas

- The `JTextArea` class provides the `getText` and `setText` methods for getting and setting the text.

```
String userText = textInput.getText();  
textInput.setText("Modified: " + userText);
```

- `JTextArea` components do not automatically display scroll bars.
- You must add a text area to a scroll pane.

```
JTextArea textInput = JTextArea(20, 40);  
JScrollPane scrollPane = new  
    JScrollPane(textInput);
```

Text Areas

- The `JScrollPane` object displays both vertical and horizontal scroll bars on a text area.
- By default, the scroll bars are not displayed until they are needed.
- This behavior can be altered:

```
scrollPane.setHorizontalScrollBarPolicy(  
    JScrollPane.HORIZONTAL_SCROLLBAR_NEVER) ;  
scrollPane.setVerticalScrollBarPolicy(  
    JScrollPane.VERTICAL_SCROLLBAR_ALWAYS) ;
```

Text Areas

- You can pass one of the following constants as an argument:
 - `setHorizontalScrollBarPolicy`
 - `JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED.`
 - `JScrollPane.HORIZONTAL_SCROLLBAR_NEVER`
 - `JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS`
 - `setVericalScrollBarPolicy`
 - `JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED`
 - `JScrollPane.VERTICAL_SCROLLBAR_NEVER`
 - `JScrollPane.VERTICAL_SCROLLBAR_ALWAYS`

Text Areas

- By default, JTextArea components do not perform *line wrapping*.
- To enable line wrapping:
`textInput.setLineWrap(true) ;`
- There are two different styles of line wrapping:
 - word wrapping – the line breaks always occur between words.
`textInput.setWrapStyleWord(true) ;`
 - character wrapping – lines are broken between characters (default mode).

Fonts

- Components display according to their font characteristics:
 - font – the name of the typeface
 - style – can be plain, bold, and/or italic
 - size – size of the text in points.
- A component's `setFont` method will change the appearance of the text in the component:
`setFont (Font appearance)`
- A `Font` constructor takes three parameters:
`Font(String fontName, int style, int size)`

Fonts

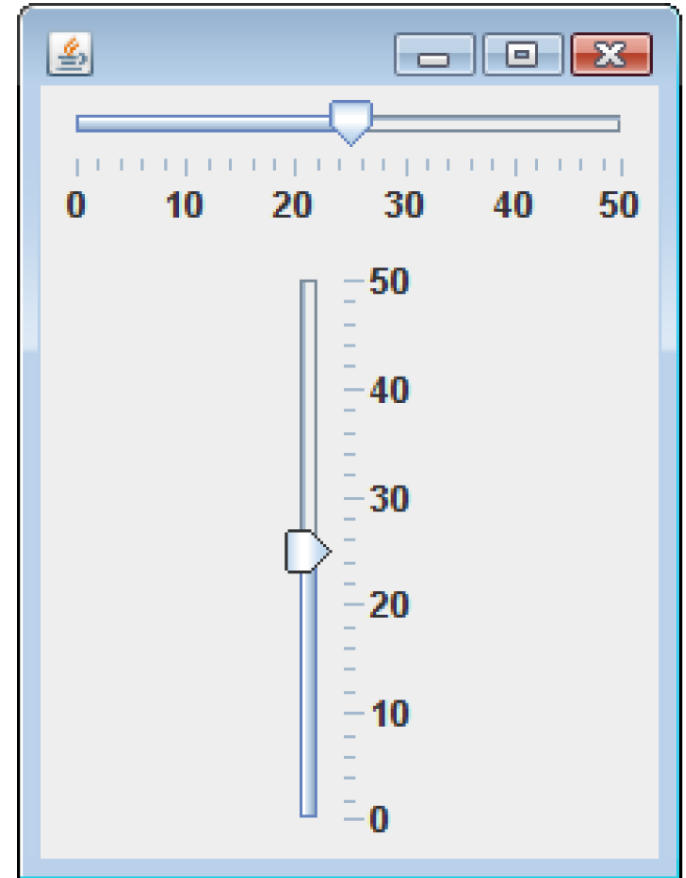
- Java guarantees that you will have the fonts:
 - Dialog, DialogInput, Monospaced, SansSerif, and Serif.
- There are three font styles:
 - Font.PLAIN, Font.BOLD, and Font.ITALIC.
- Example:

```
label.setFont(new Font(  
                    "Serif", Font.BOLD, 24));
```
- Font styles can be combined adding them.

```
label.setFont(new Font(  
                    "Serif", Font.BOLD + Font.ITALIC, 24));
```

Sliders

- A slider is a component that allows the user to graphically adjust a number within a range.
- Sliders are created from the `JSlider` class.
- They display an image of a “slider knob” that can be dragged along a track.



Sliders

- A slider is designed to represent a range of numeric values.
- As the user moves the knob along the track, the numeric value is adjusted accordingly.
- Between the minimum and maximum values, major tick marks are displayed with a label indicating the value at that tick mark.
- Between the major tick marks are minor tick marks.

Sliders

- The `JSlider` constructor has the general format:
`JSlider(int orientation, int minValue,
int maxValue, int initialValue)`
- For orientation, one of these constants should be used:
 - `JSlider.HORIZONTAL`
 - `JSlider.VERTICAL`

Sliders

- Example:

```
JSlider slider1 = new JSlider(JSlider.HORIZONTAL,  
    0, 50, 25);  
JSlider slider2 = new JSlider(JSlider.VERTICAL, 0,  
    50, 25);
```

- Set the major and minor tick mark spacing with:

- `setMajorTickSpacing`
- `setMinorTickSpacing`

- Example:

```
slider1.setMajorTickSpacing(10);  
slider1.setMinorTickSpacing(2);
```

Sliders

- Display tick marks by calling:
 - `setPaintTickMarks`
`slider1.setPaintTickMarks(true) ;`
- Display numeric labels on the slider by calling:
 - `setPaintLabels`
`slider1.setPaintLabels(true) ;`
- When the knob's position is moved, the slider component generates a *change event*.
- To handle the change event, write a *change listener* class.

Sliders

- A change listener class must meet the following requirements:
 - It must implement the `ChangeListener` interface.
 - It must have a method named `stateChanged`.
 - This method must take an argument of the `ChangeEvent` type.
- To retrieve the current value stored in a `JSlider`, use the `getValue` method.
`currentValue = slider1.getValue();`
- Example: [TempConverter.java](#)

Look and Feel

- The appearance of a particular system's GUI is known as its *look and feel*.
- Java allows you to select the look and feel of a GUI application.
- On most systems, Java's default look and feel is called *Metal*.
- There are also Motif and Windows look and feel classes for Java.
 - Motif is similar to a UNIX look and feel
 - Windows is the look and feel of the Windows operating system.

Look and Feel

- To change an application's look and feel, call the `UIManager` class's static `setLookAndFeel` method.
- Java has a class for each look and feel.
- The `setLookAndFeel` method takes the fully qualified class name for the desired look and feel as its argument.
- The class name must be passed as a string.

Look and Feel

- Metal look and feel:

```
"javax.swing.plaf.metal.MetalLookAndFeel"
```

- Motif look and feel:

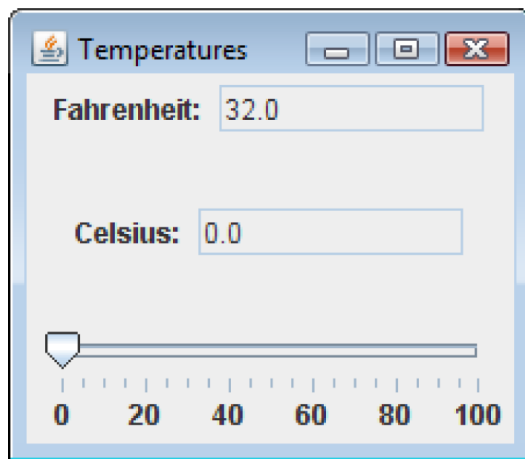
```
"com.sun.java.swing.plaf.motif.MotifLookAndFeel"
```

- Windows look and feel:

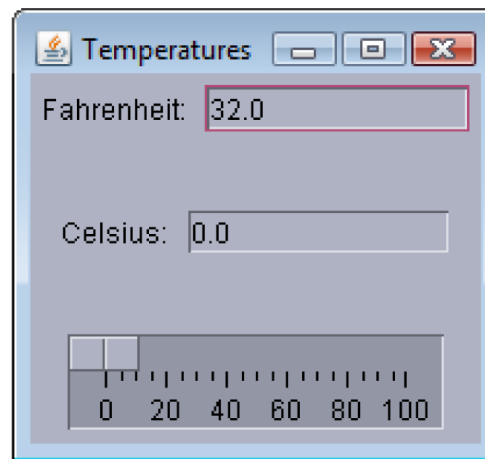
```
"com.sun.java.swing.plaf.windows.WindowsLookAndFeel"
```

Look and Feel

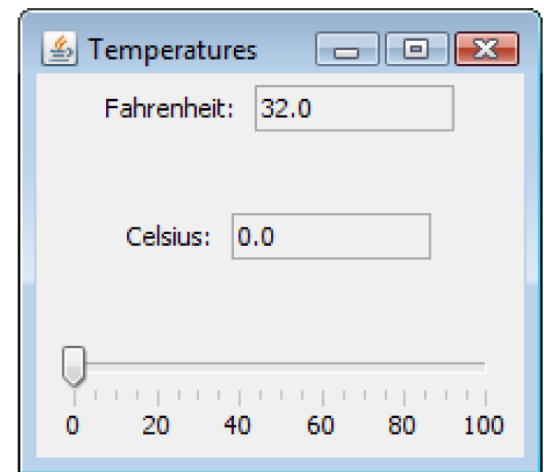
Metal look and feel



Motif look and feel



Windows look and feel



Look and Feel

- Any components that have already been created need to be updated.
`SwingUtilities.updateComponentTreeUI (...);`
- This method takes a reference to the component that you want to update as an argument.
- The `UIManager.setLookAndFeel` method throws a number of exceptions:
 - `ClassNotFoundException`
 - `InstantiationException`
 - `IllegalAccessException`
 - `UnsupportedLookAndFeelException`

Look and Feel

- Example (Motif):

```
try
{
    UIManager.setLookAndFeel (
        "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
    SwingUtilities.updateComponentTreeUI(this);
}
catch (Exception e)
{
    JOptionPane.showMessageDialog(null,
        "Error setting the look and feel.");
    System.exit(0);
}
```

Look and Feel

- Example (Windows):

```
try
{
    UIManager.setLookAndFeel(
        "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
    SwingUtilities.updateComponentTreeUI(this);
}
catch (Exception e)
{
    JOptionPane.showMessageDialog(null,
        "Error setting the look and feel.");
    System.exit(0);
}
```

CHAPTER 14

Applets and More

starting out with >>>

JAVA™

From Control Structures through Objects

6TH EDITION



TONY GADDIS

Chapter Topics

Chapter 14 discusses the following main topics:

- Introduction to Applets
- A Brief Introduction to HTML
- Creating Applets with Swing
- Using AWT for Portability
- Drawing Shapes
- Handling Mouse and Key Events
- `Timer` Objects
- Playing Audio

Introduction to Applets

- There are two types of programs you can create with Java:
 - applications
 - applets.
- An *application* is a stand-alone program that runs on your computer.
- *Applets* are Java programs that are usually part of a Web site.
- If a user opens the Web site with a Java-enabled browser, the applet is executed inside the browser window.

Introduction to Applets

- It appears to the user that the applet is part of the Web site.
- Applets are stored on a Web server along with the site's Web pages.
- Applets associated with a viewed web page are transmitted to the user's system.
- Once the applets are transmitted, the user's system executes them.
- Applets can be used to extend the capabilities of a Web page.

Introduction to Applets

- Web pages are normally written in Hypertext Markup Language (HTML).
- HTML is static content; whereas, applets are dynamic.
- An applet does not have to be on a web server in order to be executed.
 - They can be stored on the local computer.

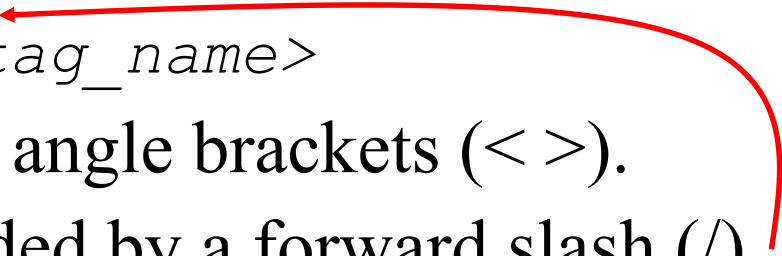
Applet Limitations

- Applets run on the user's system, not the server.
- For security purposes, applets can not:
 - access the local computer file system,
 - run any other program on the user's system.
 - execute operating system procedures.
 - retrieve information about the user or their system.
 - make network connections with any system except the server from which the applet was transmitted.
 - run anonymously.
 - If an applet displays a window, it will automatically have a message such as “Warning: Applet Window” displayed in it.

Introduction to HTML

- Hypertext Markup Language (HTML) is the language that Web pages are written in.
 - *Hypertext* can contain a link to other content on the web page, or another web page.
 - A *Markup Language* allows you to “mark up” a text file by inserting special instructions.
 - These instructions tell the browser how to format the text and create any hypertext links.
- To make a web page, create a text file:
 - that contains HTML instructions (known as *tags*),
 - the text that should be displayed on the Web page, and
 - typically has a .html file extension.

Introduction to HTML

- This document is called an *HTML document*.
 - The tags instruct the browser:
 - how to format the text,
 - where to place images,
 - what to do when the user clicks on a link, etc.
 - Most HTML tags have an opening tag and a closing tag.
 - `<tag_name>Text</tag_name>`
 - The tags are enclosed in angle brackets (< >).
 - The closing tag is preceded by a forward slash (/).
- 

Document Structure Tags

- The `<html></html>` tag marks the beginning and ending of an HTML document.
- The tag `<head></head>` marks the *document head*, a section containing information about the document.
- The document head contains the `<title></title>` tag, which contains the title of the document.
- Example: [BasicWebPage1.html](#)

Document Structure Tags

- After the document head comes the `<body></body>` tag.
- The *document body* contains all of the tags and text that produce output in the browser.
- Example: [BasicWebPage2.html](#)

Text Formatting Tags

- There are many HTML tags that you can use to change the appearance of text.
- For example, there are six different header tags.
 - `<h1></h1>` through `<h6></h6>`
- A level one header appears in boldface, and is much larger than regular text.
- A level two header also appears in boldface, but is smaller than a level one header.
- This pattern continues with the other header tags.

Text Formatting Tags

- Many tags allow an *align* attribute to be used to modify where the text shows on the web page:
 - `<h1 align="center">Text</h1>`
 - `<h1 align="left">Text</h1>`
 - `<h1 align="right">Text</h1>`
- An old way of centering text is to use the `<center></center>` tag to center a line of text.
- You can display text:
 - in boldface `<b></b>`, and italics `<i></i>` .
- Example: [BasicWebPage3.html](#)

Breaks in Text

- The `<br />` tag causes a line break to appear at the point in the text where it is inserted.
- Browsers usually ignore the newline characters that are created when you press the Enter key.
- The `<p />` tag causes a paragraph break.
 - A paragraph break typically inserts more space into the text than a line break.

Breaks in Text

- The `<hr />` tag causes a horizontal rule to appear at the point in the text where it is inserted.
- A horizontal rule is a thin, horizontal line that is drawn across the web page.
- Example: [BasicWebPage4.html](#)

HTML Links

- A link is some element in a Web page that can be clicked on by the user.
- The tag that is used to insert a link has the following general format:
 - `<a href="Address">Text</a>`
- The *Text* that appears between the opening and closing tags is the text that will be displayed in the web page.
- The web resource that is located at *Address* will be displayed in the browser.

HTML Links

- This address is a *uniform resource locator (URL)*.
- The address is enclosed in quotation marks.
- Example:
 - `<a href="http://www.gaddisbooks.com">Click here to go to the textbook's web site.</a>`
- Example: [LinkDemo.html](#)

Creating Applets With Swing

- Applets are very similar to the GUI applications.
- Instead of displaying its own window, an applet appears in the browser's window.
- The differences between GUI application code and applet code are:
 - A GUI application class is derived from `JFrame`.
 - An applet class is derived from `JApplet`.
 - The `JApplet` class is part of the `javax.swing` package.
 - A GUI application class has a constructor that creates other components and sets up the GUI.
 - An applet class does not normally have a constructor.
 - Instead, it has a method named `init` that performs the same operations as a constructor.

