Java: How To Program

Mix Edition

CHAPTER 1

Introduction to Computers & Java

Objectives

In this chapter we'll

- Introduction to Computer Hardware & Software.
- Data Hierarchy
- Computer Organization
- Computer Languages
- Introduction to Object Oriented Programming

1.1- Introduction **to Computer Hardware & Software**

Computer:

A computer is a device that can perform computations and make logical decisions phenomenally faster than human beings can.

Program:

Computers process data under the control of sets of instructions called computer programs.

Programmer:

These programs guide the computer through orderly sets of actions specified by people called computer programmers.

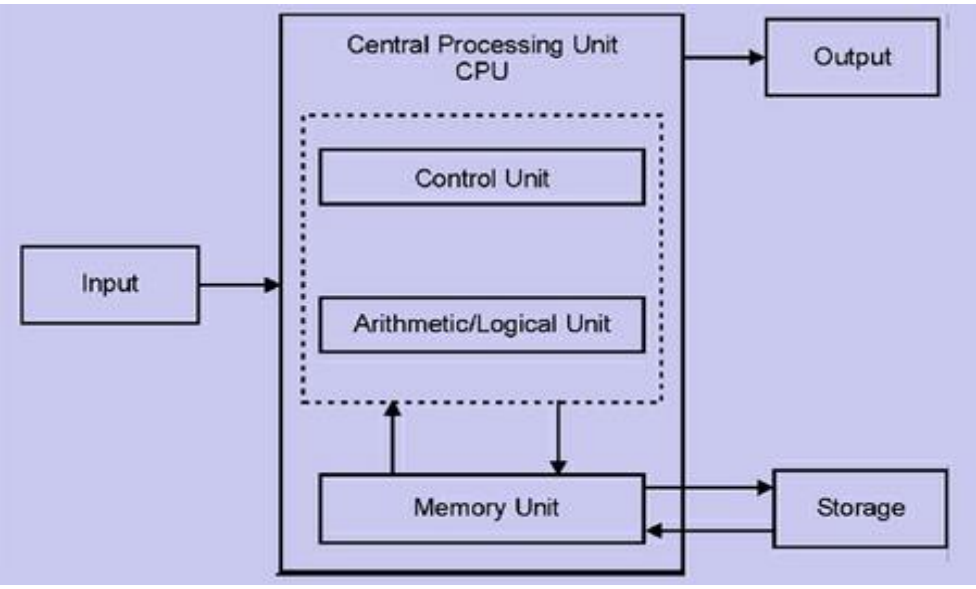
Software:

The programs that run on a computer are referred to as software.

Moore's Law:

Moore's law refers to an observation made by Intel co-founder Gordon Moore in 1965. He noticed that the number of transistors per square inch on integrated circuits had doubled every year since their invention. Moore's law predicts that this trend will continue into the foreseeable future.

Computer Organization



Computer Languages:

Following are the programming languages used by the programmers.

- *Machine Language*
- *Assembly Language*
- *High-Level Languages*

1.2- Object Oriented Programming & Designing

Object-oriented programming (OOP) is a programming language model organized around objects rather than "actions" and data rather than logic. Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data.

Objects:

Object refers to a particular instance of a class where the object can be a combination of variables, functions, and data structures. An object is referred to the instance of the class.

Methods & Class:

A function is independent and not associated with a class. Object-oriented programming uses a number of core concepts: abstraction, encapsulation, inheritance and polymorphism. These concepts are implemented using classes, objects and methods.

Attributes and Instance Variables:

In object-oriented programming with classes, an instance variable is a variable defined in a class (i.e. a member variable), for which each instantiated object of the class has a separate copy, or instance. An instance variable is similar to a class variable. Variables are properties an object knows about itself.

Encapsulation:

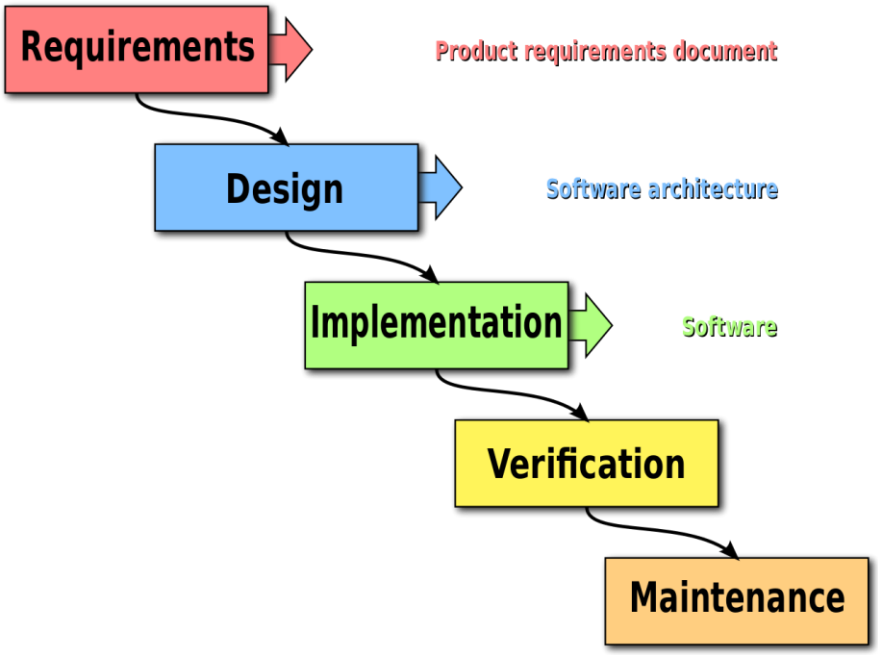
Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse (Data Hiding).

Inheritance:

In object-oriented programming, inheritance is when an object or class is based on another object or class, using the same implementation or specifying a new implementation to maintain the same behavior.

1.3- Object-oriented analysis and design (OOAD):

Object-oriented analysis and design (OOAD) is a popular technical approach for analyzing, designing an application, system, or business by applying the object-oriented paradigm and visual modeling throughout the development life cycles to foster better stakeholder communication and product quality.



1.4- The UML (Unified Modeling Language):

The Unified Modeling Language (UML) is a general-purpose, developmental, modeling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system.

CHAPTER 2

Introduction to Java Applications

Objectives

In this chapter we'll learn

- To write simple Java applications.
- To use input and output statements.
- Java's primitive types.
- To use arithmetic operators.
- The precedence of arithmetic operators
- To write decision-making statements.
- To use relational and equality operators

2.1- Your First Program in Java: Printing a Line of Text

Commenting Your Program:

We insert comments to document programs and improve their readability. The Java compiler ignores comments, so they do *not* cause the computer to perform any action when the program is run.

The commenting of the program line is done by using '//'.For example

```
// Text-printing program.
```

Declaring a Class:

Every Java program consists of at least one class that you (the programmer) define. The **class** keyword introduces a class declaration and is immediately followed by the class name.eg.

```
public class Welcome1
```

Declaring a Method:

public static void main (String[ ] args) is the starting point of every Java application. The parentheses after the identifier main indicate that it's a program building block called a method. Java class declarations normally contain one or more methods.

*Performing Output with **System.out.println:***

System.out.println ("Welcome to Java Programming!") Instructs the computer to perform an action—namely, to print the string of characters contained between the double quotation marks (but not the quotation marks themselves).

Implementation:

```
// Text-printing program.
public class Welcome1
 {
 // main method begins execution of Java application
 public static void main( String[] args )
 {
 System.out.println( "Welcome to Java Programming!" );
 } // end method main
 } // end class Welcome1
```

Output: Welcome to Java Programming!

2.2- Modifying Your First Program in Java

Displaying a Single Line of Text with Multiple Statements:

Implementation:

```
// Text-printing program.
public class Welcome1
 {
 // main method begins execution of Java application
 public static void main( String[] args )
 {
 System.out.print( "Welcome to " );
System.out.println( "Java Programming!" )
} // end method main
 } // end class Welcome1
```

Output: Welcome to Java Programming!

2.3- Displaying Text with printf:

Displaying Multiple Lines of Text with a Single Statement:

```
// Text-printing program.
public class Welcome1
 {
 // main method begins execution of Java application
 public static void main( String[] args )
 {
 System.out.println( "Welcome\n to\n Java\n Programming!" );
} // end method main
 } // end class Welcome1
```

Output:

```
Welcome
 to
 Java
 Programming!
```

```
System.out.printf ("%s\n%s\n","Welcome to", "Java Programming!");
```

```
// Displaying multiple lines with method System.out.printf.
public class Welcome4
{
// main method begins execution of Java application
public static void main( String[] args )
{
System.out.printf( "%s\n%s\n","Welcome to", "Java Programming!" );

} // end method main
} // end class Welcome4
```

Output:

```
Welcome to
Java Programming!
```

2.4- Another Application (Adding Integers):

Declaring & Creating a Scanner to Obtain User Input from the Keyboard:

A variable is a location in the computer's memory where a value can be stored for use later in a program. All Java variables *must* be declared with a name and a type *before* they can be used. A variable's name enables the program to access the value of the variable in memory. A variable's name can be any valid identifier.

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

is a variable declaration statement that specifies the name (input) and type (Scanner) of a variable that's used in this program. A **Scanner** enables a program to read data (e.g., numbers and strings) for use in a program. The data can come from many sources, such as the user at the keyboard or a file on disk. Before using a Scanner, you must create it and specify the source of the data.

Using Variables in a Calculation:

sum = number1 + number2; // add numbers then store total in sum
is an assignment statement that calculates the sum of the variables number1 and number2 then assigns the result to variable sum by using the assignment operator, =. The statement is read

as "sum *gets* the value of number1 + number2."

Implementation:

```
// Addition program that displays the sum of two numbers.
import java.util.Scanner;               // program uses class Scanner
public class Addition
 {
                    // main method begins execution of Java application
 public static void main( String[] args )
{
                  // create a Scanner to obtain input from the command window
Scanner input = new Scanner( System.in );
int number1;             // first number to add
int number2;            // second number to add
int sum;                // sum of number1 and number2
System.out.print( "Enter first integer: " );         // prompt
number1 = input.nextInt();              // read first number from user
System.out.print( "Enter second integer: " );          // prompt
number2 = input.nextInt();                 // read second number from user
sum = number1 + number2;                // add numbers, then store total in sum
System.out.printf( "Sum is %d\n", sum );          // display sum
}                                                  // end method main
 }                                                // end class Addition
```

Output:
Enter first integer: **45**
Enter second integer: **72**
Sum is 117

2.5 **Airthematic Operators**

Java operation	Operator	Algebraic expression	Java expression
Addition	+	$f + 7$	`f + 7`
Subtraction	–	$p - c$	`p - c`
Multiplication	*	bm	`b * m`
Division	/	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Remainder	%	$r \bmod s$	`r % s`

Rules of Operator Precedence:

Following are the rules of operator precedence.

1. Multiplication, division and remainder operations are applied first. If an expression contains several such operations, they're applied from left to right. Multiplication, division and remainder operators have the same level of precedence.
2. Addition and subtraction operations are applied next. If an expression contains several such operations, the operators are applied from left to right. Addition and subtraction operators have the same level of precedence.

Decision Making Equality and Relational Operators:

Standard algebraic equality or relational operator	Java equality or relational operator	Sample Java condition	Meaning of Java condition
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y

Implementation:

```
// Compare integers using if statements, relational operators
// and equality operators.
import java.util.Scanner;              // program uses class Scanner
public class Comparison
{
                    // main method begins execution of Java application
public static void main( String[] args )
{
                  // create Scanner to obtain input from command line
Scanner input = new Scanner( System.in );
int number1;              // first number to compare
int number2;              // second number to compare
System.out.print( "Enter first integer: " );          // prompt
number1 = input.nextInt();            // read first number from user
```

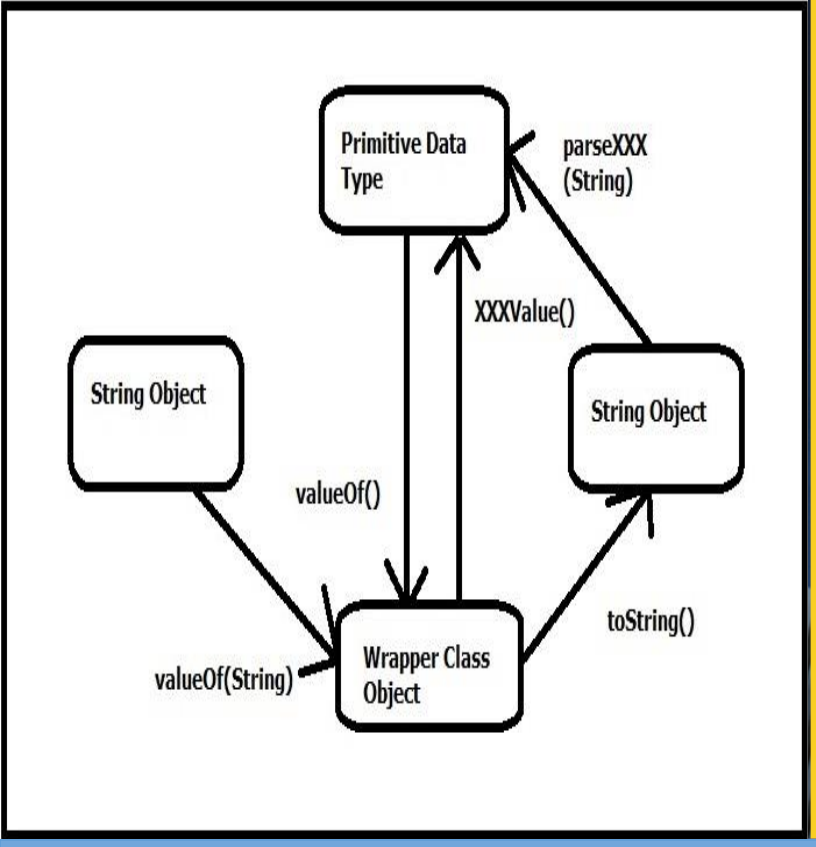
```
 System.out.print( "Enter second integer: " );          // prompt
 number2 = input.nextInt();              // read second number from user
if ( number1 == number2 )
System.out.printf( "%d == %d\n", number1, number2 );
if ( number1 != number2 )
System.out.printf( "%d != %d\n", number1, number2 );
if ( number1 < number2 )
System.out.printf( "%d < %d\n", number1, number2 );
if ( number1 > number2 )
System.out.printf( "%d > %d\n", number1, number2 );
if ( number1 <= number2 )
System.out.printf( "%d <= %d\n", number1, number2 );
if ( number1 >= number2 )
System.out.printf( "%d >= %d\n", number1, number2 );
} // end method main
} // end class Comparison
```

Output:

Enter first integer: **777**
Enter second integer: **777**
777 == 777
777 <= 777
777 >= 777

Precedence and associativity of operators discussed

Operators	Associativity	Type
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
=	right to left	assignment



CHAPTER 3

Introduction to Classes, Objects, Methods & String

Objectives

In this chapter we'll

- How to create a class and use it to create an object.
- How to implement a class's behaviors as methods.
- How to call an object's methods to make them perform their tasks.
- What instance variables of a class and local variables of a method are.
- How to use a constructor to initialize an object's data.
- The differences between primitive and reference types
- How to implement a class's attributes as instance variables and properties.

3.1 Declaring a Class with a Method and Instantiating an Object of a Class

Creating a Class:

Create a class having name GradeBook that simply shows the message "Welcome to the Grade Book".