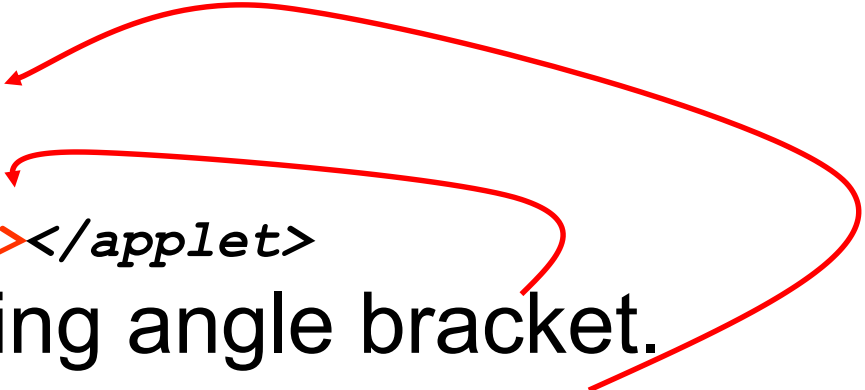
Creating Applets With Swing

- The differences are (continued):
 - The following methods are not called in an applet:
 - `super`
 - `setSize`
 - `setDefaultCloseOperation`
 - `pack`
 - `setVisible`
 - No `main` method is needed to create an `Applet` object.
 - The browser creates an instance of the class automatically.
- Example:
 - [SimpleApplet.java](#)
 - [SimpleApplet.html](#)

Running an Applet

- The process of running an applet is different from that of running an application.
- To run an applet, create an HTML document with an `APPLET` tag, which has the following general format:

```
<applet  
  code="Filename.class"  
  width="width_value"  
  height="height_value"></applet>
```



- Don't forget the closing angle bracket.
- Attributes should be enclosed in quotes.

Running an Applet

- *Filename.class* is the compiled bytecode of the applet, not the `.java` file.
- You can optionally specify a path along with the file name.
- If you specify only the file name, it is assumed that the file is in the same directory as the HTML
- The browser:
 - loads specified byte code, and
 - executes it in an area that is the size specified by the *width_value* and *height_value*.

Using appletviewer

- The `appletviewer` program loads and executes an applet without the need for a Web browser.
- When running the program, specify the name of an HTML document as a command line argument.

`appletviewer SimpleApplet.html`

- This command executes any applet referenced by an `APPLET` tag in the file *SimpleApplet.html*.
- If the document has more than one `APPLET` tag, it will execute each applet in a separate window.

Applet Event Handling

- Events in applets are handled with event listeners exactly as they are in GUI applications.
- Example:
 - [TempConverter.java](#)
 - [TempConverter.html](#)

Using AWT for Portability

- AWT is the original library that has been part of Java since its earliest version.
- Swing is an improved library that was introduced with Java 2.
- Some browsers do not directly support the Swing classes in applets.
- These browsers require a *plug-in* to run swing applets.
- This plug-in is automatically installed on a computer when the Java SDK is installed.

Using AWT for Portability

- Other people running applets might not have the required plug-in.
- The AWT classes can be used instead of the Swing classes for the components in the applet.
- The AWT component classes:
 - there is a corresponding AWT class for each of the Swing classes covered so far.
 - The names of the AWT classes names do not start with the letter J.
- Example:
 - [AWTTempConverter.java](#), [TempConverter.html](#)

Drawing Shapes

- Components have an associated `Graphics` object that may be used to draw lines and shapes.
- Java allows drawing of lines and graphical shapes such as rectangles, ovals, and arcs.
- Frame or panels can become a canvas for your drawings.

XY Coordinates

- The location of each pixel in a component is identified with an X coordinate and a Y coordinate.
- The coordinates are usually written in the form (X, Y) .
- Unlike Cartesian coordinates, the upper-left corner of a drawing area $(0, 0)$.
- The X coordinates increase from left to right, and the Y coordinates increase from top to bottom.
- When drawing a line or shape on a component, you must indicate its position using X and Y coordinates.

Graphics Objects

- Each component has an internal object that is derived from the `Graphics` class, which is part of the `java.awt` package.
- This object has numerous methods for drawing graphical shapes on the surface of the component.

Graphics Objects

- Some of the methods of the `Graphics` class:
 - `setColor(Color c)` – Sets the drawing color for this object.
 - `getColor()` – Returns the current drawing color for this object.
 - `drawLine(int x1, int y1, int x2, int y2)` – Draws a line on the component
 - `drawRect(int x, int y, int width, int height)` – Draws the outline of a rectangle on the component.
 - `fillOval(int x, int y, int width, int height)` – Draws a filled oval.
 - `drawString(String str, int x, int y)` – Draws the string passed into *str* using the current font.

Graphics Objects

- In order to call these methods, you must get a reference to a component's `Graphics` object.
- One way to do this is to override the `paint` method.
- You can override the `paint` method in any class that is derived from:
 - `JApplet`
 - `JFrame`
 - Any AWT class
- The `paint` method is responsible for displaying, or “painting,” a component on the screen.

Graphics Objects

- The `paint` method is automatically called
 - when the component is first displayed and
 - any time the component needs to be redisplayed.
- The header for the `paint` method is:
`public void paint(Graphics g)`
- The method's argument is a `Graphics` object, which is automatically passed by the calling component.
- Overriding the `paint` method, allows drawing of graphics on the `Graphics` object argument.

Example: [LineDemo.java](#), [LineDemo.html](#)

Graphics Objects

- The `Graphics` object argument is responsible for drawing the entire applet window.
- It is advisable to call the base class `paint` method passing the `Graphics` object, `g`, as an argument:

```
super.paint(g) ;  
g.setColor(Color.red) ;  
g.drawLine(20, 20, 280, 280) ;
```

- This is a red diagonal line drawn from the top-left area of the applet window to the bottom-right area.

Rectangles

- Rectangles can be drawn or filled.

```
g.drawRect(10, 10, 50, 50);  
g.fillRect(10, 10, 50, 50);
```

- The `fillRect` and `drawRect` take four integers as parameters:

```
drawRect(int x, int y, int width, int height)
```

- Example:
 - [RectangleDemo.java](#)
 - [RectangleDemo.html](#)

Ovals and Bounding Rectangles

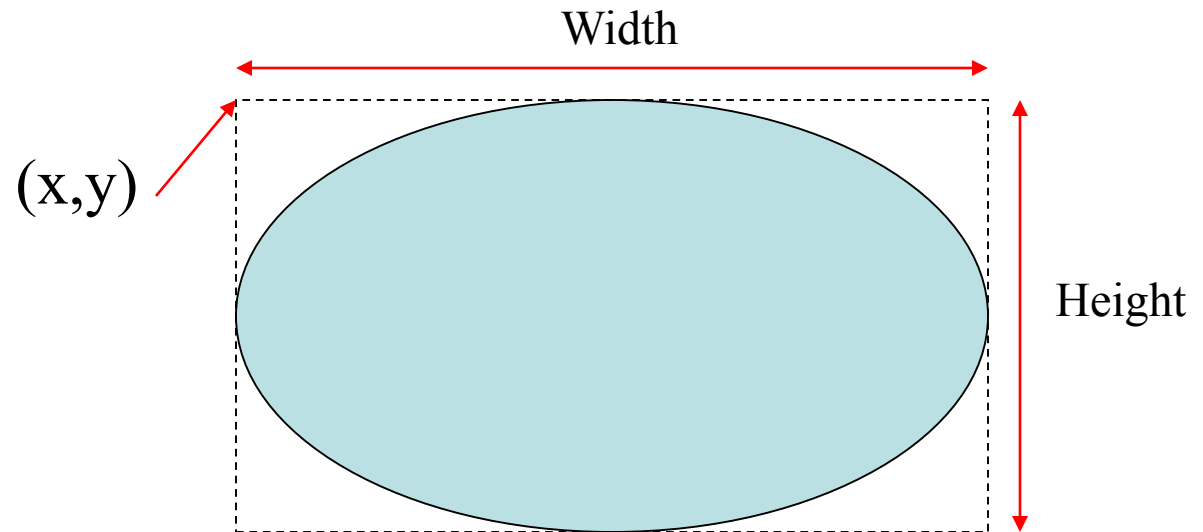
- Ovals are created by drawing the oval inside of a “bounding rectangle”.
- This rectangle is invisible to the viewer of the `Graphics` object.

```
g.fillOval(x, y, width, height);
```

Example:

[OvalDemo.java](#)

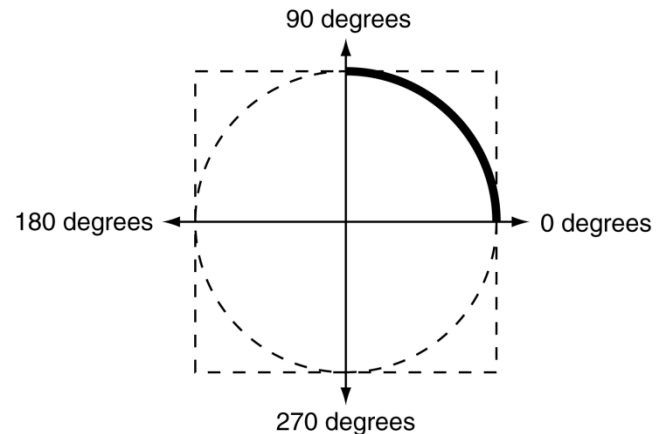
[OvalDemo.html](#)



Arcs

- Arcs are drawn from the 90 degree position counterclockwise and can be filled or unfilled

```
g.drawArc(0, 20, 120, 120, 0, 90);  
g.fillArc(0, 20, 120, 120, 0, 90);
```
- The `fillArc` and `drawArc` take six integers as parameters:
`drawArc(int x, int y, int width, int height, int start, int end)`
- Example:
 - [ArcDemo.java](#)
 - [ArcDemo.html](#)

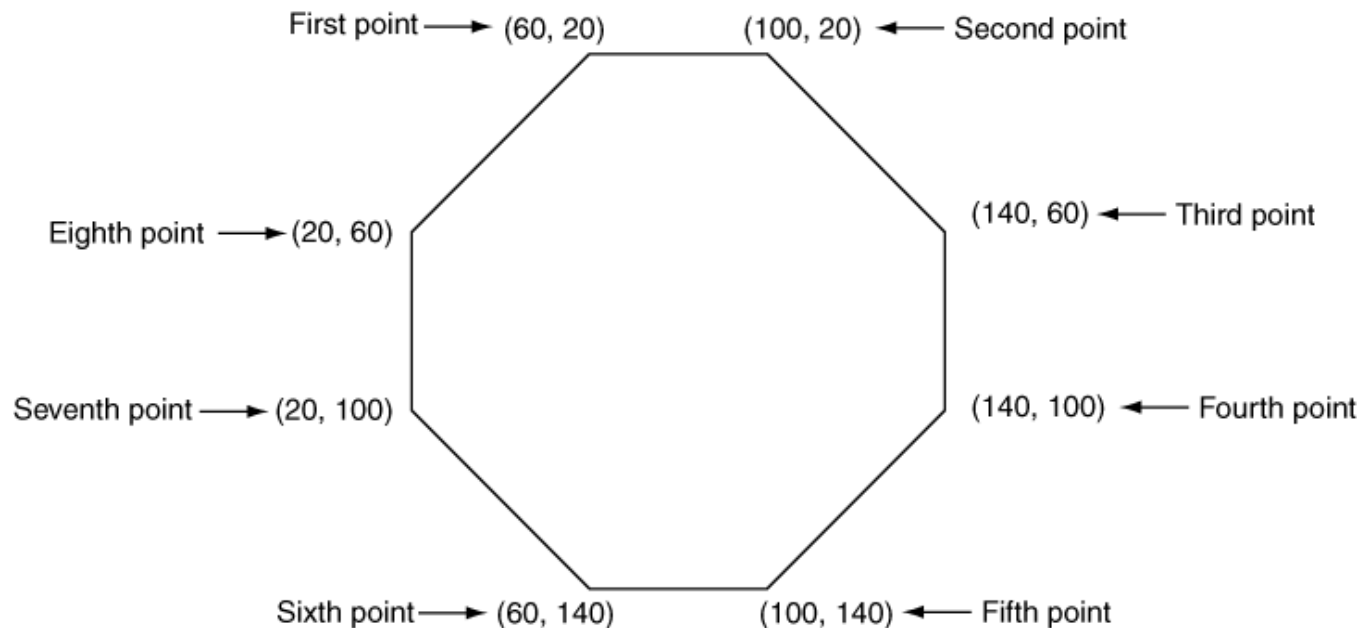


Polygons

- Polygons are drawn using arrays of integers representing x, y coordinates

```
int [] xCoords = { 60, 100, 140, 140, 100, 60, 20, 20 } ;
```

```
int [] yCoords = { 20, 20, 60, 100, 140, 140, 100, 60 } ;
```



Polygons

- The `fillPolygon` and `drawPolygon` use the arrays as parameters:
- Example:
 - [PolygonDemo.java](#)
 - [PolygonDemo.html](#)

The `repaint` Method

- We do not call a component's `paint` method.
- It is automatically called when the component must be redisplayed.
- We can force the application or applet to call the `paint` method.

`repaint()` ;

- The `repaint` method clears the surface of the component and then calls the `paint` method.

Drawing on Panels

- To draw on a panel, get a reference to the panel's `Graphics` object and use that object's methods.
- The resulting graphics are drawn only on the panel.
- Getting a reference to a `JPanel` component's `Graphics` object is similar to previous examples.
- Instead of overriding the `JPanel` object's `paint` method, override its `paintComponent` method.
- This is true for all Swing components except `JApplet` and `JFrame`.

Drawing on Panels

- The `paintComponent` method serves the same purpose as the `paint` method.
- When it is called, the component's `Graphics` object is passed as an argument.

```
public void paintComponent(Graphics g)
```

- When overriding this method, first call the base class's `paintComponent` method.

```
super.paintComponent(g) ;
```

Drawing on Panels

- After this you can call any of the `Graphics` object's methods to draw on the component.
- Example:
 - [GraphicsWindow.java](#),
 - [DrawingPanel.java](#),
 - [GraphicsWindow.html](#)

Handling Mouse Events

- The mouse generates two types of events:
 - mouse events and mouse motion events.
- Any component derived from the `Component` class can handle events generated by the mouse.
- To handle mouse events you create:
 - a *mouse listener* class and/or
 - a *mouse motion listener* class.

Handling Mouse Events

- A mouse listener class can respond to any of the follow events:
 - The mouse button is pressed.
 - The mouse button is released.
 - The mouse button is clicked on (pressed, then released without moving the mouse).
 - The mouse cursor enters a component's screen space.
 - The mouse cursor exits a component's screen space.
- A mouse listener class must implement the `MouseListener` interface.

Mouse Listener Methods

- `public void mousePressed(MouseEvent e)`
 - called if the mouse button is pressed over the component.
- `public void mouseClicked(MouseEvent e)`
 - called if the mouse is pressed and released over the component without moving the mouse.
- `public void mouseReleased(MouseEvent e)`
 - called when the mouse button is released.
- `public void mouseEntered(MouseEvent e)`
 - called when the mouse cursor enters the screen area of the component.
- `public void mouseExited(MouseEvent e)`
 - This method is called when the mouse cursor leaves the screen area of the component.

Mouse Events

- The `MouseEvent` object contains data about the mouse event.
- `getX` and `getY` are two common methods of the `MouseEvent` class.
- They return the *X* and *Y* coordinates of the mouse cursor when the event occurs.
- Once a mouse listener class is created, it can be registered with a component using the `addMouseListener` method

Mouse Motion Events

- The appropriate methods in the mouse listener class are automatically called when their corresponding mouse events occur.
- A mouse motion listener class can respond to the following events:
 - The mouse is dragged
 - The mouse moved.
- A mouse motion listener class must implement the `MouseMotionListener` interface and its methods.

Mouse Motion Listener Methods

- `public void mouseDragged(MouseEvent e)`
 - called when a dragging operation begins over the component.
 - The `mousePressed` method is always called just before this method.
- `public void mouseMoved(MouseEvent e)`
 - called when the mouse cursor is over the component and it is moved.
- Example:
 - [MouseEvents.java](#)
 - [MouseEvents.html](#)

Using Adapter Classes

- The mouse listener class must implement *all* of the methods required by the interfaces they implement.
- If any of the methods are omitted, a compiler error results.
- The `MouseAdapter` and `MouseMotionAdapter` classes provide empty implementations of the methods.
- They can serve as base classes for mouse listener and mouse motion listener classes.
- Examples: [DrawBoxes.java](#), [DrawBoxes.html](#), [DrawBoxes2.java](#), [DrawBoxes2.html](#)

Timer Objects

- `Timer` objects automatically generate action events at regular time intervals.
- This is useful when you want a program to:
 - perform an operation at certain times or
 - after an amount of time has passed.
- `Timer` objects are created from the `Timer` class.
- The general format of the `Timer` class's constructor:

`Timer(int delay, ActionListener listener)`

Timer Objects

- The *delay* parameter is the amount of time between action events in milliseconds.
- The the *listener* parameter is a reference to an action listener to be registered with the `Timer` object.
 - Passing `null` will cause no action listener to be registered.
 - the `Timer` object's `addActionListener` method can register an action listener after the object's creation.