Class GradeBook

```
// Class declaration with one method.

public class GradeBook
 {
// display a welcome message to the GradeBook user
 public void displayMessage()
 {
 System.out.println( "Welcome to the Grade Book!" );} // end method displayMessage
 } // end class GradeBook
```

Creating a Main Class:

Create a main class having name GradeBookTest. Then create an object of class GradeBook and call the member function of the class using that object.

Main Class GradeBookTest

```
public class GradeBookTest
 {
 // main method begins program execution
 public static void main( String[] args )
 {
 // create a GradeBook object and assign it to myGradeBook
GradeBook myGradeBook = new GradeBook();
 // call myGradeBook's displayMessage method
myGradeBook .displayMessage();
} // end main
} //
```

Output:

```
Welcome to the Grade Book!
```

3.2 Declaring a Method with a Parameter

Arguments to a Method:

A method call supplies values—called *arguments*—for each of the method's parameters. For example, the method System.out.println requires an argument that specifies the data to output in a command window.

Class GradeBook

```
 // Class declaration with one method that has a parameter.
public class GradeBook
 {
 // display a welcome message to the GradeBook user
 public void displayMessage( )
 {
System.out.printf( "Welcome to the grade book for\n%s!\n",
courseName );
 } // end method displayMessage
 } // end class GradeBook
```

Main Class GradeBookTest

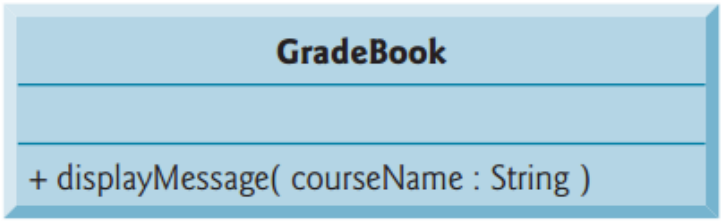
```
// GradeBookTest.java
 // Create GradeBook object and pass a String to
// its displayMessage method.
 import java.util.Scanner; // program uses Scanner
public class GradeBookTest
 {
 // main method begins program execution
 public static void main( String[] args )
 {
 // create Scanner to obtain input from command window
 Scanner input = new Scanner( System.in );
 // create a GradeBook object and assign it to myGradeBook
 GradeBook myGradeBook = new GradeBook();
 // prompt for and input course name
 System.out.println( "Please enter the course name:" );
 String nameOfCourse = input.nextLine(); // read a line of text System.out.println(); // outputs
a blank line

 // call myGradeBook's displayMessage method
```

```
 // and pass nameOfCourse as an argument
 myGradeBook.displayMessage( nameOfCourse );
 } // end main
} // end class GradeBookTest
```

Output:

```
Please enter the course name:
CS101 Introduction to Java Programming
Welcome to the grade book for
CS101 Introduction to Java Programming!
```

UML Diagram of Class GradeBook:



3.3 Instance Variables, set Methods and get Methods

Class GradeBook

```
public class GradeBook
 {
private String courseName; // course name for this GradeBook
// method to set the course name
public void setCourseName( String name )
{
courseName = name; // store the course name
} // end method setCourseName
// method to retrieve the course name
public String getCourseName()
{
return courseName;
} // end method getCourseName
```

```
// display a welcome message to the GradeBook user
public void displayMessage
 {
 // calls getCourseName to get the name of
 // the course this GradeBook represents
 System.out.printf( "Welcome to the grade book for\n%s!\n");
 } // end method displayMessage
 } // end class GradeBook
```

Main Class GradeBook

```
.// Creating and manipulating a GradeBook object.
 import java.util.Scanner; // program uses Scanner
public class GradeBookTest
 {
 // ma Chapter#3: Introduction to Classes, Objects, Methods & String
 public static void main( String[] args )
 {
 // create Scanner to obtain input from command window
 Scanner input = new Scanner( System.in );

 // create a GradeBook object and assign it to myGradeBook
 GradeBook myGradeBook = new GradeBook();
// display initial value of courseName
System.out.printf( "Initial course name is: %s\n\n" );
// prompt for and read course name
 System.out.println( "Please enter the course name:" );
 String theName = input.nextLine(); // read a line of text
 myGradeBook.setCourseName( theName ); // set the course name

 System.out.println(); // outputs a blank line

// display welcome message after specifying course name
} // end main
 } // end class GradeBookTest
```

Output:

Initial course name is: null
Please enter the course name:
CS101 Introduction to Java Programming
Welcome to the grade book for
CS101 Introduction to Java Programming!

3.4 (Optional) GUI and Graphics Case Study: Using Dialog Boxes

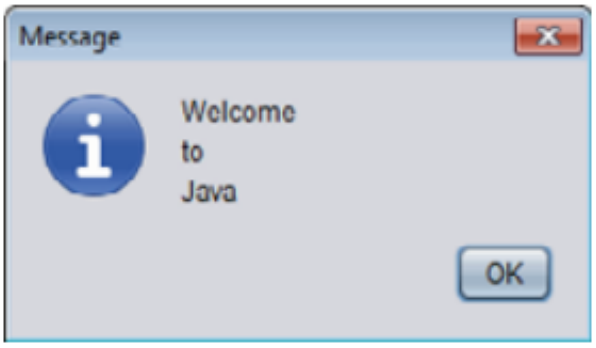
Displaying Text in a Dialog Box:

The programs presented thus far display output in the command window. Many applications use windows or dialog boxes (also called dialogs) to display output. Typically, dialog boxes are windows in which programs display important messages to users. Class **JOptionPane** provides prebuilt dialog boxes that enable programs to display windows containing messages—such windows are called message dialogs.

Displaying Text in a Dialog Box:

```
// Fig. 3.17: Dialog1.java
 // Using JOptionPane to display multiple lines in a dialog box.
 import javax.swing.JOptionPane; // import class JOptionPane

public class Dialog1
 {
7 public static void main( String[] args )
 {
 // display a dialog with a message
 JOptionPane.showMessageDialog( null, "Welcome\nto\nJava" );
} // end main
 } // end class Dialog1
```

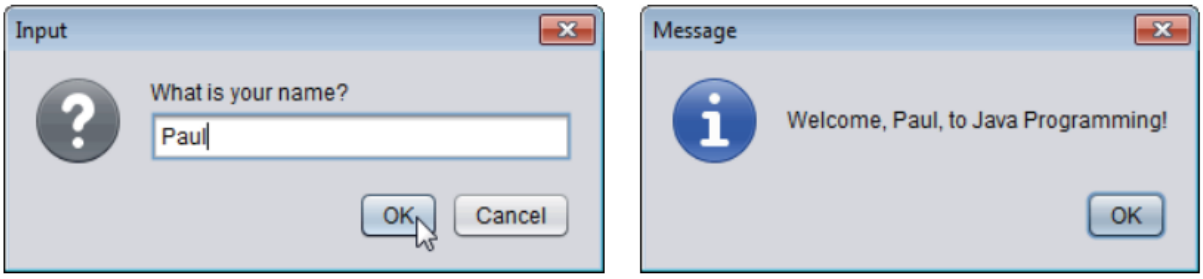
Output:



Entering Text in a Dialog:

```
 // Basic input with a dialog box.
 import javax.swing.JOptionPane;
public class NameDialog
 {
 public static void main( String[] args )
{
 // prompt user to enter name
String name =
JOptionPane.showInputDialog( "What is your name?" );
String message =
String.format( "Welcome, %s, to Java Programming!", name );
// display the message to welcome the user by name
 JOptionPane.showMessageDialog( null, message );
 } // end main
} // end class NameDialog
```

Output:



Control Statements In Java

CHAPTER 4

Control Statements: Part 1

Objectives

In this chapter you'll learn:

- Basic problem-solving techniques.
- To develop algorithms through the process of top down, stepwise refinement.
- To use the `if` and `if…else` selection statements to choose among alternative actions.
- To use the `while` repetition statement to execute statements in a program repeatedly.
- To use counter-controlled repetition and sentinel controlled repetition.
- To use the compound assignment, increment and decrement operators.
- The portability of primitive data types.