Timer Object Methods

- `void addActionListener (ActionListener listener)`
 - Registers the object referenced by *listener* as an action listener.
- `int getDelay()`
 - Returns the current time delay in milliseconds.
- `boolean isRunning()`
 - Returns true if the Timer object is running.
- `void setDelay(int delay)`
 - Sets the time delay in milliseconds.
- `void start()`
 - Starts the Timer object.
- `void stop()`
 - Stops the Timer object.

Timer Object Methods

- An application can use a `Timer` object to automatically execute code at regular time intervals.
- Example:
 - [BouncingBall.java](#)
 - [BouncingBall.html](#)

Playing Audio

- Java programs can play audio that is stored in a variety sound file formats.
 - .aif or .aiff (Macintosh Audio File)
 - .au (Sun Audio File)
 - .mid or .rmi (MIDI File)
 - .wav (Windows Wave File)
- One way to play an audio file is to use the `Applet` class's `play` method.
- One version of this method is:
 - `void play(URL baseLocation, String fileName)`

Playing Audio

- The argument passed to *baseLocation* is a URL object that specifies the location of the file.
- The argument passed to *fileName* is and name of the file.
- The sound that is recorded in the file is played one time.
- The `getDocumentBase` or `getCodeBase` methods can get a URL object for the first argument.

Playing Audio

- The `getDocumentBase` method returns a URL object containing the location of the HTML file that invoked the applet.
`play(getDocumentBase(), "mysound.wav");`
- The `getCodeBase` method returns a URL object containing the location of the applet's .class file.
`play(getCodeBase(), "mysound.wav");`
- If the sound file specified by the arguments to the play method cannot be found, no sound will be played.

Using an AudioClip Object

- The Applet class's play method:
 - loads a sound file,
 - plays it one time, and
 - releases it for garbage collection.
- If you need to load a sound file to be played multiple times, use an AudioClip object.
- An AudioClip object is an object that implements the AudioClip interface.

Using an AudioClip Object

- The AudioClip interface specifies the following three methods:
 - play – plays a sound one time.
 - loop – repeatedly plays a sound.
 - stop – causes a sound to stop playing.
- The Applet class's getAudioClip method can be used to create an AudioClip object:

```
AudioClip getAudioClip(URL baseLocation,  
                        String fileName)
```

- The method returns an AudioClip object that can be used to play the sound file.
- Example: [AudioDemo2.java](#), [AudioDemo2.html](#)

Playing Audio in an Application

- Playing audio in from a JFrame is slightly different than playing audio from an applet.

```
// Create a file object for the step.wav file.  
File file = new File("step.wav");
```

```
// Get a URI object for the audio file.  
URI uri = file.toURI();
```

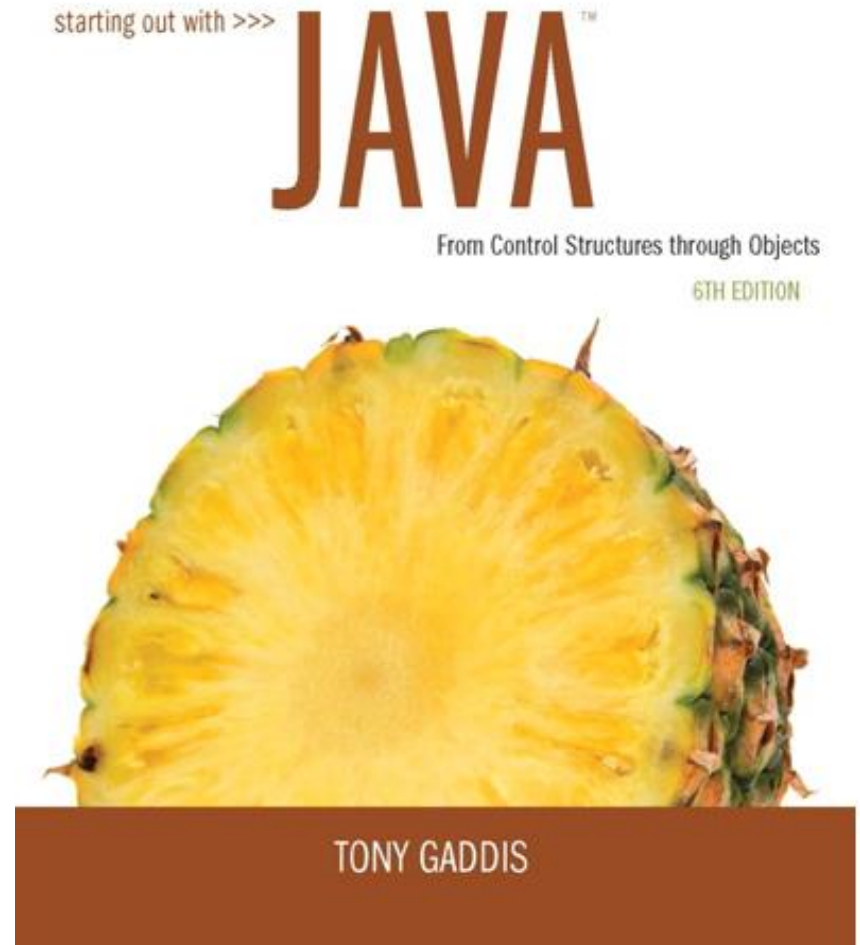
```
// Get a URL for the audio file.  
URL url = uri.toURL();
```

```
// Get an AudioClip object for the sound  
// file using the Applet class's static  
// newAudioClip method.  
sound = Applet.newAudioClip(url);
```

Example:
[AudioFrame.java](#)

CHAPTER 15

Creating GUI Applications with JavaFX and Scene Builder



Topics

- Introduction
- Scene Graphs
- Using Scene Builder to Create JavaFX Applications
- Writing the Application Code
- RadioButtons and CheckBoxes
- Displaying Images

Introduction

- *JavaFX* is a Java library for developing rich applications that employ graphics.
- You can use it to create:
 - GUI applications, as well as applications that display 2D and 3D graphics
 - standalone graphics applications that run on your local computer
 - applications that run from a remote server
 - applications that are embedded in a Web page

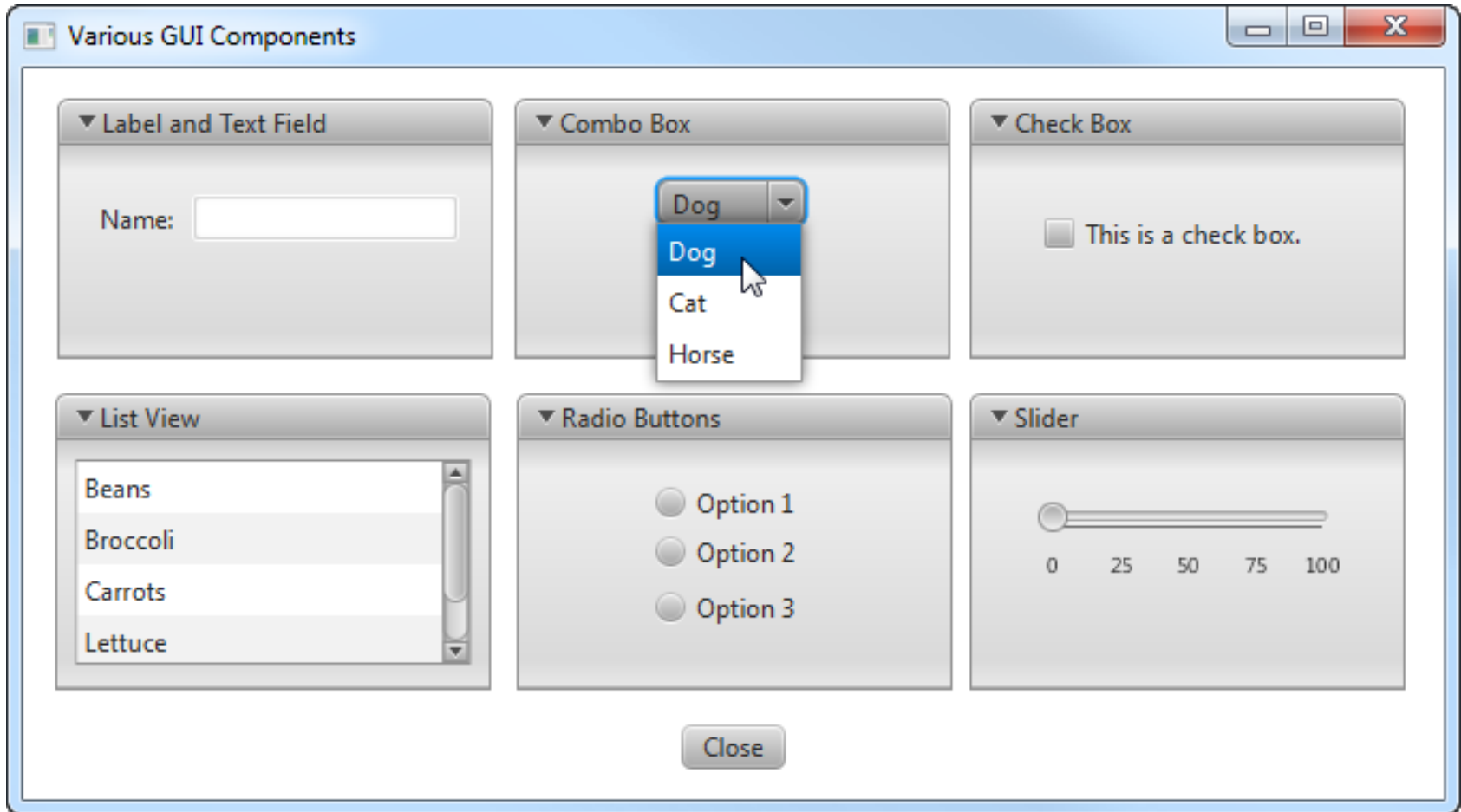
Introduction

- A GUI (pronounced “gooey”) is a graphical window or a system of graphical windows presented by an application for interaction with the user.
- In addition to accepting input from the keyboard, GUIs typically accept input from a mouse, or a touch screen.

Introduction

- A window in a GUI commonly consists of several *components* that present data to the user and/or allow interaction with the application.
- Some of the common GUI components are buttons, labels, text fields, check boxes, and radio buttons.

Various GUI Components



Some GUI Components

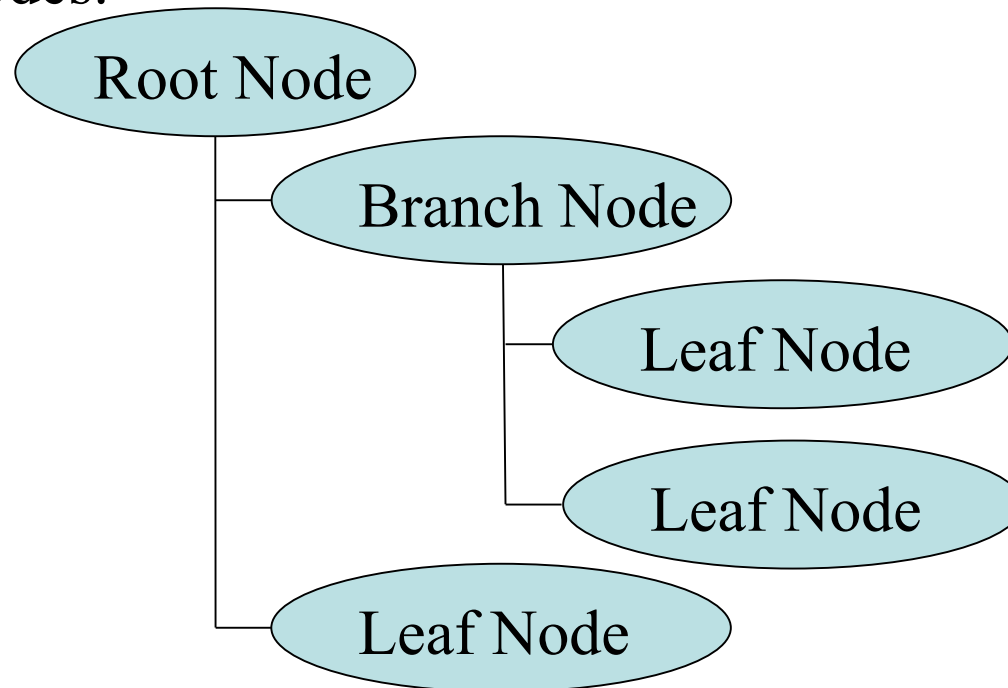
Component	Description
AnchorPane	A container for other components. The components that are contained inside an AnchorPane can be “anchored” at a certain distance from one or more of the AnchorPane’s edges.
Button	A button that can cause an action to occur when it is clicked.
CheckBox	A component that has a box that may be checked or unchecked.
ComboBox	A component that displays a drop-down list of items from which the user may select. A ComboBox also provides a TextField in which the user may type input. It is called a ComboBox because it can behave as the combination of a list and a TextField.
Label	An area that can display text.
List	A list from which the user may select an item.
RadioButton	A component that can be either selected or deselected. RadioButtons usually appear in groups and allow the user to select one of several options.
Slider	A component that allows the user to select a value by moving a slider along a track.
TextField	An area in which the user may type a single line of input from the keyboard.
TitledPane	A container for other components. A TitledPane has a title bar at its top, in which a title can be displayed. A TitledPane may also be closed or opened.

Event-Driven Programming

- Programs that operate in a GUI environment must be *event-driven*.
- An *event* is an action that takes place within a program, such as the clicking of a button.
- Part of writing a GUI application is creating event listeners.
- An *event listener* is a method that automatically executes when a specific event occurs.

Scene Graphs

- In JavaFX, the components that are in a GUI are organized as a scene graph.
- A *scene graph* is a tree-like, hierarchical data structure that contains nodes.



Scene Graphs

- A scene graph can have three types of nodes:
 - Root Node:
 - The scene graph can have only one root node.
 - It is the parent of all the other nodes in the scene graph.
 - It is the first node in the structure.
 - Branch Node:
 - A branch node can have other nodes as children.
 - Leaf Node:
 - A leaf node cannot have children.
- In a nutshell, the root node and branch nodes can have children, but leaf nodes cannot.*

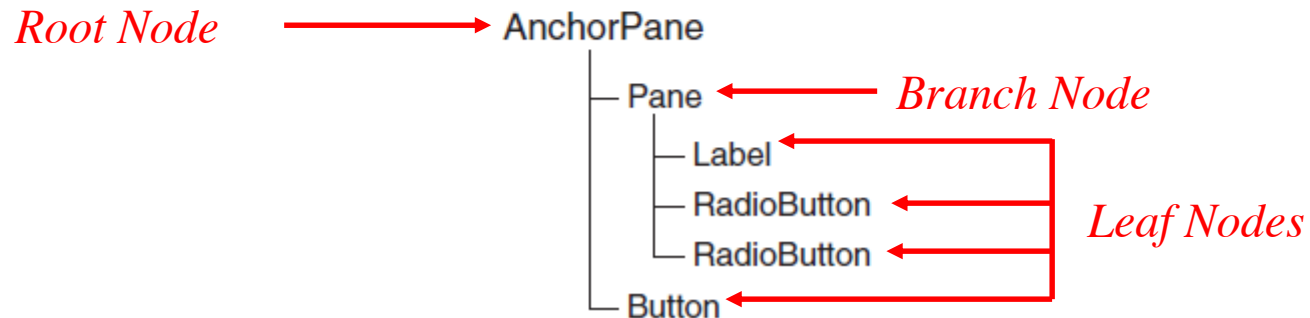
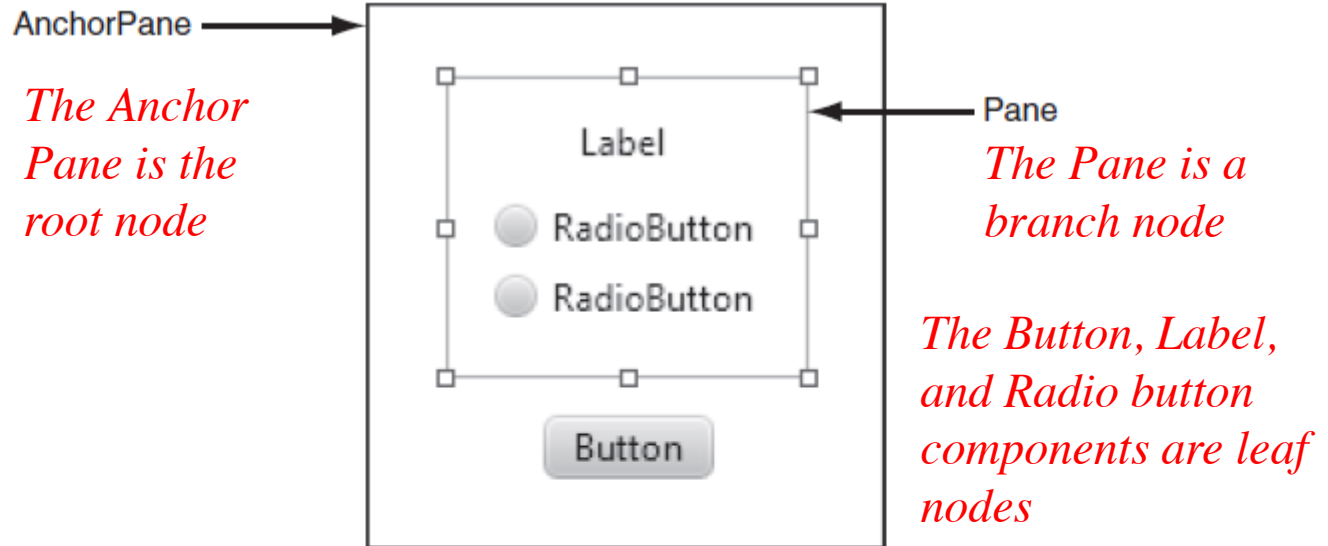
Scene Graphs

- In JavaFX, a node that can have children is a *container*.
- A container is a component that can hold other components inside of it.
- The JavaFX library provides several different types of containers.