Introduction:

Before writing a program to solve a problem, you should have a thorough understanding of the problem and a carefully planned approach to solving it.

Algorithms:

Any computing problem can be solved by executing a series of actions in a specific order.
A procedure for solving a problem in terms of
1. The actions to execute and
2. The order in which these actions execute

is called an algorithm.

Pseudo code:

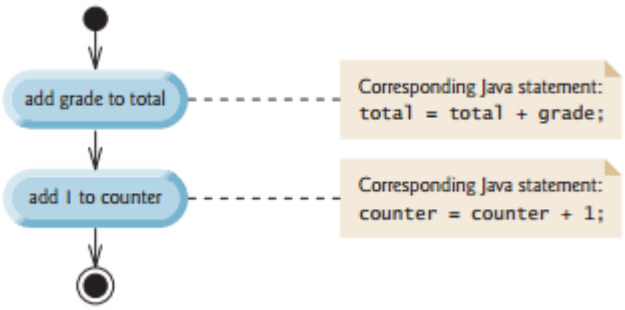
Pseudo code is an informal language that helps you develop algorithms without having to worry about the strict details of Java language syntax.

Pseudo code is similar to everyday English—it's convenient and user friendly.

Control structures:

Normally, statements in a program are executed one after the other in the order in which they're written. This process is called sequential execution.

The term structured programming became almost synonymous with "goto elimination." [*Note:* Java does *not* have a statement; however, the word is *reserved* by Java and should *not* be used as an identifier in programs.]



A UML activity diagram models the workflow (also called the activity) of a portion of a software system. Such workflows may include a portion of an algorithm, like the sequence structure in Fig. 4.1. Activity diagrams are composed of symbols, such as action state symbols (rectangles with their left and right sides replaced with outward arcs), diamonds and small circles. These symbols

Chapter#4: Control Statements: Part I

the flow of the activity—that is, the order in which th

Selection Statements in Java:

Java has three types of selection statements single-selection statement because it selects or ignores a single action double-selection statement because it selects between two different actionsmultiple-selection statement because it selects
among many different actions

Repetition Statements in Java:

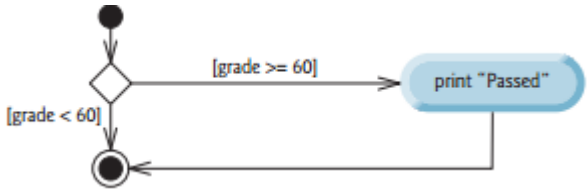
Java provides three repetition statements (also called looping statements) that enable programs to perform statements repeatedly as long as a condition (called the loop-continuation condition) remains true.

If single selection statement:

Programs use selection statements to choose among alternative courses of action. For example, suppose that the passing grade on an exam is 60. The pseudocode statement

```
If student's grade is greater than or equal to 60
    Print "Passed"
```

```
if ( studentGrade >= 60 )
System.out.println( "Passed" );
```

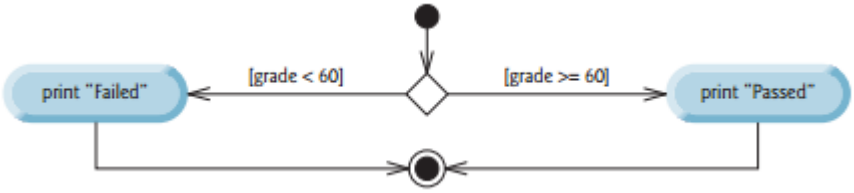


If…else double Selection statement:

The if single-selection statement performs an indicated action only when the condition is true; otherwise, the action is skipped. The **if…else** double-selection statement allows you to specify an action to perform when the condition is true and a different action when the condition is false. For example, the pseudocode statement

```
If student's grade is greater than or equal to 60
    Print "Passed"
Else
    Print "Failed"
```

```
if ( grade >= 60 )
System.out.println( "Passed" );
else
System.out.println( "Failed" );
```

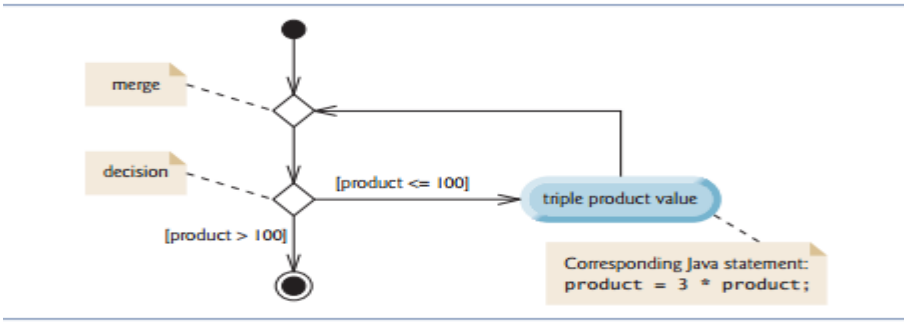


While Repetition Statement:

A repetition (or looping) statement allows you to specify that a program should repeat an action while some condition remains true. The pseudocode statement

While there are more items on my shopping list
Purchase next item and cross it off my list

```
while ( product <= 100 )
product = 3 * product;
```



4.8 Formulating Algorithms: Counter-Controlled Repetition:

```
 import java.util.Scanner;
public class GradeBook
 private String courseName;
public GradeBook( String name )
 courseName = name; // initializes courseName
1 public void setCourseName( String name )
courseName = name; // store the course name
public String getCourseName()
 return courseName;}
public void displayMessage()
{System.out.printf( "Welcome to the grade book for\n%s!\n\n",
 getCourseName() );
} // end method displayMessage
{ Scanner input = new Scanner( System.in );
 int total; // sum of grades entered by user
int grade; // grade value entered by user
int average; // average of grades
 total = 0;
while ( ) // loop 10 times
 {System.out.print( "Enter grade: " ); // prompt
 grade = input.nextInt(); // input next grade
 total = total + grade; }
```

```
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100
Total of all 10 grades is 846
Class average is 84
```

4.9 Formulating Algorithms: Sentinel-Controlled Repetition:

Developing the Pseudocode Algorithm with Top-Down, Stepwise Refinement:
The Top and First Refinement.

Initialize variables

Initialize total to zero
Initialize counter to zero

Input, sum and count the quiz grades

Prompt the user to enter the first grade
Input the first grade (possibly the sentinel)
While the user has not yet entered the sentinel
Add this grade into the running total
Add one to the grade counter
Prompt the user to enter the next grade
Input the next grade (possibly the sentinel

Calculate and print the class average

If the counter is not equal to zero
Set the average to the total divided by the counter
Print the average
else
Print "No grades were entered")

Implementing Sentinel-Controlled Repetition in Class **GradeBook**

Figure 4.9 shows the Java class Grade Book containing method determine Class Average that implements the pseudocode algorithm of Fig. 4.8. Although each grade is an integer, the averaging calculation is likely to produce a number with a decimal point—in other words, a real (i.e., floating-point) number. The type `int` cannot represent such a number, so this class uses type `double` to do so.

```
import java.util.Scanner; // program uses class Scanner
public class GradeBook
 { private String courseName; // name of course this GradeBook represents

 public GradeBook( String name )
 { courseName = name; // initializes courseName
 public void setCourseName( String name )
 { courseName = name; // store the course name
 public String getCourseName()
 {return courseName;
 public void displayMessage()

{ System.out.printf( "Welcome to the grade book for\n%s!\n\n",
 getCourseName() );

public void determineClassAverage(){
 Scanner input = new Scanner( System.in );
 int total; // sum of grades
 int gradeCounter; // number of grades entered
 int grade; // grade value

double average;
 total = 0;

gradeCounter = 0; // initialize loop counter

System.out.print( "Enter grade or -1 to quit: " );
grade = input.nextInt();while ( grade != -1 )
 {total = total + grade; // add grade to total
 gradeCounter = gradeCounter + 1; // increment counter
System.out.print( "Enter grade or -1 to quit: " );
grade = input.nextInt();}
```