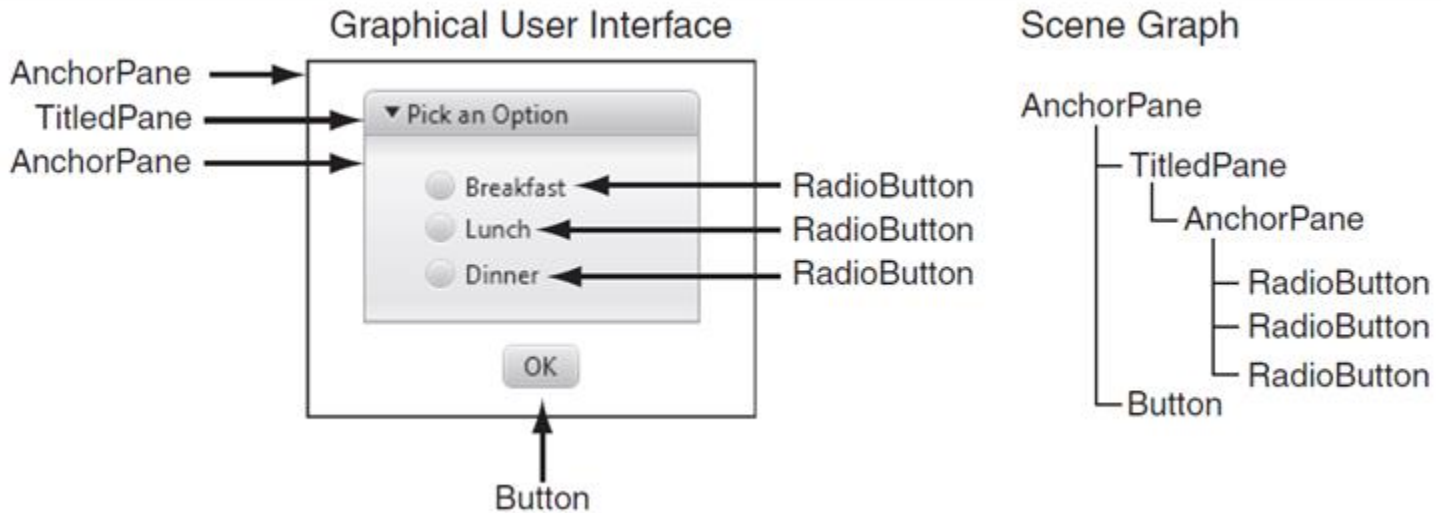
Scene Graphs

- The AnchorPane container is commonly used as a GUI's root node.
- A branch node is a container that is placed inside the root node or inside another branch node.
 - **For example, you might have a Pane (one of the simplest JavaFX containers) inside of an AnchorPane.**
- A leaf node is a component that is not a container.
 - **For example, Button components and Label components are leaf nodes.**

Example GUI and Scene Graph



A More Complex GUI and Scene Graph



- The AnchorPane is the root node
 - The TitledPane is a child of the AnchorPane
 - it is contained inside the AnchorPane
 - Another AnchorPane is a child of the TitledPane
 - it is contained inside the TitledPane
 - The Three RadioButtons are children of the innermost AnchorPane
 - The Button is a child of the root node AnchorPane.

Using Scene Builder to Create JavaFX Applications

- Scene Builder is a free design tool from Oracle for visually creating GUIs.
- It works like this:
 - Use Scene Builder to construct a GUI by dragging and dropping the components that you need onto a blank window.
 - Visually arrange the components on the window and set various component properties to create the appearance that you want for the GUI.
 - Save the GUI to an FXML file.
- *FXML* is a markup language that describes the components in a JavaFX scene graph.
- FXML uses tags to organize data, in a manner similar to the way that HTML uses tags to format text in a Web browser.

Using Scene Builder to Create JavaFX Applications

- Visually creating a GUI with Scene Builder is only part of the process.
- Once you save a GUI's scene graph to an FXML file, the next step is to write Java code that reads the FXML file and displays the GUI on the screen and handles any events that occur while the application is running.

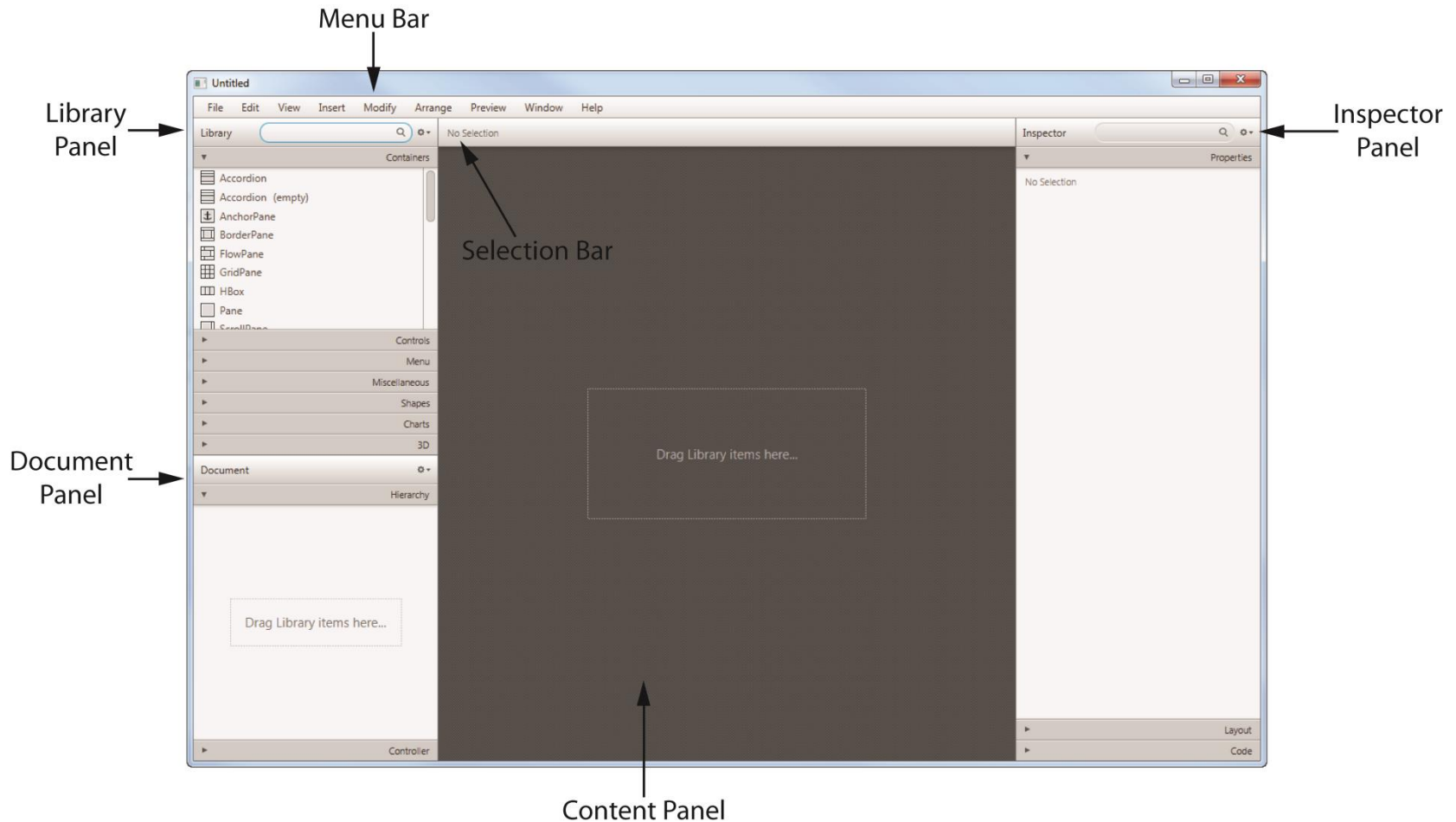
Starting Scene Builder

- You can download Scene Builder from the following location:

www.oracle.com/technetwork/java/javafx/downloads/

- When you install Scene Builder in Windows, a shortcut is automatically created on the desktop.
- You can launch Scene Builder either by double-clicking the shortcut, or by going to *All Programs > JavaFX Scene Builder* and clicking *JavaFX Scene Builder x.x*
 - where *x.x* will be a version number such as 2.0
- If you installed Scene Builder on a Mac, go to the Applications folder and double-click the shortcut for *JavaFX Scene Builder x.x*
 - where *x.x* will be a version number such as 2.0

The Scene Builder Main Window



The Scene Builder Main Window

- Here is a brief summary of each part of the main window:
- Menu Bar
 - Scene Builder's commands are located on the menus that access the menu bar at the top of the main window.

The Scene Builder Main Window

- Library Panel
 - The Library Panel provides a list of JavaFX components that you can use in an application.
 - To place a component in a GUI, you simply drag it from the Library Panel, and drop it into the Content Panel.

The Scene Builder Main Window

- Content Panel
 - The Content Panel is where you visually design an application's GUI.
 - You create components in the GUI by dragging them from the Library Panel and dropping them into the Content Panel

The Scene Builder Main Window

- Document Panel
 - Has two sections:
 - Hierarchy
 - Controller
 - Hierarchy shows the GUI's scene graph as you build it.
 - Controller allows you to connect the GUI to a Java class containing the application's event listeners.

The Scene Builder Main Window

- Selection Bar
 - This area of the screen shows the hierarchical path of the currently selected node in the scene graph.

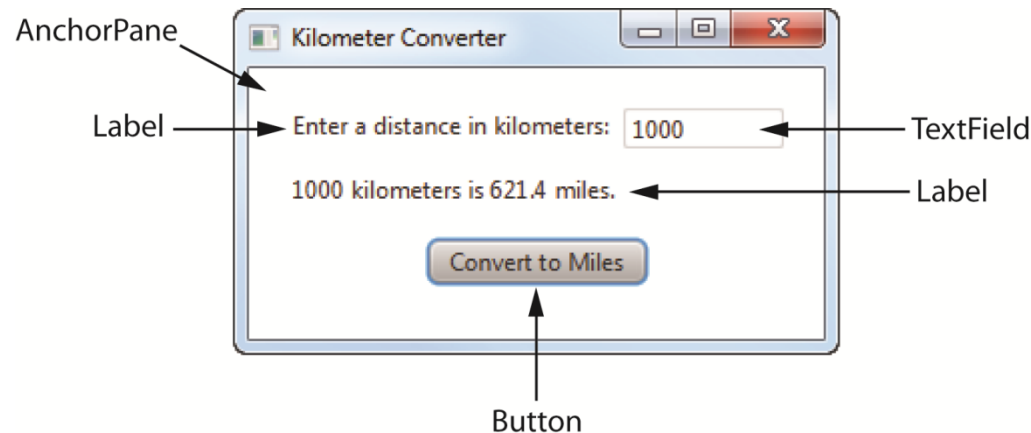
The Scene Builder Main Window

- Inspector Panel
 - The Inspector Panel is divided into three sections:
 - **Properties**
 - **Layout**
 - **Code**

The Scene Builder Main Window

- The Properties section
 - allows you to view and edit the values of the selected component's properties, which are values that determine the component's appearance.
- The Layout section
 - lets you specify values that control the way the component is displayed when the GUI's window is resized.
- The Code section
 - allows you to assign an `fx:id` to a component, which is similar to assigning a variable name to the component.
 - also allows you to designate event handlers to execute when specific events happen to the selected component.

Using Scene Builder to Create the Kilometer Converter GUI



- An AnchorPane, as the root node
- A Label to display the prompt *Enter a distance in kilometers:*
- A TextField in which the user will enter a distance
- A Label to display a message showing the distance converted to miles
- A Button that performs the conversion

Writing the Application Code

- Once you have saved an application's GUI to an FXML file, you can write the Java code that runs the application.
- A simple JavaFX application uses:
 - a main application class
 - a controller class

The Main Application Class

- Once you have created a GUI with Scene Builder, and saved it to an FXML file, you need to write a Java class that performs the following:
 - Loads the FXML file
 - Builds the scene graph in memory
 - Displays the GUI

Example: [KilometerConverter.java](#)

The Controller Class

- The *controller class* is responsible for handling events that occur while the application is running.
- The controller class contains the following items:
 - The necessary import statements
 - Private variables to reference the components that have a `fx:id` in the scene graph
 - An optional initialize method that is automatically called after the FXML file is loaded
 - Event listener methods

Example: [KilometerConverterController.java](#)

Using the Sample Controller Skeleton

- An alternative for manually typing the code for the controller class, Scene Builder can provide a sample "skeleton" for the controller class.
- To see the sample controller skeleton, click the *View* menu, then click *Show Sample Controller Skeleton*
- You can click the Copy button to copy the sample code to the clipboard, and then paste it into an editing window in your IDE.
- The obvious benefit of using the sample skeleton controller is that a lot of the code is written for you.
- The skeleton has all of the necessary import statements, and the class already has private field declarations for all of the components that have an fx:id.
- You just need to change the name of the class, and write the code for the event listener methods.

Summary of Creating a JavaFX Application

- Use Scene Builder to design the GUI. Be sure to give an fx:id to all of the components that you plan to access in your Java code. Save the GUI as an FXML file.
- Write the code for the main application class, which loads the FXML file and launches the application. Save and compile the code in the same location as the FXML file.
- Write the code for the controller class, which contains the event handler methods for the GUI. Save and compile the code in the same location as the FXML file.
- In Scene Builder, register the controller class, then register an event handler method for each component that needs to respond to events. Save the FXML file again.

RadioButtons and CheckBoxes

- *RadioButtons normally appear in groups of two or more and allow the user to select one of several possible options.*
- *CheckBoxes, which may appear alone or in groups, allow the user to make yes/no or on/off selections.*

RadioButtons

- *RadioButtons* are useful when you want the user to select one choice from several possible options.
- A radio button may be selected or deselected.
- Each radio button has a small circle that appears filled-in when the radio button is selected and appears empty when the radio button is deselected.

Creating a RadioButton

- To create a RadioButton, you simply drag it from the Library panel and drop it onto the Content panel.
- (The RadioButton component is found in the Controls section of the Library panel.)
- RadioButtons have a Text property that determines the text they display.
- RadioButtons normally are in a *toggle group*.

Adding RadioButtons to a Toggle Group

- Here are the steps for adding RadioButtons to a toggle group:
 - Create the first RadioButton component in the Content panel.
 - Open the Properties section of the Inspector Panel, and find the Toggle Group property.
 - Enter the name you wish to give the toggle group.
 - Create the next RadioButton.
 - For its Toggle Group property, you should be able to click the down-arrow and select the toggle group that you created for the first RadioButton.
 - Repeat this for each subsequent RadioButton that you want in the same toggle group.

Determining in Code Whether a RadioButton Is Selected

- In the controller class, you can use the `RadioButton`'s `isSelected` method to determine whether the `RadioButton` is selected or not.
- The `isSelected` method returns a boolean value.
 - If the `RadioButton` is selected, the method returns `true` . Otherwise, it returns `false` .

```
if (radio.isSelected())  
{  
    // Code here executes if the  
radio  
    // button is selected.  
}
```

Example: [RadioButtonDemo.java](#), [RadioButtonDemoController.java](#)

Responding to RadioButton Events

- In many situations you want an action to take place at the time the user clicks a RadioButton.
 - you must write an event listener method in the controller class for each RadioButton
 - and then select the appropriate method as the event listener in Scene Builder.

Example: [RadioButtonEvent.java](#), [RadioButtonEventController.java](#)

CheckBoxes

- A *CheckBox* is a small box with text appearing next to it.
- Like `RadioButtons`, `CheckBoxes` may be selected or deselected at run time.
- When a `CheckBox` is selected, a small check mark appears inside the box.
- Although `CheckBoxes` are often displayed in groups, they are not usually grouped in a toggle group like `RadioButtons` are.