Output

Enter grade or -1 to quit: **97**
Enter grade or -1 to quit: **88**
Enter grade or -1 to quit: **72**
Enter grade or -1 to quit: **-1**

4.10 Formulating Algorithms: Nested Control Statements

```
import java.util.Scanner;
public class Analysis
 {
 public static void main( String[] args )
{ Scanner input = new Scanner( System.in );
int passes = 0; // number of passes
int failures = 0; // number of failures
int studentCounter = 1;

int result; // one exam result (obtains value from user)
while ( studentCounter <= 10 )
{ System.out.print( "Enter result (1 = pass, 2 = fail): " );
 result = input.nextInt();

if ( result == 1 ) // if result 1,
passes = passes + 1; // increment passes;
else // else result is not 1, so
failures = failures + 1;

studentCounter = studentCounter + 1; }

System.out.printf( "Passed: %d\nFailed: %d\n", passes, failures );

if ( passes > 8 )
System.out.println( "Bonus to instructor!" );}}
```

Output:

Enter result (1 = pass, 2 = fail): **1**
Enter result (1 = pass, 2 = fail): **2**
Enter result (1 = pass, 2 = fail): **1**
Enter result (1 = pass, 2 = fail): **2**
Enter result (1 = pass, 2 = fail): **1**
Enter result (1 = pass, 2 = fail): **2**
Enter result (1 = pass, 2 = fail): **2**
Enter result (1 = pass, 2 = fail): **1**
Enter result (1 = pass, 2 = fail): **1**
Enter result (1 = pass, 2 = fail): **1**
Passed: 6
Failed: 4

4.11 Compound Assignment Operators:

The compound assignment operators abbreviate assignment expressions. Statements like variable = variable operator expression; where *operator* is one of the binary operators +, -, *, / or % (or others we discuss later in the text) can be written in the form

c=c+3

For example, you can abbreviate the statement with the addition compound assignment operator, +=, as

C+=3

The += operator adds the value of the expression on its right to the value of the variable on its left and stores the result in the variable on the left of the operator. Thus, the assignment expression c += 3 adds 3 to c. Figure 4.13 shows the arithmetic compound assignment operators, sample expressions using the operators and explanations of what the operators do.

4.12 Increment and Decrement Operators:

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to c
`-=`	`d -= 4`	`d = d - 4`	1 to d
`*=`	`e *= 5`	`e = e * 5`	20 to e
`/=`	`f /= 3`	`f = f / 3`	2 to f
`%=`	`g %= 9`	`g = g % 9`	3 to g

Operator	Operator name	Sample expression	Explanation
++	prefix increment	++a	Increment a by 1, then use the new value of a in the expression in which a resides.
++	postfix increment	a++	Use the current value of a in the expression in which a resides, then increment a by 1.
--	prefix decrement	--b	Decrement b by 1, then use the new value of b in the expression in which b resides.
--	postfix decrement	b--	Use the current value of b in the expression in which b resides, then decrement b by 1.

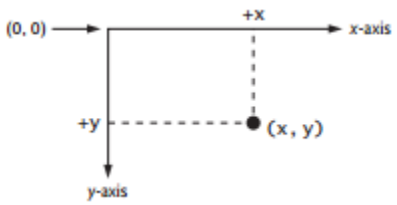
```
public class Increment
 { public static void main( String[] args )
{ int c;
 c = 5;
 System.out.println( c );

System.out.println( c++ );
System.out.println( c );

System.out.println();

System.out.println( ++c );
System.out.println( c );}}
```

Output:

Operators						Associativity	Type
++	--					right to left	unary postfix
++	--	+	-	(*type*)		right to left	unary prefix
*	/	%				left to right	multiplicative
+	-					left to right	additive
<	<=	>	>=			left to right	relational
==	!=					left to right	equality
?:						right to left	conditional
=	+=	-=	*=	/=	%=	right to left	assignment

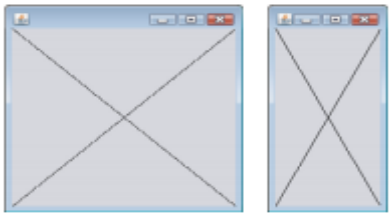
(optional) GUI and graphic case study: cresting simple drawings:

An appealing feature of Java is its graphics support, which enables you to visually enhance your applications. We now introduce one of Java's graphical capabilities—drawing lines. It also covers the basics of creating a window to display a drawing on the computer screen.



```
import java.awt.Graphics;
import javax.swing.JPanel;

public class DrawPanel extends JPanel

{public void paintComponent( Graphics g )

{super.paintComponent( g );

int width = getWidth();
int height = getHeight();

g.drawLine( 0, 0, width, height );

g.drawLine( 0, height, width, 0 );

}

}
```

Output:



G

CHAPTER 5

Control Statement: Part 2

Objectives

In this chapter you'll learn:

- The essentials of counter controlled repetition.
- To use the `for` and `do…while` repetition statements to execute statements in a program repeatedly.
- To understand multiple selection using the `switch` selection statement.
- To use the `break` and `continue` program control statements to alter the flow of control.
- To use the logical operators to form complex conditional expressions in control statements

.

5.1 Introduction:

This chapter continues our presentation of structured programming theory and principles by introducing all but one of Java's remaining control statements. We demonstrate Java's for, do…while and switch statements.

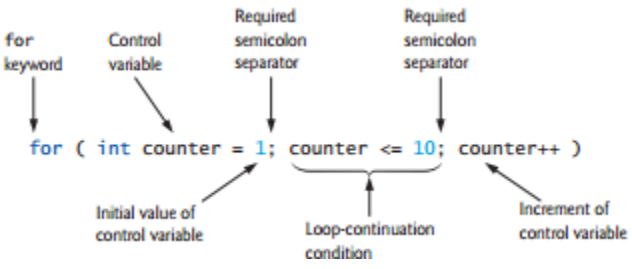
5.2 Essentials of Counter-Controlled Repetition:

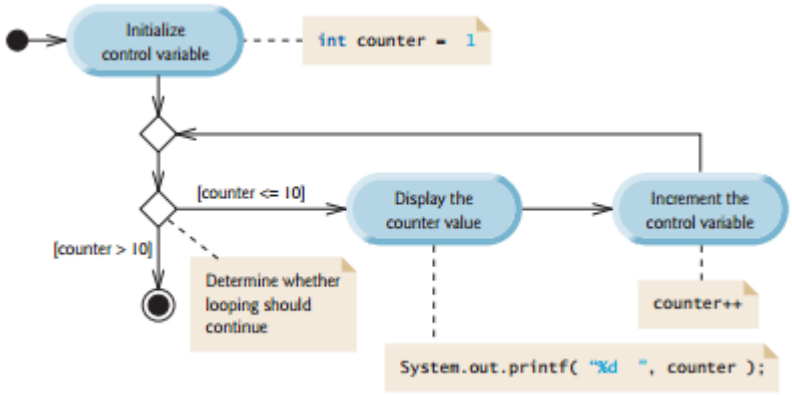
This section uses the while repetition statement introduced in Chapter 4 to formalize the elements required to perform counter-controlled repetition, which requires

1. a **control variable** (or loop counter)
2. the **initial value** of the control variable
3. the **increment** (or **decrement**) by which the control variable is modified each time through the loop (also known as **each iteration of the loop**)
4. the **loop-continuation condition** that determines if looping should continue.

```
public class WhileCounter

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

int counter = 1;

while(counter<=10)

System.out.printf( "%d ", counter );

++counter;

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

```
1  2  3  4  5  6  7  8  9  10
```





5.4 Examples Using the for Statement:

The following examples show techniques for varying the control variable in a for statement. In each case, we write the appropriate for header. Note the change in the relational operator for loops that decrement the control variable.

a) Vary the control variable from 1 to 100 in increments of 1.

```
for ( int i = 1; i <= 100; i++ )
```

b) Vary the control variable from 100 to 1 in decrements of 1.

```
for ( int i = 100; i >= 1; i-- )
```

c) Vary the control variable from 7 to 77 in increments of 7.

```
for ( int i = 7; i <= 77; i += 7 )
```

d) Vary the control variable from 20 to 2 in decrements of 2.

```
for ( int i = 20; i >= 2; i -= 2 )
```

e) Vary the control variable over the values 2, 5, 8, 11, 14, 17, 20.

```
for ( int i = 2; i <= 20; i += 3 )
```

f) Vary the control variable over the values 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

```
for ( int i = 99; i >= 0; i -= 11 )
```

```
public class Sum
 {
 public static void main( String[] args )
{
int total = 0;

for ( int number = 2; number <= 20; number += 2 )
total += number;

System.out.printf( "Sum is %d\n", total );}}

public static void main( String[] args )
{
8 double amount; // amount on deposit at end of each year
9 double principal = 1000.0; // initial amount before interest
 double rate = 0.05; // interest rate
13 System.out.printf( "%s \n", "Year", "Amount on deposit" );
for ( int year = 1; year <= 10; year++ )
{
amount = principal * Math.pow( 1.0 + rate, year );
System.out.printf( "%4d%,20.2f\n", year, amount );
} /}}
```

Output:

```
Year Amount on deposit
1 1,050.00
2 1,102.50
3 1,157.63
4 1,215.51
5 1,276.28
6 1,340.10
7 1,407.10
8 1,477.46
9 1,551.33
10 1,628.89
```

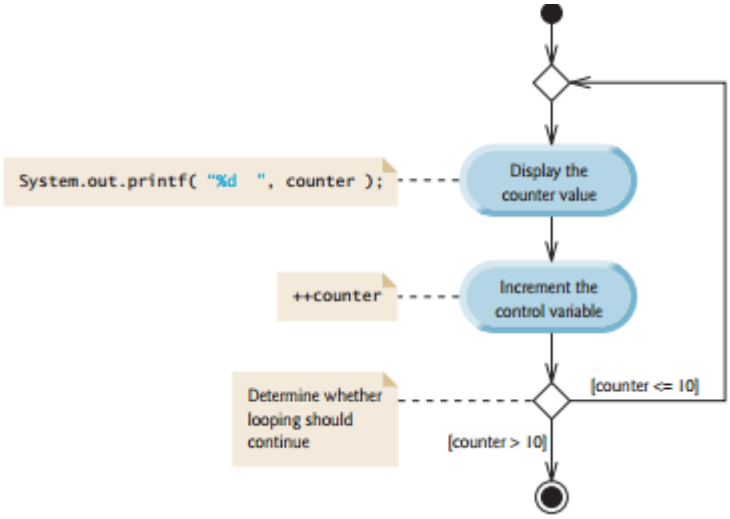
5.5 do…while Repetition Statement:

The **do…while repetition statement** is similar to the while statement. In the while, the program tests the loop-continuation condition at the beginning of the loop, before executing the loop's body; if the condition is false, the body *never* executes.

```
public class DoWhileTest
5 {
6 public static void main( String[] args )
7 { int counter = 1;
{
System.out.printf( "%d ", counter );
++counter;
} while ( counter <= 10 );

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

Output:

```
1 2 3 4 5 6 7 8 9 10
```



5.6 switch Multiple-Selection Statement:

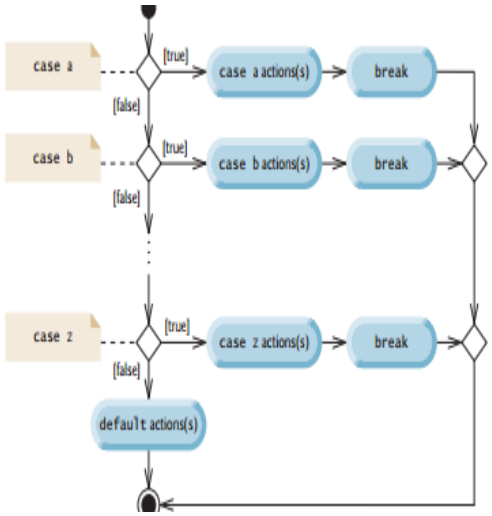
in this example to determine which counter to increment based on the grade. When the flow of control reaches the switch, the program evaluates the expression in the parentheses (grade / 10) following keyword switch. This is the switch's **controlling expression**. The program compares this expression's value (which must evaluate to an integral value of type byte, char, short or int) with each case label.

```
public class GradeBookTest
 {
 public static void main( String[] args )
 {
 GradeBook myGradeBook = new GradeBook(
 "CS101 Introduction to Java Programming" );
 myGradeBook.displayMessage(); // display welcome message

myGradeBook.inputGrades(); // read grades from user
myGradeBook.displayGradeReport();

}

}
```

Output:

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

Enter the integer grades in the range 0-100.
Type the end-of-file indicator to terminate input:
   On UNIX/Linux/Mac OS X type <Ctrl> d then press Enter
   On Windows type <Ctrl> z then press Enter
99
92
45
57
63
71
76
85
90
100
^Z

Grade Report:
Total of the 10 grades entered is 778
Class average is 77.80
```

UML diagram:



Logical Operators:

Java's **logical operators** enable you to form more complex conditions by *combining* simple conditions. The logical operators are && (conditional AND), || (conditional OR), & (boolean logical AND), | (boolean logical inclusive OR), ^ (boolean logical exclusive OR) and ! (logical NOT). [*Note:* The &, | and ^ operators are also bitwise operators when they're applied to integral operands. We discuss the bitwise operators in Appendix N.]

expression1	expression2	expression1 || expression2
false	false	false
false	true	true
true	false	true
true	true	true

```
public class LogicalOperators
{
 public static void main( String[] args )
 {
 System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
 "Conditional AND (&&)", "false && false", ,
 "false && true", ,
 "true && false", ,
"true && true", );
 System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n","Conditional OR (||)", "false || false", ,
"false || true", , "true || false", ,"true || true" );
 System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n","Boolean logical AND (&)", "false &
false", "false & true", "true & false");
}}
```

Output:

```
false && false: false
false && true: false
true && false: false
true && true: true

Conditional OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Boolean logical AND (&)
false & false: false
false & true: false
true & false: false
true & true: true

Boolean logical inclusive OR (|)
false | false: false
false | true: true
true | false: true
true | true: true

Boolean logical exclusive OR (^)
false ^ false: false
false ^ true: true
true ^ false: true
true ^ true: false