Creating a CheckBox

- To create a CheckBox, you simply drag it from the Library panel and drop it onto the Content panel.
- (The CheckBox component is found in the Controls section of the Library panel.)
- CheckBoxes have a Text property that determines the text they display.

Determining in Code Whether a CheckBox Is Selected

- In the controller class, you can use the CheckBox's `isSelected` method to determine whether the CheckBox is selected or not.
- The `isSelected` method returns a boolean value.
 - If the CheckBox is selected, the method returns `true` . Otherwise, it returns `false` .

```
if (checkbox.isSelected())  
{  
    // Code here executes if the  
    // CheckBox is selected.  
}
```

Example: [CheckBoxDemo.java](#), [CheckBoxDemoController.java](#)

Responding to CheckBox Events

- Sometimes you want an action to take place at the time the user clicks a CheckBox.
 - you must write an event listener method in the controller class for the CheckBox
 - and then select the method as the event listener in Scene Builder.

Example: [CheckBoxEvent.java](#), [CheckBoxEventController.java](#)

Displaying Images

- You can use the ImageView component to display images in an application's GUI.
- You simply drag the component from the Library panel (you will find it in the Controls section) and drop it onto the Content Panel
- Once you have created a ImageView component, you use its Image property to specify the image that it will display.

Displaying an Image with Code

- Sometimes you might need to write code that will change the image being displayed in an `ImageView` component, as the application is running.
- In your controller class, you can call the `ImageView` component's `setImage` method to do this.

Displaying an Image with Code

- First, you must create an instance of the `Image` class, which can read the contents of an image file.
- The `Image` class is in the `javafx.scene.image` package.
- The `Image` class constructor accepts a `String` argument that is the name of an image file.
- Here is an example:

```
Image myImage = new Image("Dog.jpg");
```

- Here is an example that uses a path.

```
Image myImage = new Image("C:\\Chapter14" +  
                           "\\Images\\Dog.jpg");
```

Displaying an Image with Code

- Once you have created an `Image` object, you pass a reference to that object to the `ImageView` component's `setImage` method.
- The following is an example:
 - Assume that `myImageView` references an `ImageView` component, and `myImage` references an `Image` object.

```
myImageView.setImage(myImage) ;
```

Example: [ImageViewDemo.java](#), [ImageViewDemoController.java](#)

CHAPTER 16

Recursion

starting out with >>>

JAVA™

From Control Structures through Objects

6TH EDITION



TONY GADDIS

Chapter Topics

Chapter 16 discusses the following main topics:

- Introduction to Recursion
- Solving Problems with Recursion
- Examples of Recursive Methods
- A Recursive Binary Search Method
- The Towers of Hanoi

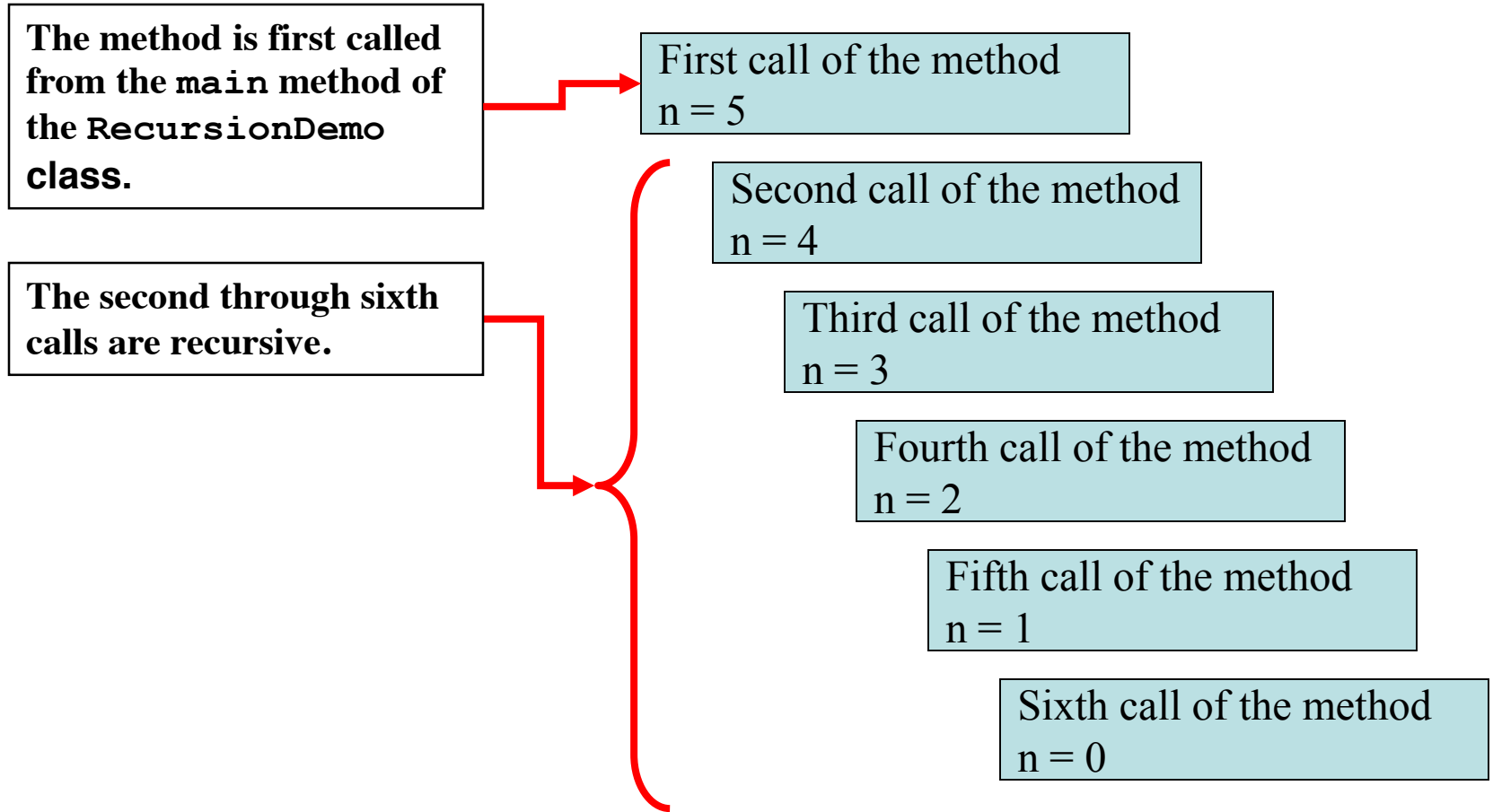
Introduction to Recursion

- We have been calling other methods from a method.
- It's also possible for a method to call itself.
- A method that calls itself is a *recursive method*.
- Example: [EndlessRecursion.java](#)

Introduction to Recursion

- This method in the example displays the string “This is a recursive method.”, and then calls itself.
- Each time it calls itself, the cycle is repeated endlessly.
- Like a loop, a recursive method must have some way to control the number of times it repeats.
- Example: [Recursive.java](#), [RecursionDemo.java](#)

Introduction to Recursion



Solving Problems With Recursion

- Recursion can be a powerful tool for solving repetitive problems.
- Recursion is never absolutely required to solve a problem.
- Any problem that can be solved recursively can also be solved iteratively, with a loop.
- In many cases, recursive algorithms are less efficient than iterative algorithms.

Solving Problems With Recursion

- Recursive solutions repetitively:
 - allocate memory for parameters and local variables, and
 - store the address of where control returns after the method terminates.
- These actions are called *overhead* and take place with each method call.
- This overhead does not occur with a loop.
- Some repetitive problems are more easily solved with recursion than with iteration.
 - Iterative algorithms might execute faster; however,
 - a recursive algorithm might be designed faster.

Solving Problems With Recursion

- Recursion works like this:
 - A base case is established.
 - If matched, the method solves it and returns.
 - If the base case cannot be solved now:
 - the method reduces it to a smaller problem (recursive case) and calls itself to solve the smaller problem.
- By reducing the problem with each recursive call, the base case will eventually be reached and the recursion will stop.
- In mathematics, the notation $n!$ represents the factorial of the number n .

Solving Problems With Recursion

- The factorial of a nonnegative number can be defined by the following rules:
 - If $n = 0$ then $n! = 1$
 - If $n > 0$ then $n! = 1 \times 2 \times 3 \times \dots \times n$
- Let's replace the notation $n!$ with `factorial( $n$ )`, which looks a bit more like computer code, and rewrite these rules as:
 - If $n = 0$ then `factorial( $n$ ) = 1`
 - If $n > 0$ then `factorial( $n$ ) = 1 × 2 × 3 × ... ×  $n$`

Solving Problems With Recursion

- These rules state that:
 - when n is 0, its factorial is 1, and
 - when n greater than 0, its factorial is the product of all the positive integers from 1 up to n .
- Factorial(6) is calculated as
 - $1 \times 2 \times 3 \times 4 \times 5 \times 6$.
- The base case is where n is equal to 0:
if $n = 0$ then factorial(n) = 1
- The recursive case, or the part of the problem that we use recursion to solve is:
 - ***if $n > 0$ then factorial(n) = $n \times \text{factorial}(n - 1)$***

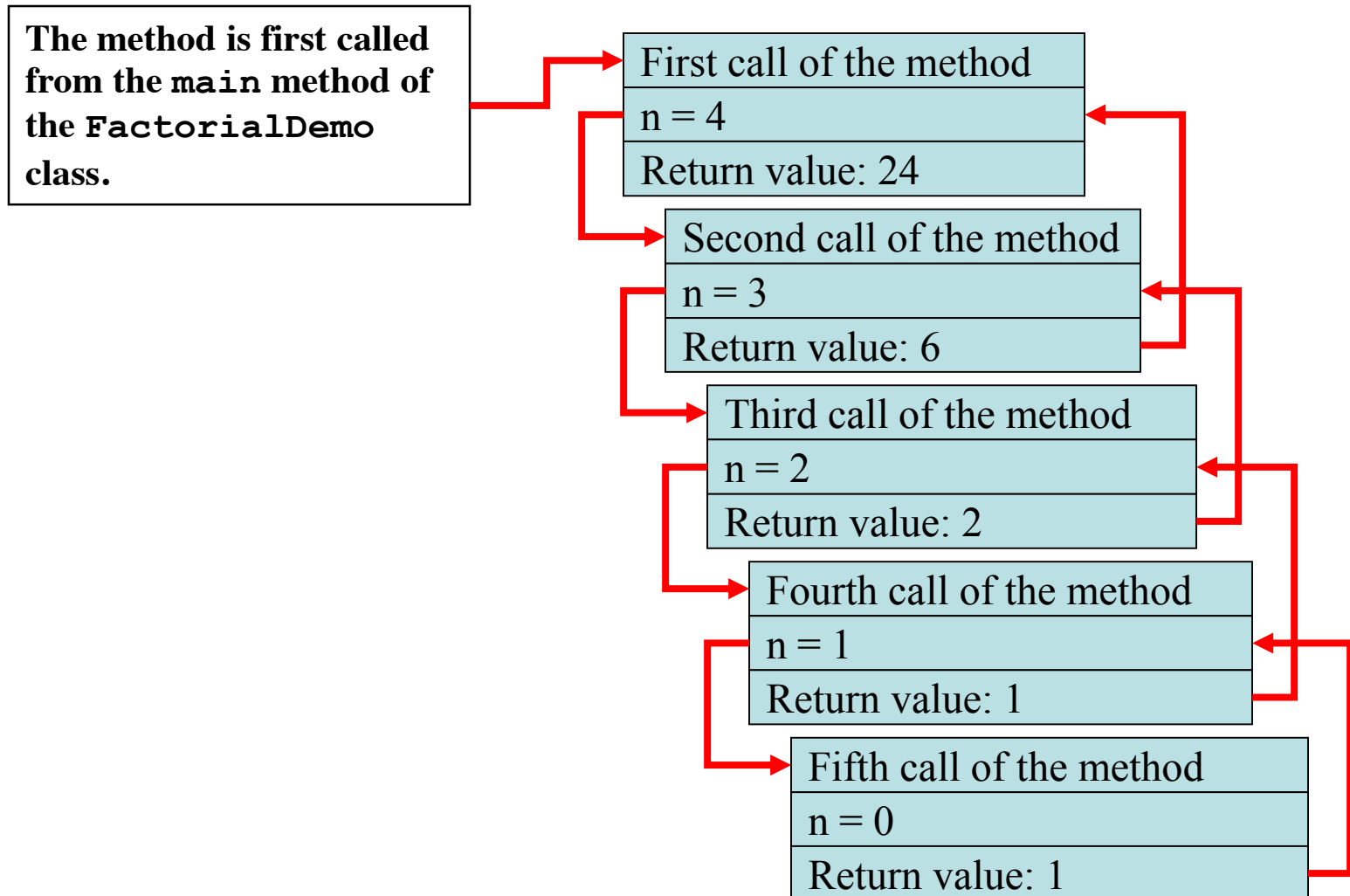
Solving Problems With Recursion

- The recursive call works on a reduced version of the problem, $n - 1$.
- The recursive rule for calculating the factorial:
 - If $n = 0$ then $\text{factorial}(n) = 1$
 - If $n > 0$ then $\text{factorial}(n) = n \times \text{factorial}(n - 1)$
- A Java based solution:

```
private static int factorial(int n)
{
    if (n == 0) return 1; // Base case
    else return n * factorial(n - 1);
}
```

- Example: [FactorialDemo.java](#)

Solving Problems With Recursion



Direct and Indirect Recursion

- When recursive methods directly call themselves it is known as *direct recursion*.
- *Indirect recursion* is when method **A** calls method **B**, which in turn calls method **A**.
- There can even be several methods involved in the recursion.
- Example, method **A** could call method **B**, which could call method **C**, which calls method **A**.
- Care must be used in indirect recursion to ensure that the proper base cases and return values are handled.

Summing a Range of Array Elements

- Recursion can be used to sum a range of array elements.
- A method, `rangeSum` takes following arguments:
 - an `int` array,
 - an `int` specifying the starting element of the range, and
 - an `int` specifying the ending element of the range.
 - How it might be called:

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};  
int sum;  
sum = rangeSum(numbers, 3, 7);
```

Summing a Range of Array Elements

- The definition of the `rangeSum` method:

```
public static int rangeSum(int[] array,  
                           int start, int end)  
{  
    if (start > end)  
        return 0;  
    else  
        return array[start]  
            + rangeSum(array, start + 1, end);  
}
```

- Example: [RangeSum.java](#)

Drawing Concentric Circles

- The definition of the `drawCircles` method:

```
private void drawCircles(Graphics g, int n, int
                        topXY, int fsize)
{
    if (n > 0)
    {
        g.drawOval(topXY, topXY, size, size);
        drawCircles(g, n - 1, topXY + 15, size - 30);
    }
}
```

- Example: [Circles.java](#)

The Fibonacci Series

- Some mathematical problems are designed to be solved recursively.
- One well known example is the calculation of *Fibonacci numbers*.:
 - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233,...
- After the second number, each number in the series is the sum of the two previous numbers.
- The Fibonacci series can be defined as:
 - $F0 = 0$
 - $F1 = 1$
 - $FN = FN-1 + FN-2$ for $N \geq 2$.

The Fibonacci Series

```
public static int fib(int n)
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```

- This method has two base cases:
 - when n is equal to 0, and
 - when n is equal to 1.
- Example: [FibNumbers.java](#)

Greatest Common Divisor (GCD)

- The definition of the gcd method:

```
public static int gcd(int x, int y)
{
    if (x % y == 0)
        return y;
    else
        return gcd(y, x % y);
}
```

- Example: [GCDdemo.java](#)

Recursive Binary Search

- The binary search algorithm can be implemented recursively.
- The procedure can be expressed as:
`If array[middle] equals the search value, then the value
is found.`
`Else`
`if array[middle] is less than the search value, do a
binary search on the upper half of the array.`
`Else`
`if array[middle] is greater than the search value,
perform a binary search on the lower half of the
array.`
- Example: [RecursiveBinarySearch.java](#)

The Towers of Hanoi

- The Towers of Hanoi is a mathematical game that uses:
 - three pegs and
 - a set of discs with holes through their centers.
- The discs are stacked on the leftmost peg, in order of size with the largest disc at the bottom.
- The object of the game is to move the discs from the left peg to the right peg by these rules:
 - Only one disk may be moved at a time.
 - A disk cannot be placed on top of a smaller disc.
 - All discs must be stored on a peg except while being moved.

The Towers of Hanoi

- The overall solution to the problem is to move n discs from peg 1 to peg 3 using peg 2 as a temporary peg.
- This algorithm solves the game.

If $n > 0$ Then

**Move $n - 1$ discs from peg A to peg B,
using peg C as a temporary peg.**

Move the remaining disc from the peg A to peg C.