Logical NOT (!)
!false: true
!true: false
```

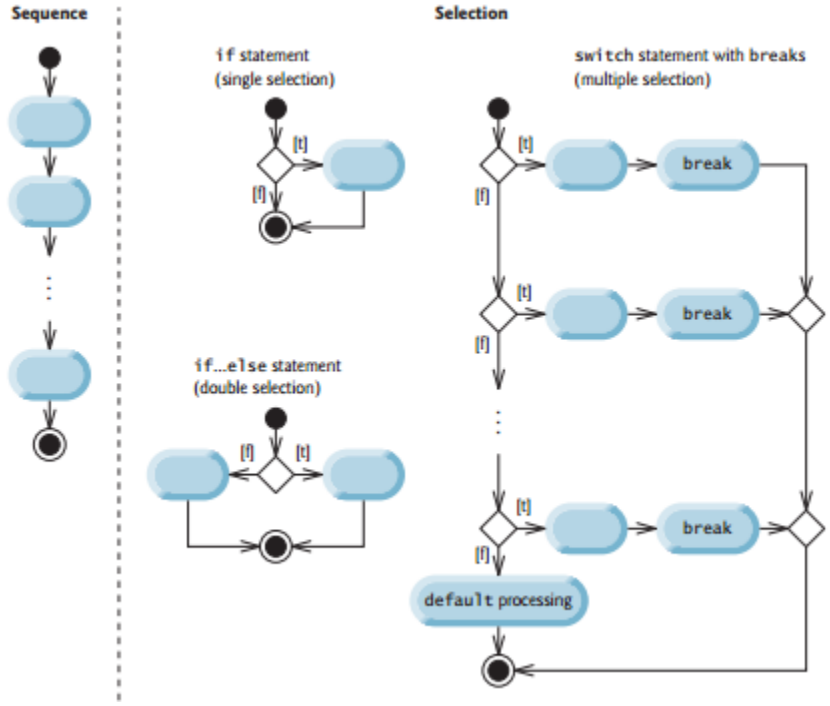
Operations:

Operators	Associativity	Type
++ --	right to left	unary postfix
++ -- + - ! (*type*)	right to left	unary prefix
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&	left to right	boolean logical AND
^	left to right	boolean logical exclusive OR
|	left to right	boolean logical inclusive OR
&&	left to right	conditional AND
||	left to right	conditional OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

Operators	Associativity	Type
++ --	right to left	unary postfix
++ -- + - ! (*type*)	right to left	unary prefix
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&	left to right	boolean logical AND
^	left to right	boolean logical exclusive OR
|	left to right	boolean logical inclusive OR
&&	left to right	conditional AND
||	left to right	conditional OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

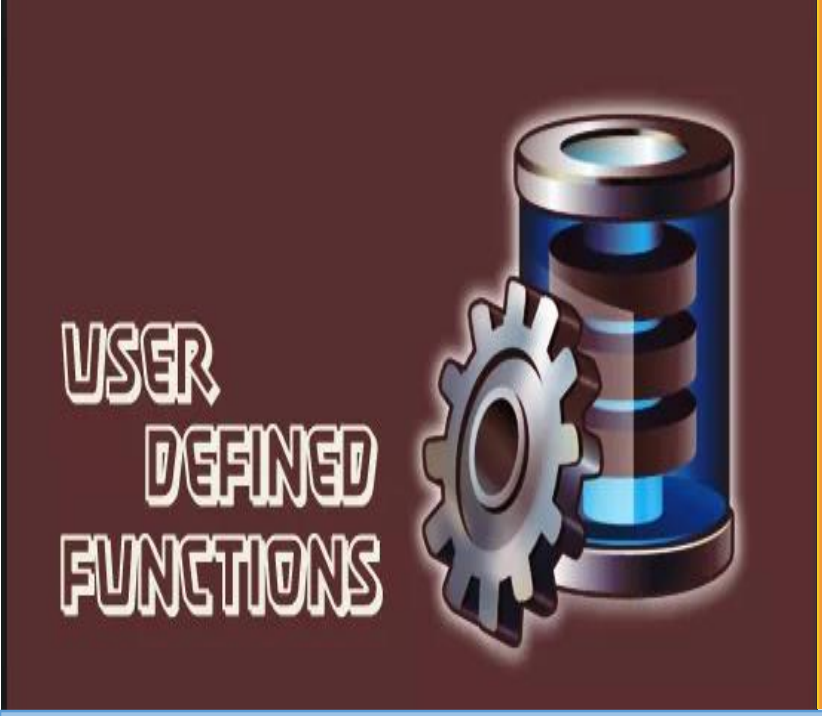
5.9 Structured Programming Summary:

Just as architects design buildings by employing the collective wisdom of their profession, so should programmers design programs. Our field is much younger than architecture, and our collective wisdom is considerably sparser. We've learned that structured programming produces programs that are easier than unstructured programs to understand, test, debug, modify and even prove correct in a mathematical sense



5.11 Wrap-Up:

In this chapter, we completed our introduction to Java's control statements, which enable you to control the flow of execution in methods.



CHAPTER 6

User Defined Functions

Objectives

In this chapter you'll learn:

- How static methods and fields are associated with classes rather than objects.
- How the method call/return mechanism is supported by the method-call stack.
- How packages group related classes.
- How to use random-number generation to implement game-playing applications.
- How the visibility of declarations is limited to specific regions of programs.
- What method overloading is and how to create overloaded methods.

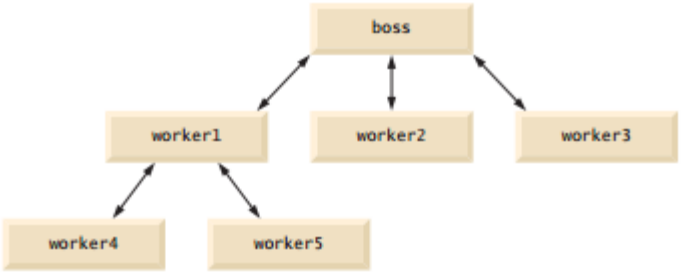
Chapter#6: User Defined Functions

6.1 Introduction:

Experience has shown that the best way to develop and maintain a large program is to construct it from small, simple pieces, or modules. This technique is called divide and conquer.

6.2 Program Modules in Java:

You write Java programs by combining new methods and classes with predefined ones available in the Java Application Programming Interface (also referred to as the Java API or Java class library) and in various other class libraries. Related classes are typically grouped into packages so that they can be imported into programs and reused



6.3 static Methods, static Fields and Class Math:

Although most methods execute in response to method calls on specific objects, this is not always the case. Sometimes a method performs a task that does not depend on the contents of any object. Such a method applies to the class in which it's declared as a whole and is known as a static method or aclassmethod.

ClassName.methodName(arguments)

```
Math.sqrt( 900.0 )

System.out.println( Math.sqrt( 900.0 ) );

System.out.println( Math.sqrt( c + d * f ) );
```

Method	Description	Example
`abs( x )`	absolute value of x	`abs( 23.7 )` is 23.7 `abs( 0.0 )` is 0.0 `abs( -23.7 )` is 23.7
`ceil( x )`	rounds x to the smallest integer not less than x	`ceil( 9.2 )` is 10.0 `ceil( -9.8 )` is -9.0
`cos( x )`	trigonometric cosine of x (x in radians)	`cos( 0.0 )` is 1.0
`exp( x )`	exponential method e^x	`exp( 1.0 )` is 2.71828 `exp( 2.0 )` is 7.38906
`floor( x )`	rounds x to the largest integer not greater than x	`floor( 9.2 )` is 9.0 `floor( -9.8 )` is -10.0
`log( x )`	natural logarithm of x (base e)	`log( Math.E )` is 1.0 `log( Math.E * Math.E )` is 2.0
`max( x, y )`	larger value of x and y	`max( 2.3, 12.7 )` is 12.7 `max( -2.3, -12.7 )` is -2.3
`min( x, y )`	smaller value of x and y	`min( 2.3, 12.7 )` is 2.3 `min( -2.3, -12.7 )` is -12.7
`pow( x, y )`	x raised to the power y (i.e., x^y)	`pow( 2.0, 7.0 )` is 128.0 `pow( 9.0, 0.5 )` is 3.0
`sin( x )`	trigonometric sine of x (x in radians)	`sin( 0.0 )` is 0.0
`sqrt( x )`	square root of x	`sqrt( 900.0 )` is 30.0
`tan( x )`	trigonometric tangent of x (x in radians)	`tan( 0.0 )` is 0.0

6.4 Declaring Methods with Multiple Parameters:

Methods often require more than one piece of information to perform their tasks. We now consider how to write your own methods with multiple parameters.