**Move $n - 1$ discs from peg B to peg C,
using peg A as a temporary peg.**

End If

- The base case for the algorithm is reached when there are no more discs to move.
- Example: [Hanoi.java](#), [HanoiDemo.java](#)

CHAPTER 17

Databases

starting out with >>>

JAVA™

From Control Structures through Objects

6TH EDITION



TONY GADDIS

Chapter Topics

Chapter 16 discusses the following main topics:

- Introduction to Database Management Systems
- Tables, Rows, and Columns
- Introduction to the SQL `SELECT` Statement
- Inserting Rows
- Updating and Deleting Existing Rows
- Creating and Deleting Tables
- Creating a New Database with JDBC

Chapter Topics (2)

- Scrollable Result Sets
- Result Set Metadata
- Displaying Query Results in a `JTable`
- Relational Data
- Advanced Topics

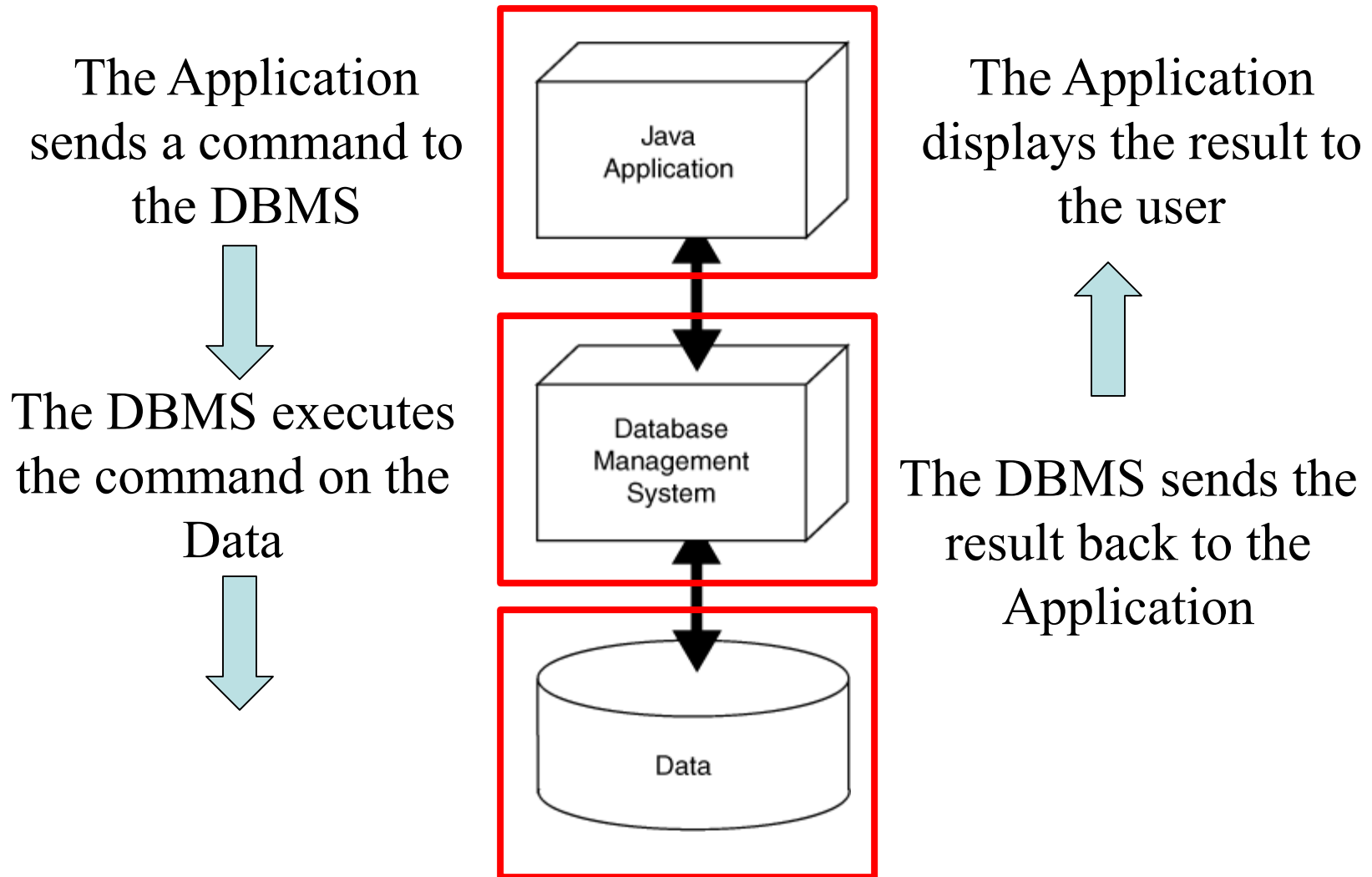
Introduction to Database Management Systems

- Storing data in traditional text or binary files has its limits
 - well suited for applications that store only a small amount of data
 - not practical for applications that must store a large amount of data
 - simple operations become cumbersome and inefficient as data increases

Introduction to Database Management Systems

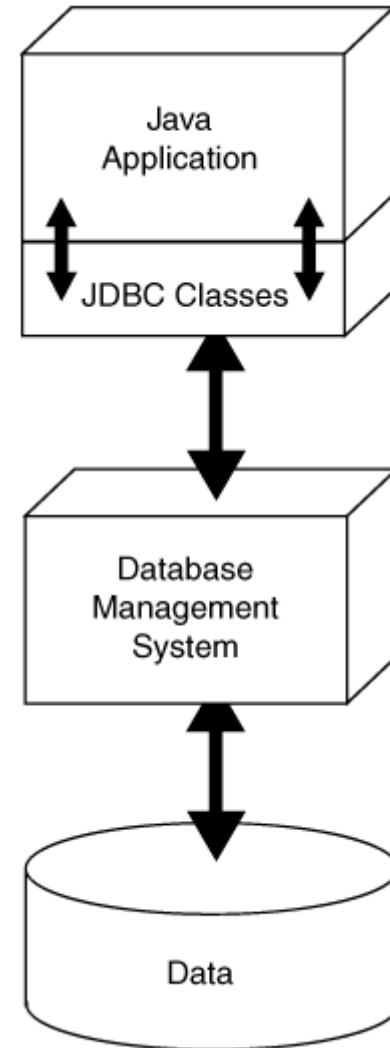
- A *database management system (DBMS)* is software that is specifically designed to work with large amounts of data in an efficient and organized manner
 - Data is stored using the database management system
 - Applications written in Java or other languages communicate with the DBMS rather than manipulate the data directly
 - DBMS carries out instructions and sends the results back to the application

A Java Application Interacts with a DBMS, Which Manipulates Data



JDBC Provides Connectivity to the DBMS

- JDBC stands for *Java database connectivity*
- It is the technology that makes communication possible between the Java application and DBMS
- The Java API contains numerous JDBC classes that allow your Java applications to interact with a DBMS



SQL Sends Commands to the DBMS

- SQL stands for *structured query language*
- A standard language for working with database management systems
- Not used as a general programming language
- Consists of several key words, used to construct statements known as *queries*
- Statements or queries are strings passed from the application to the DBMS using API method calls
- Serve as instructions for the DBMS to carry out operations on its data

JDBC Needs a DBMS

- To use JDBC to work with a database you will need a DBMS
 - Java DB
 - Oracle
 - Microsoft SQL Server
 - DB2
 - MySQL
- The examples in this chapter were created with Java DB

JDBC Classes

- Java comes with a standard set of JDBC classes
 - `java.sql` and `javax.sql`
- Using JDBC in a Java application requires the following steps
 1. Get a connection to the database
 2. Pass a string containing an SQL statement to the DBMS
 3. If the SQL statement has results to send back, they will be sent back as a result set
 4. When finished working with the database , close the connection

Getting a Database Connection

- The static `DriverManager.getConnection` method is used to get a connection to the database

- General format of the simplest version:

```
DriverManager.getConnection(DatabaseURL) ;
```

- General format if a user name and a password are required:

```
DriverManager.getConnection(DatabaseURL,  
                             Username,  
                             Password) ;
```

- *Username* is a string containing a valid username
 - *Password* is a string containing a password
 - *DatabaseURL* lists the protocol used to access the database

Getting a Database Connection

- *DatabaseURL* is a string known as a *database URL*
 - URL stands for uniform resource locator
- A simple database URL has the following general format:

protocol:subprotocol:databaseName

- *protocol* is the database protocol
 - value is `jdbc` when using JDBC
 - *subprotocol* varies depending on the type of DBMS
 - value is `derby` when using Java DB
 - *databaseName* is the name of the database
- Using Java DB, the URL for the `CoffeeDB` database is:

`jdbc:derby:CoffeeDB`

Getting a Database Connection

- The `DriverManager.getConnection` method
 - Searches for and loads a compatible JDBC driver for the database specified by the URL
 - Returns a reference to a `Connection` object
 - Should be saved in a variable, so it can be used later
 - Throws an `SQLException` if it fails to load a compatible JDBC driver

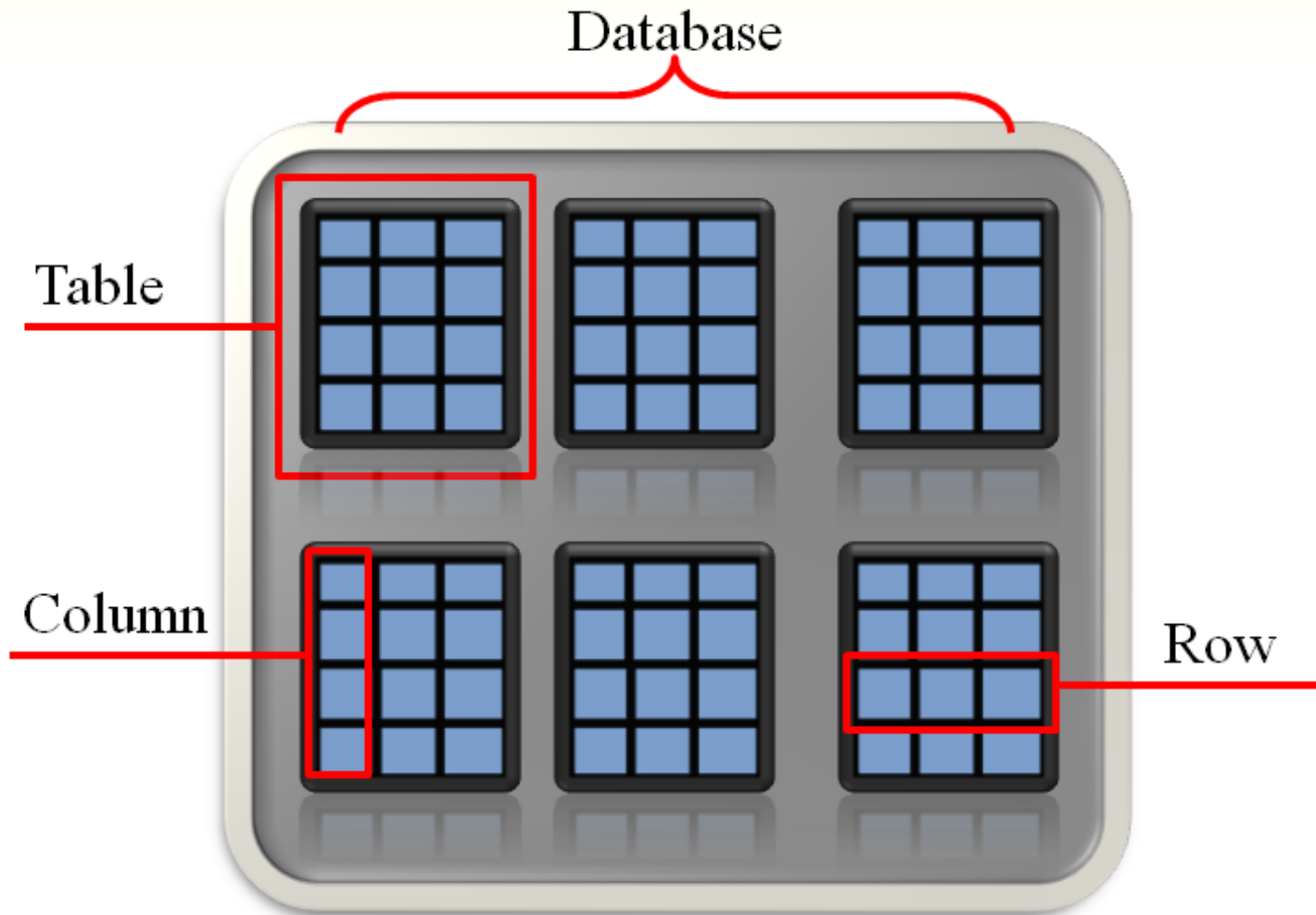
```
Final String DB_URL = "jdbc:derby:CoffeeDB";  
Connection conn = DriverManager.getConnection(DB_URL);
```

- Example: [TestConnection.java](#)

Tables, Rows, and Columns

- A database management system stores data in a database
- A *database* is organized into one or more tables
- Each *table* holds a collection of related data, organized into rows and columns
- A *row* is a complete set of information about a single item, divided into columns
- Each *column* is an individual piece of information about the item

Database Organization



Parts of the Coffee Database Table

Each row
contains
data for a
single item.



Description	ProdNum	Price
Bolivian Dark	14-001	8.95
Bolivian Medium	14-002	8.95
Brazilian Dark	15-001	7.95
Brazilian Medium	15-002	7.95
Brazilian Decaf	15-003	8.55
Central American Dark	16-001	9.95
Central American Medium	16-002	9.95
Sumatra Dark	17-001	7.95
Sumatra Decaf	17-002	8.95
Sumatra Medium	17-003	7.95
Sumatra Organic Dark	17-004	11.95
Kona Medium	18-001	18.45
Kona Dark	18-002	18.45
French Roast Dark	19-001	9.65
Galapagos Medium	20-001	6.85
Guatemalan Dark	21-001	9.95
Guatemalan Decaf	21-002	10.45
Guatemalan Medium	21-003	9.95



Description
Column



ProdNum
Column



Price
Column

Column Data Types

- Columns in a database are assigned an SQL data type
 - SQL data types are generally compatible with Java data types

SQL Data Type	Description	Corresponding Java Data Type
INTEGER or INT	An integer number	int
CHARACTER (<i>n</i>) or CHAR (<i>n</i>)	A fixed-length string with a length of <i>n</i> characters	String
VARCHAR (<i>n</i>)	A variable-length string with a maximum length of <i>n</i> characters.	String
REAL	A single-precision floating point number	float
DOUBLE	A double-precision floating point number	double
DECIMAL (<i>t</i> , <i>d</i>)	A decimal value with <i>t</i> total digits and <i>d</i> digits appearing after the decimal point.	java.math.BigDecimal
DATE	A date	java.sql.Date

The Coffee Table Column Data Types

- `Description` column data type is `CHAR(25)`
 - String with a fixed length of 25 characters
 - Compatible with the `String` type in Java
- `ProdNum` column data type is `CHAR(10)`
 - String with a fixed length of 10 characters
 - Compatible with the `String` type in Java
- `Price` column data type is `DOUBLE`
 - Double-precision floating-point number
 - Compatible with the `double` data type in Java

Primary Keys

- A *primary key* is a column that holds a unique value for each row in a database table
- In the `Coffee` table, `ProdNum` is the primary key
 - Each type of coffee has a unique product number
 - Used to identify any coffee stored in the table
- A primary key can be the combination of several columns in a table

Introduction to the SQL SELECT Statement

- The SELECT statement is used to retrieve the rows in a table

```
SELECT Columns FROM Table
```

- *Columns* is one or more column names
- *Table* is a table name

- Example 1:

```
SELECT Description FROM Coffee
```

- Example 2:

```
SELECT Description, Price FROM Coffee
```

- Multiple column names are separated with a comma

- Example 3:

```
SELECT * FROM Coffee
```

- The * character can be used to retrieve all columns in the table

More About SQL Statements

- SQL statements are free form
 - tabs, new lines, and spaces between key words are ignored
- SQL key words and table names are case insensitive
- Example: *The following statements all work the same:*

```
SELECT * FROM Coffee
```

```
SELECT
    *
FROM
    Coffee
```

```
select * from coffee
```

Passing an SQL Statement to the DBMS

- Once you have established a connection, you must get a reference to a `Statement` object before you can issue SQL statements to the DBMS
 - A `Statement` object has an `executeQuery` method that returns a reference to a `ResultSet` object
 - A `ResultSet` object contains the results of the query
- Example:

```
Connection conn = DriverManager.getConnection(DB_URL);  
Statement stmt = conn.createStatement();  
String sqlStatement = "SELECT Description FROM Coffee";  
ResultSet result = stmt.executeQuery(sqlStatement);
```


Getting a Row from the `ResultSet` Object

- A `ResultSet` object has an internal *cursor*
 - Points to a specific row in the `ResultSet`
 - The row to which it points is the *current row*
 - Initially positioned just before the first row
 - Can be moved from row to row to examine all rows

Initially the cursor is positioned just before the first row in the `ResultSet`.

Cursor	→			
Row 1		Sumatra Organic Dark	17-004	11.95
Row 2		Kona Medium	18-001	18.45
Row 3		Kona Dark	18-002	18.45
Row 4		Guatemalan Decaf	21-002	10.45

Getting a Row from the ResultSet Object

- A `ResultSet` object's `next` method moves the cursor to the next row in the `ResultSet`

```
result.next();
```

- moves to first row in a newly created `ResultSet`
- moves to the next row each time it is called

After the `ResultSet` object's `next` method is called the first time, the cursor is positioned at the first row.

Cursor →	Row 1	Sumatra Organic Dark	17-004	11.95
	Row 2	Kona Medium	18-001	18.45
	Row 3	Kona Dark	18-002	18.45
	Row 4	Guatemalan Decaf	21-002	10.45

Getting a Row from the `ResultSet` Object

- A `ResultSet` object's `next` method returns a `Boolean` value
 - `true` if successfully moved to the next row
 - `false` if there are no more rows
- A `while` loop can be used to move through all the rows of a newly created `ResultSet`

```
while (result.next())  
{  
    // Process the current row.  
}
```

Getting Columns in a ResultSet Object

- You use one of the `ResultSet` object's "get" methods to retrieve the contents of a specific column in the current row.
- Can pass an argument for either the column number or the column name

```
System.out.println(result.getString(1));  
System.out.println(result.getString(1));  
System.out.println(result.getString(1));
```

```
System.out.println(result.getString("Description"));  
System.out.println(result.getString("ProdNum"));  
System.out.println(result.getDouble("Price"));
```

Examples: [ShowCoffeeDescriptions.java](#)
 [ShowDescriptionsAndPrices.java](#)

Getting Columns in a ResultSet Object

ResultSet Method	Description
<code>double getDouble(int colNumber)</code> <code>double getDouble(String colName)</code>	Returns the <code>double</code> that is stored in the column specified by <code>colNumber</code> or <code>colName</code> . The column must hold data that is compatible with the <code>double</code> data type in Java. If an error occurs, the method throws an <code>SQLException</code> .
<code>int getInt(int colNumber)</code> <code>int getInt(String colName)</code>	Returns the <code>int</code> that is stored in the column specified by <code>colNumber</code> or <code>colName</code> . The column must hold data that is compatible with the <code>int</code> data type in Java. If an error occurs, the method throws an <code>SQLException</code> .
<code>String getString(int colNumber)</code> <code>String getString(String colName)</code>	Returns the <code>string</code> that is stored in the column specified by <code>colNumber</code> or <code>colName</code> . The column must hold data that is compatible with the <code>String</code> type in Java. If an error occurs, the method throws an <code>SQLException</code> .

Specifying Search Criteria with the WHERE clause

- The WHERE clause can be used with the SELECT statement to specify a search criteria

`SELECT Columns FROM Table WHERE Criteria`

– *Criteria* is a conditional expression

- Example:

`SELECT * FROM Coffee WHERE Price > 12.00`



<u>Description</u>	<u>ProdNum</u>	<u>Price</u>
Kona Medium	18-001	18.45
Kona Dark	18-002	18.45

- Only the rows that meet the search criteria are returned in the result set
- A *result set* is an object that contains the results of an SQL statement

SQL Relational Operators

- Standard SQL supports the following relational operators:

Operator	Meaning
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Not equal to

- Notice a few SQL relational operators are different than in Java
 - SQL equal to operator is =
 - SQL not equal to operator is <>

Example: [CoffeeMinPrice.java](#)

String Comparisons in SQL

- Example 1:

```
SELECT * FROM Coffee WHERE Description = 'French Roast Dark'
```

- In SQL, strings are enclosed in single quotes

- Warning!

```
SELECT * FROM Coffee WHERE Description = 'french roast dark'
```

- String comparisons in SQL are case sensitive

- Example 2:

```
SELECT * FROM Coffee  
WHERE UPPER(Description) = 'FRENCH ROAST DARK'
```

- The UPPER() or LOWER() functions convert the string to uppercase or lowercase and can help prevent case sensitive errors when comparing strings

- Example 3:

```
SELECT * FROM Coffee WHERE Description = 'Joe''s Special Blend'
```

- If a single quote (') is part of a string, use two single quotes (' ')

Using the LIKE Operator

- In SQL, the LIKE operator can be used to search for a substring
- Example 1:

```
SELECT * FROM Coffee WHERE Description LIKE '%Decaf%'
```

- The % symbol is used as a wildcard for multiple characters

- Example 2:

```
SELECT * FROM Coffee WHERE ProdNum LIKE '2_-00_'
```

- The underscore () is used as a wildcard for a single character

- Example 3:

```
SELECT * FROM Coffee  
WHERE Description NOT LIKE '%Decaf%'
```

- The NOT operator is used to disqualify the search criteria

Using AND and OR

- The AND and OR operators can be used to specify multiple search criteria in a WHERE clause
- Example 1:

```
SELECT * FROM Coffee
      WHERE Price > 10.00 AND Price < 14.00
```

- The AND operator requires that *both* search criteria be true

- Example 2:

```
SELECT * FROM Coffee
      WHERE Description LIKE '%Dark%' OR ProdNum LIKE '16%'
```

- The OR operator requires that *either* search criteria be true

Sorting the Results of a `SELECT` Query

- Use the `ORDER BY` clause to sort results according to a column value
- Example 1:

```
SELECT * FROM Coffee ORDER BY Price
```

- Sorted in ascending order (ASC) by default

- Example 2:

```
SELECT * FROM Coffee  
WHERE Price > 9.95 ORDER BY Price DESC
```

- Use the `DESC` operator to sort results in descending order

Mathematical Functions

Example: [CoffeeMath.java](#)

- The AVG function
 - calculates the average value in a particular column

```
SELECT AVG(Price) FROM Coffee
```
- The SUM function
 - calculates the sum of a column's values

```
SELECT SUM(Price) FROM Coffee
```
- The MIN and MAX functions
 - calculate the minimum and maximum values found in a column

```
SELECT MIN(Price) FROM Coffee
```

```
SELECT MAX(Price) FROM Coffee
```
- The COUNT function
 - can be used to determine the number of rows in a table

```
SELECT COUNT(*) FROM Coffee
```

Inserting Rows

- In SQL, the INSERT statement inserts a row into a table

```
INSERT INTO TableName VALUES (Value1, Value2, ...)
```

- *TableName* is the name of the database table
- *Value1*, *Value2*, ... is a list of column values

- Example:

```
INSERT INTO Coffee  
VALUES ('Honduran Dark', '22-001', 8.65)
```

- Strings are enclosed in single quotes
- Values appear in the same order as the columns in the table

- Inserts a new row with the following column values:

```
Description: Honduran Dark  
ProdNum: 22-001  
Price: 8.65
```

Inserting Rows

- If column order is uncertain, the following general format can be used

```
INSERT INTO TableName
    (ColumnName1, ColumnName2, ...)
VALUES
    (Value1, Value2, ...)
```

- *ColumnName1*, *ColumnName2*, ... is a list of column names
 - *Value1*, *Value2*, ... is a list of corresponding column values
- Example:

```
INSERT INTO Coffee
    (ProdNum, Price, Description)
VALUES
    ('22-001', 8.65, 'Honduran Dark')
```
- Keep in mind that primary key values must be unique
- For example, a duplicate `ProdNum` is not allowed in the `Coffee` table

Inserting Rows with JDBC

- To issue an INSERT statement, you must get a reference to a Statement object
 - The Statement object has an executeUpdate method
 - Accepts a string containing the SQL INSERT statement as an argument
 - Returns an int value for the number of rows inserted
- Example:

```
String sqlStatement = "INSERT INTO Coffee " +  
                      "(ProdNum, Price, Description)" +  
                      " VALUES " +  
                      "('22-001', 8.65, 'Honduran Dark')";  
  
int rows = stmt.executeUpdate(sqlStatement);
```

- rows should contain the value 1, indicating that one row was inserted

Example: [CoffeeInserter.java](#)

Updating an Existing Row

- In SQL, the UPDATE statement changes the contents of an existing row in a table

```
UPDATE Table
      SET Column = Value
      WHERE Criteria
```

- *Table* is a table name
 - *Column* is a column name
 - *Value* is the value to store in the column
 - *Criteria* is a conditional expression
- **Example:**

```
UPDATE Coffee
      SET Price = 9.95
      WHERE Description = 'Galapagos Organic Medium'
```


Updating More Than One Row

- It is possible to update more than one row
- Example:

```
UPDATE Coffee
  SET Price = 12.95
  WHERE ProdNum LIKE '21%'
```

- Updates the price of all rows where the product number begins with 21

- Warning!

```
UPDATE Coffee
  SET Price = 4.95
```

- Because this statement does not have a WHERE clause, it will change the price for every row

Updating Rows with JDBC

- To issue an UPDATE statement, you must get a reference to a Statement object
 - The Statement object has an executeUpdate method
 - Accepts a string containing the SQL UPDATE statement as an argument
 - Returns an int value for the number of rows affected
- Example:

```
String sqlStatement = "UPDATE Coffee " +  
                      "SET Price = 9.95" +  
                      " WHERE " +  
                      "Description = 'Brazilian Decaf'";  
  
int rows = stmt.executeUpdate(sqlStatement);
```

- rows indicates the number of rows that were changed

Example: [CoffeePriceUpdater.java](#)

Deleting Rows with the DELETE Statement

- In SQL, the DELETE statement deletes one or more rows in a table

`DELETE FROM Table WHERE Criteria`

- *Table* is the table name
- *Criteria* is a conditional expression

- Example 1:

`DELETE FROM Coffee WHERE ProdNum = '20-001'`

- Deletes a single row in the `Coffee` table where the product number is 20-001

- Example 2:

`DELETE FROM Coffee WHERE Description LIKE 'Sumatra%'`

- Deletes all rows in the `Coffee` table where the description begins with Sumatra

- Warning!

`DELETE FROM Coffee`

- Because this statement does not have a WHERE clause, it will delete every row in the `Coffee` table

Deleting Rows with JDBC

- To issue a DELETE statement, you must get a reference to a Statement object
 - The Statement object has an executeUpdate method
 - Accepts a string containing the SQL DELETE statement as an argument
 - Returns an int value for the number of rows that were deleted
- Example:

```
String sqlStatement = "DELETE FROM Coffee " +  
                      "WHERE ProdNum = '20-001'";  
int rows = stmt.executeUpdate(sqlStatement);
```

- rows indicates the number of rows that were deleted

Example: [CoffeeDeleter.java](#)

Creating Tables with the CREATE TABLE Statement

- In SQL, the CREATE TABLE statement adds a new table to the database

```
CREATE TABLE TableName
    (ColumnName1 DataType1,
     ColumnName2 DataType2, ...)
```

- *TableName* is the name of the table
 - *ColumnName1* is the name of the first column
 - *DataType1* is the SQL data type for the first column
 - *ColumnName2* is the name of the second column
 - *DataType2* is the SQL data type for the second column
- Example:

```
CREATE TABLE Customer
    ( Name CHAR(25), Address CHAR(25),
      City CHAR(12), State CHAR(2), Zip CHAR(5) )
```

 - Creates a new table named Customer with the columns Name, Address, City, State, and Zip

Creating Tables with the CREATE TABLE Statement

- The PRIMARY KEY qualifier is used to specify a column as the primary key
- The NOT NULL qualifier is used to specify that the column must contain a value for every row
 - Qualifiers should be listed *after* the column's data type
- Example: [CreateCustomerTable.java](#)

```
CREATE TABLE Customer
( CustomerNumber CHAR(10) NOT NULL PRIMARY KEY
  Name CHAR(25), Address CHAR(25),
  City CHAR(12), State CHAR(2), Zip CHAR(5) )
```

- Creates a new table named Customer with the columns CustomerNumber, which is the primary key, Name, Address, City, State, and Zip

Removing a Table with the DROP TABLE Statement

- In SQL, the DROP TABLE statement deletes an existing table from the database

```
DROP TABLE TableName
```

- *TableName* is the name of the table you wish to delete

- Example:

```
DROP TABLE Customer
```

- Deletes the Customer table from the CoffeeDB database
- Useful if you make a mistake creating a table
- Simply delete the table and recreate

Creating a New Database with Java DB

- The `;create=true` attribute creates a new database when appended to the database URL
 - `"jdbc:derby:EntertainmentDB;create=true"`
 - Creates an empty database named `EntertainmentDB`
 - The `CREATE TABLE` statement can be used to create tables
 - Java DB creates a folder with the name of the database on your system
 - Delete the database folder to delete the database
- Example: [BuildEntertainmentDB.java](#)

Scrollable Result Sets

- By default, a `ResultSet` object is created with a read-only concurrency level and the cursor is limited to forward movement
- A *scrollable result set* can be created with the overloaded version the `Connection` object's `createStatement` method

```
conn.createStatement(type, concur);
```

- *type* is a constant for the scrolling type
- *concur* is a constant for the concurrency level
- Example:

```
Statement stmt =  
    conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,  
                          ResultSet.CONCUR_READ_ONLY);
```

 - Creates a scrollable result set that is read-only and insensitive to database changes

The ResultSet Scrolling Types

- `ResultSet.TYPE_FORWARD_ONLY`
 - Default scrolling type
 - Cursor moves forward only
- `ResultSet.TYPE_SCROLL_INSENSITIVE`
 - Cursor moves both forward and backward
 - Changes made to the database do not appear
- `ResultSet.TYPE_SCROLL_SENSITIVE`
 - Cursor moves both forward and backward
 - Changes made to the database appear as soon as they are made

The ResultSet Concurrency Levels

- `ResultSet.CONCUR_READ_ONLY`
 - Default concurrency level
 - Read-only version of data from the database
 - Cannot change database by altering result set
- `ResultSet.CONCUR_UPDATEABLE`
 - Result set is updateable
 - Changes can be made to the result set and saved to the database
 - Uses methods that allow changes to be made to the database without issuing SQL statements

ResultSet Navigation Methods

- `first()`
 - Moves the cursor to the first row
- `last()`
 - Moves the cursor to the last row
- `next()`
 - Moves the cursor to the next row
- `previous()`
 - Moves the cursor to the previous row

ResultSet Navigation Methods

- `relative(rows)`
 - Moves the cursor the number specified by the *rows* argument relative to the current row
 - A positive *rows* value will move the cursor forward
 - A negative *rows* value will move the cursor backward
- `absolute(rows)`
 - Moves the cursor to the row number specified by the *rows* argument
 - A *rows* value of 1 will move the cursor to the first row
 - A *rows* value of 2 will move cursor to the second row
 - And so on until the last row

Determining the Number of Rows in a Result Set

- ResultSet navigation methods can be used to determine the number of rows in a result set
- Example:

```
resultSet.last()           // Move to the last row
int numRows = resultSet.getRow(); // Get the current row number
resultSet.first();         // Move back to the first row
```

- Move cursor to last row
- Get the last row's number and store the value
- Move back to the first row

Result Set Metadata

- Metadata refers to data that describes other data
- A `ResultSet` object has metadata that describes a result set
- Can be used to determine many things about a result set
 - Number of columns
 - Column names
 - Column data types
 - And much more
- Useful for submitting SQL queries in applications

Result Set Metadata

- `ResultSetMetaData` is an interface in the `java.sql` package
- The `getMetaData` method of a `ResultSet` object returns a reference to a `ResultSetMetaData` object.
- Example: [MetaDataDemo.java](#)

```
ResultSetMetaData meta = resultSet.getMetaData();
```

- Creates a `ResultSetMetaData` object reference variable named `meta`

A Few ResultSetMetaData Methods

Method	Description
<code>int getColumnCount()</code>	Returns the number of columns in the result set.
<code>String getColumnName(int col)</code>	Returns the name of the column specified by the integer <code>col</code> . The first column is column 1.
<code>String getColumnName(int col)</code>	Returns the name of the data type of the column specified by the integer <code>col</code> . The first column is column 1. The data type name returned is the database-specific SQL data type.
<code>int getColumnDisplaySize(int col)</code>	Returns the display width, in characters, of the column specified by the integer <code>col</code> . The first column is column 1.
<code>String getTableName(int col)</code>	Returns the name of the table associated with the column specified by the integer <code>col</code> . The first column is column 1.

The JTable Class

- The JTable class is a Swing component that displays data in a two-dimensional table

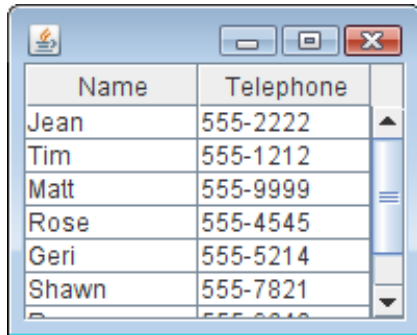
```
Jtable(Object[][] rowData, Object[] colNames)
```

- *rowData* is a two-dimensional array of objects
 - Contains data that will be displayed in the table
 - Each row becomes a row of data in the table
 - Each column becomes a column of data in the table
 - JTable calls toString method of each object to get values
- *colNames* is a one-dimensional array of objects
 - Contains the column names to display
 - JTable calls toString method of each object to get value

Setting Up a Simple JTable Component

- Example:

```
String[] colNames = {"Name", "Telephone" };  
String[][] rowData = {{ "Jean",  "555-2222" },  
                       { "Tim",    "555-1212" },  
                       { "Matt",   "555-9999" },  
                       { "Rose",   "555-4545" },  
                       { "Geri",   "555-5214" },  
                       { "Shawn",  "555-7821" },  
                       { "Renee",  "555-9640" },  
                       { "Joe",    "555-8657" } };
```



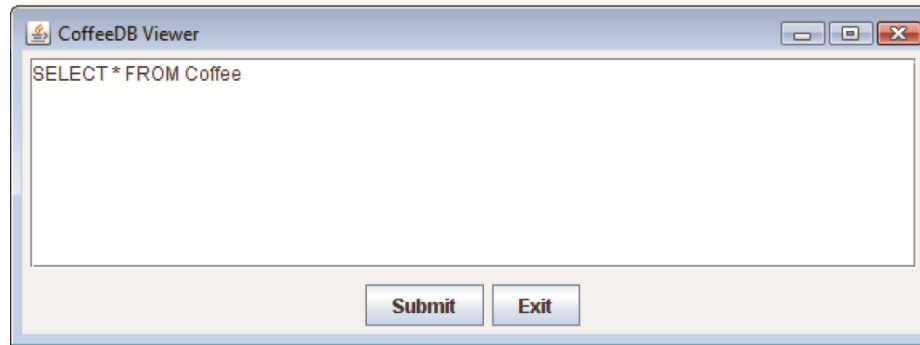
Name	Telephone
Jean	555-2222
Tim	555-1212
Matt	555-9999
Rose	555-4545
Geri	555-5214
Shawn	555-7821
Renee	555-9640

```
JTable myTable = new JTable(rowData, colNames);  
JScrollPane scrollPane = new JScrollPane(myTable);
```

- The figure shows an example of how the table appears in a frame

Displaying Query Results in a JTable

This window appears first



The user enters a SELECT statement and clicks the Submit button

DESCRIPTION	PRODNUM	PRICE
Bolivian Dark	14-001	8.95
Bolivian Medium ...	14-002	8.95
Brazilian Dark	15-001	7.95
Brazilian Medium ...	15-002	7.95
Brazilian Decaf	15-003	8.55
Central American D...	16-001	9.95
Central American M...	16-002	9.95
Sumatra Dark	17-001	7.95
Sumatra Decaf	17-002	8.95



This window appears next

It displays the results in a Jtable component

- Example: [TableFormatter.java](#), [CoffeeDBQuery.java](#), [CoffeeDBViewer.java](#)

Relational Data

- A *foreign key* is a column in one table that references a primary key in another table
 - Creates a relationship between the tables
- Example:

UnpaidOrder table:

CustomerNumber	CHAR (10)	<i>Foreign Key</i>
ProdNum	CHAR (10)	<i>Foreign Key</i>
OrderDate	CHAR (10)	
Quantity	DOUBLE	
Cost	DOUBLE	

- The CustomerNumber column references the Customer table
- The ProdNum column references the Coffee table
- This creates a relationship between the tables of the CoffeeDB database

Creating the UnpaidOrder Table

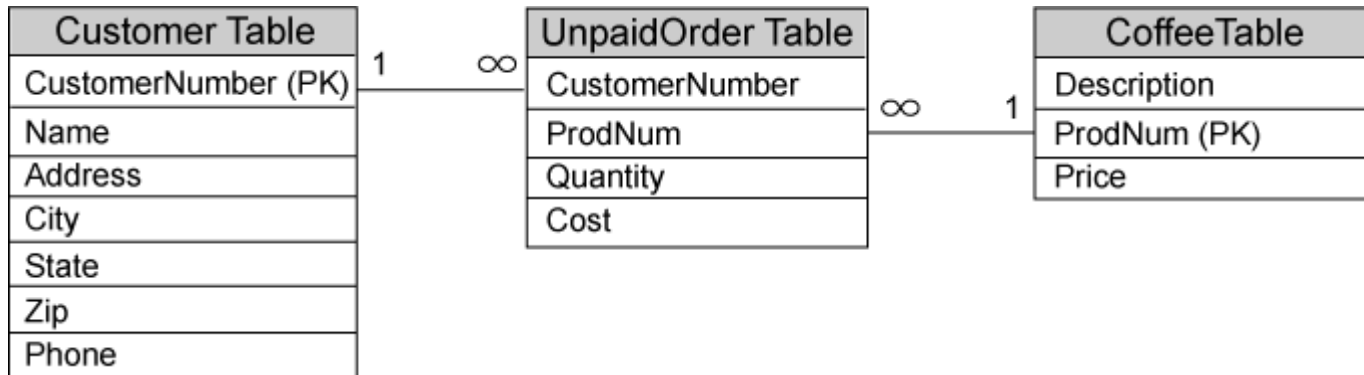
- The following SQL statement creates the UnpaidOrder table in the CoffeeDB database:

```
CREATE TABLE UnpaidOrder
( CustomerNumber CHAR(10) NOT NULL
  REFERENCES Customer(CustomerNumber),
  ProdNum CHAR(10) NOT NULL
  REFERENCES Coffee(ProdNum),
  OrderDate CHAR(10),
  Quantity DOUBLE,
  Cost DOUBLE )
```

- The REFERENCES qualifier ensures *referential integrity* between tables
 - The CustomerNumber in the UnpaidOrder table must contain a valid CustomerNumber from the Customer table
 - The ProdNum in the UnpaidOrder table must contain a valid ProdNum from the Coffee table

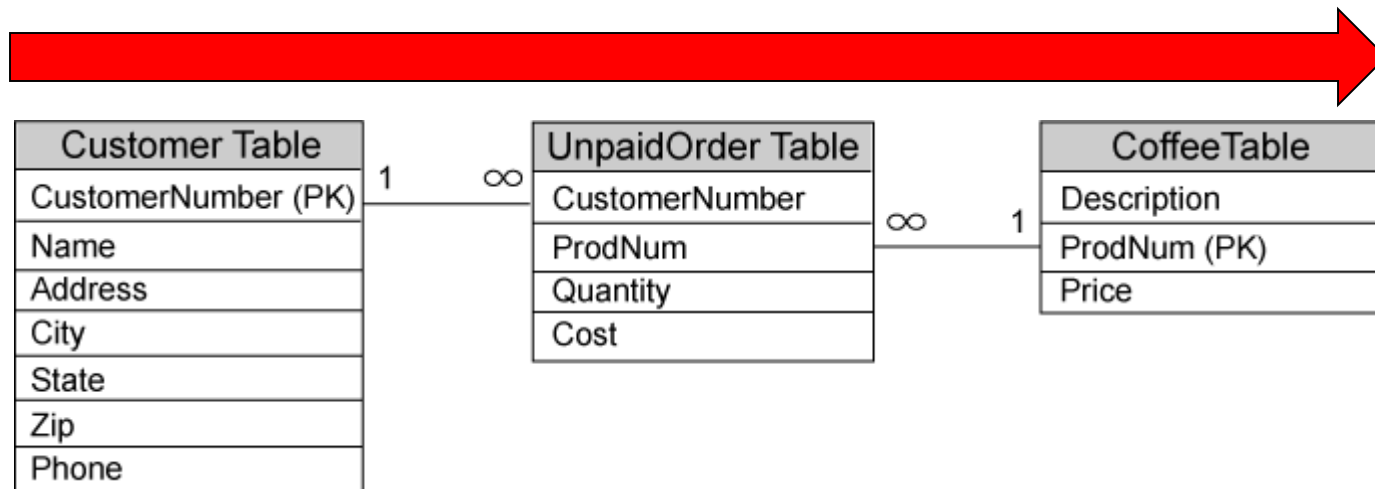
Entity Relationship Diagrams

- An *entity relationship diagram* shows the relationships between tables



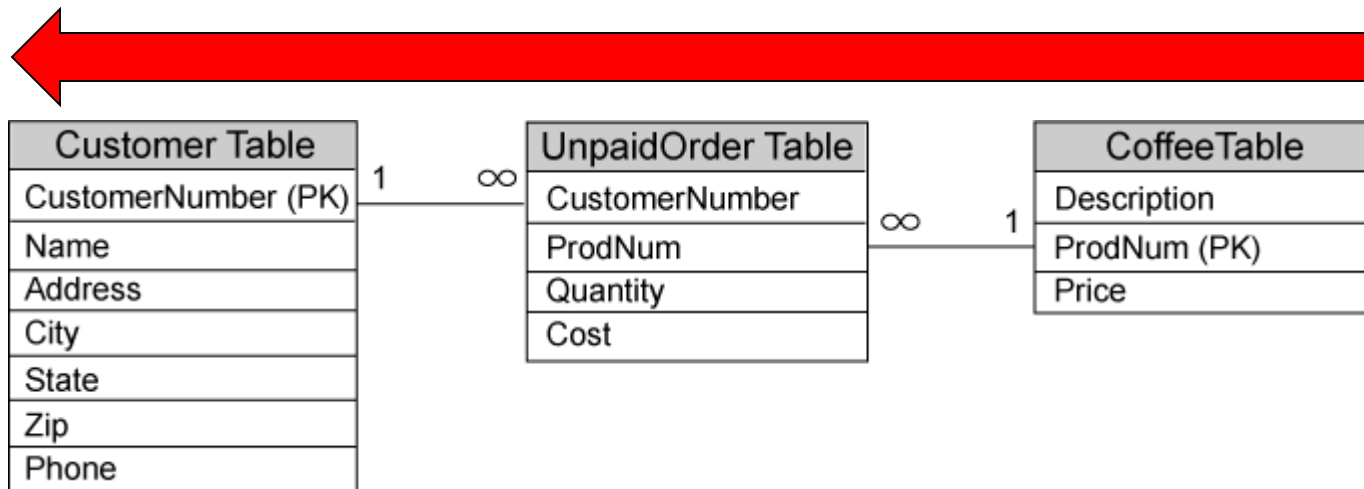
- Primary keys are denoted with (PK)
- Lines drawn between tables show how they are related
 - The ends of each line show either a 1 or an infinity symbol (∞)
 - The infinity symbol means *many* and number 1 means *one*.
 - A *one to many relationship* means that for each row in table A there can be many rows in table B that reference it.
 - A *many to one relationship* means that many rows in table A can reference a single row in table B.

CoffeeDB Relationships Left to Right



- One to many relationship between Customer and UnpaidOrder
 - One row in the Customer table may be referenced by many rows in the UnpaidOrder table
- Many to one relationship between the UnpaidOrder and Coffee tables
 - Many rows in the UnpaidOrder table may reference a single row in the Coffee table.

CoffeeDB Relationships Right to Left



- One to many relationship between Coffee and UnpaidOrder
 - One row in the Coffee table may be referenced by many rows in the UnpaidOrder table
- Many to one relationship between UnpaidOrder and Customer
 - Many rows in the UnpaidOrder table may reference a single row in the Customer table.

Joining Data from Multiple Tables

- In SQL, you must use qualified column names in a `SELECT` statement if the tables have columns with the same name
- A *qualified column name* takes the following form:

TableName.ColumnName

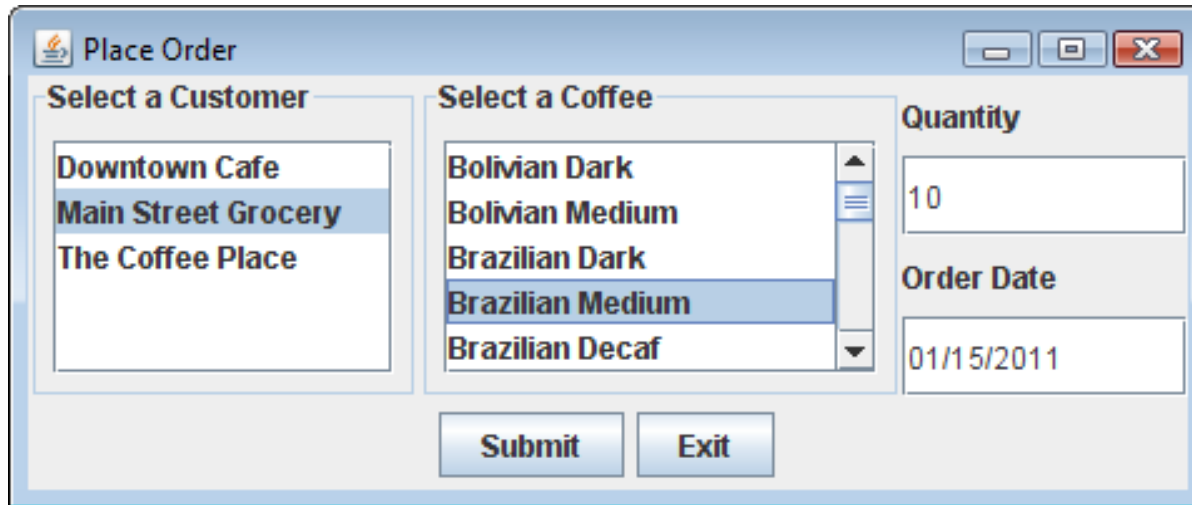
- Example:

```
SELECT
    Customer.CustomerNumber, Customer.Name,
    UnpaidOrder.OrderDate, UnpaidOrder.Cost,
    Coffee.Description
FROM
    Customer, UnpaidOrder, Coffee
WHERE
    UnpaidOrder.CustomerNumber = Customer.CustomerNumber
    AND
    UnpaidOrder.ProdNum = Coffee.ProdNum
```

- The search criteria tell the DBMS how to link the rows in the tables

An Order Entry System

- The *Place Order* application uses a relational database (CoffeeDB)
- Requires the Coffee, Customer, and UnpaidOrder tables



The screenshot shows a Java Swing window titled "Place Order". It contains three main input areas: a list box for "Select a Customer" with options "Downtown Cafe", "Main Street Grocery" (selected), and "The Coffee Place"; a list box for "Select a Coffee" with options "Bolivian Dark", "Bolivian Medium", "Brazilian Dark", "Brazilian Medium" (selected), and "Brazilian Decaf"; and a text field for "Quantity" with the value "10". Below these is a text field for "Order Date" with the value "01/15/2011". At the bottom are "Submit" and "Exit" buttons.

- Example: [CoffeeDBManager.java](#), [CustomerPanel.java](#),
[CoffeePanel.java](#), [QtyDatePanel.java](#),
[PlaceOrder.java](#)

Transactions

- An operation that requires multiple database updates is known as a *transaction*.
- For a transaction to be complete
 - All of the steps involved in the transaction must be performed.
- If any single step within a transaction fails
 - None of the steps in the transaction should be performed.
- When you write transaction-processing code, there are two concepts you must understand:
 - Commit
 - Rollback
- The term *commit* refers to making a permanent change to a database
- The term *rollback* refers to undoing changes to a database

JDBC Auto Commit Mode

- By default, the `JDBC Connection` class operates in auto commit mode.
- In *auto commit* mode
 - All updates that are made to the database are made permanent as soon as they are executed.
- When auto commit mode is turned off
 - Changes do not become permanent until a commit command is executed
 - A rollback command can be used to undo changes

JDBC Transaction Methods

- To turn auto commit mode off
 - Call the `Connection` class's `setAutoCommit` method
 - Pass the argument `false`

```
conn.setAutoCommit(false);
```

- To execute a commit command
 - Call the `Connection` class's `commit` method

```
conn.commit();
```

- To execute a rollback command
 - Call the `Connection` class's `rollback` method

```
conn.rollback();
```

JDBC Transaction Example

```
conn.setAutoCommit(false);
// Attempt the transaction
try
{
    // Update the inventory records.
    stmt.executeUpdate(updateStatement);
    // Add the order to the UnpaidOrder table.
    stmt.executeUpdate(insertStatement);
    // Commit all these updates.
    conn.commit();
}
catch (SQLException ex)
{
    // Roll back the changes.
    conn.rollback();
}
```

The commit
method is called in
the try block



The rollback
method is called in
the catch block

Stored Procedures

- Many commercial database systems allow you to create SQL statements and store them in the DBMS itself
- These SQL statements are called *stored procedures*
 - Can be executed by other applications using the DBMS
 - Ideal for SQL statements that are used often in a variety of applications
 - Usually execute faster than SQL statements that are submitted from applications outside the DBMS
- Each DBMS has its own syntax for creating a stored procedure in SQL
- To execute a stored procedure, you must create a `CallableStatement` object
- `CallableStatement` is an interface in the `java.sql` package
- To create a `CallableStatement` object, you call the `Connection` class's `prepareCall` statement