```
import java.util.Scanner;
45
public class MaximumFinder
 {
8 public static void main( String[] args )
{
11 Scanner input = new Scanner( System.in );
14 System.out.print(
15 "Enter three floating-point values separated by spaces: " );
```

```
double number1 = input.nextDouble(); // read first double
17 double number2 = input.nextDouble(); // read second double
18 double number3 = input.nextDouble(); // read third double
20 // determine the maximum value
21
22
23 // display maximum value
24 System.out.println( );
25 } // end main
26// returns the maximum of its three double parameters
public static double maximum( double x, double y, double z )
{
double maximumValue = x; // assume x is the largest to start
// determine whether y is greater than maximumValue
if ( y > maximumValue )
maximumValue = y;
// determine whether z is greater than maximumValue
if ( z > maximumValue )
maximumValue = z;
return maximumValue;
```

6.5 Notes on Declaring and Using Methods

There are three ways to call a method:
1. Using a method name by itself to call another method of the *same* class—such as maximum(number1, number2, number3) in line 21 of Fig. 6.3.
2. Using a variable that contains a reference to an object, followed by a dot (.) and the method name to call a non-static method of the referenced object—such as the method call in line 13 of Fig. 5.10, myGradeBook.displayMessage(), which calls a method of class GradeBook from the main method of GradeBookTest.
3. Using the class name and a dot (.) to call a static method of a class—such as Math.sqrt(900.0) in Section 6.3.

6.6 Method-Call Stack and Activation Records

To understand how Java performs method calls, we first need to consider a data structure (i.e., collection of related data items) known as a stack. You can think of a stack as analogous to a pile of dishes. When a dish is placed on the pile, it's normally placed at the top (referred to as pushing the dish onto the stack). Similarly, when a dish is removed from the pile, it's always removed from the top (referred to as popping the dish off the stack). Stacks are known as last-in, first-out (LIFO) data structures—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

6.7 Argument Promotion and Casting

Another important feature of method calls is argument promotion—converting an argument's value, if possible, to the type that the method expects to receive in its corresponding parameter. For example, a program can call Math method sqrt with an int argument even though a double argument is expected. The statement correctly evaluates Math.sqrt(4) and prints the value 2.0. The method declaration's parameter list causes Java to convert the int value 4 to the double value 4.0 before passing the value to method sqrt. Such conversions may lead to compilation errors if Java's promotion rules are not satisfied. These rules specify which conversions are allowed—that is, which ones can be performed without losing data. In the sqrt example above, an int is converted to a double without changing its value. However, converting a double to an int truncates the fractional part of the double value—thus, part of the value is lost. Converting large integer types to small integer types (e.g., long to int, or int to short) may also result in changed values.

Type	Valid promotions
double	None
float	double
long	float or double
int	long, float or double
char	int, long, float or double
short	int, long, float or double (but not char)
byte	short, int, long, float or double (but not char)
boolean	None (boolean values are not considered to be numbers in Java)

6.8 Java API Packages

As you've seen, Java contains many predefined classes that are grouped into categories of related classes called packages.

Package	Description
`java.applet`	The **Java Applet Package** contains a class and several interfaces required to create Java applets—programs that execute in web browsers. Applets are discussed in Chapter 23, Applets and Java Web Start; interfaces are discussed in Chapter 10, Object-Oriented Programming: Polymorphism.)
`java.awt`	The **Java Abstract Window Toolkit Package** contains the classes and interfaces required to create and manipulate GUIs in early versions of Java. In current versions, the Swing GUI components of the `javax.swing` packages are typically used instead. (Some elements of the `java.awt` package are discussed in Chapter 14, GUI Components: Part 1, Chapter 15, Graphics and Java 2D, and Chapter 25, GUI Components: Part 2.)
`java.awt.event`	The **Java Abstract Window Toolkit Event Package** contains classes and interfaces that enable event handling for GUI components in both the `java.awt` and `javax.swing` packages. (See Chapter 14, GUI Components: Part 1, and Chapter 25, GUI Components: Part 2.)
`java.awt.geom`	The **Java 2D Shapes Package** contains classes and interfaces for working with Java's advanced two-dimensional graphics capabilities. (See Chapter 15, Graphics and Java 2D.)
`java.io`	The **Java Input/Output Package** contains classes and interfaces that enable programs to input and output data. (See Chapter 17, Files, Streams and Object Serialization.)
`java.lang`	The **Java Language Package** contains classes and interfaces (discussed bookwide) that are required by many Java programs. This package is imported by the compiler into all programs.
`java.net`	The **Java Networking Package** contains classes and interfaces that enable programs to communicate via computer networks like the Internet. (See Chapter 27, Networking.)
`java.sql`	The **JDBC Package** contains classes and interfaces for working with databases. (See Chapter 28, Accessing Databases with JDBC.)
`java.util`	The **Java Utilities Package** contains utility classes and interfaces that enable such actions as date and time manipulations, random-number processing (class `Random`) and the storing and processing of large amounts of data. (See Chapter 20, Generic Collections.)
`java.util.concurrent`	The **Java Concurrency Package** contains utility classes and interfaces for implementing programs that can perform multiple tasks in parallel. (See Chapter 26, Multithreading.)
`javax.media`	The **Java Media Framework Package** contains classes and interfaces for working with Java's multimedia capabilities. (See Chapter 24, Multimedia: Applets and Applications.)
`javax.swing`	The **Java Swing GUI Components Package** contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs. (See Chapter 14, GUI Components: Part 1, and Chapter 25, GUI Components: Part 2.)

6.9 Case Study: Random-Number Generation

We now take a brief diversion into a popular type of programming application—simulation and game playing. In this and the next section, we develop a nicely structured gameplaying program with multiple methods. The program uses most of the control statements

presented thus far in the book and introduces several new programming concepts.There's something in the air of a casino that invigorates people—from the high rollers at the plush mahogany-and-felt craps tables to the quarter poppers at the one-armed bandits. It's the element of chance, the possibility that luck will convert a pocketful of money into a mountain of wealth. The element of chance can be introduced in a program via an object of class **Random** (package java.util) or via the static method random of class Math. Objects of class Random can produce random boolean, byte, float, double, int, long and Gaussian values, whereas Math method random can produce only double values in the range $0.0 \leq x \square\square 1.0$, where x is the value returned by method random. In the next several examples, we use objects of class Random to produce random values. A new random-number generator object can be created as follows: It can then be used to generate random boolean, byte, float, double, int, long and Gaussian values—we discuss only random int values here. For more information on the Random class, see download.oracle.com/javase/6/docs/api/java/util/Random.html. Consider the following statement: Random method **nextInt** generates a random int value in the range –2,147,483,648 to +2,147,483,647, inclusive. If it truly produces values at random, then every value in the range should have an equal chance (or probability) of being chosen each time nextInt is called. The numbers are actually pseudorandom numbers—a sequence of values projavax.swing.event The Java Swing Event Package contains classes and interfaces that enable event handling (e.g., responding to button clicks) for GUI components in package javax.swing. (See Chapter 14, GUI Components: Part 1, and Chapter 25, GUI Components: Part 2.) javax.xml.ws The JAX-WS Package contains classes and interfaces for working with web services in Java. (See Chapter 31, Web Services.) Random randomNumbers = new Random(); int randomValue = randomNumbers.nextInt(); duced by a complex mathematical calculation. The calculation uses the current time of day (which, of course, changes constantly) to seed the random-number generator such that each execution of a program yields a different sequence of random values. The range of values produced directly by method nextInt generally differs from the range of values required in a particular Java application. For example, a program that simulates coin tossing might require only 0 for "heads" and 1 for "tails." A program that simulates the rolling of a six-sided die might require random integers in the range 1–6. A program that randomly predicts the next type of spaceship (out of four possibilities) that will fly across the horizon in a video game might require random integers in the range 1–4. For cases like these, class Random provides another version of method nextInt that receives an int argument and returns a value from 0 up to, but not including, the argument's value. For example, for coin tossing, the following statement returns 0 or 1.

```
public class RandomIntegers
6 {
7 public static void main( String[] args )
 {Random randomNumbers = new Random()

import java.util.Random;
public class RollDie
6 {
7 public static void main( String[] args )
8 {
9 Random randomNumbers = new Random(); // random number generator
11 int frequency1 = 0; // maintains count of 1s rolled
12 int frequency2 = 0; // count of 2s rolled
13 int frequency3 = 0; // count of 3s rolled
14 int frequency4 = 0; // count of 4s rolled
15 int frequency5 = 0; // count of 5s rolled
16 int frequency6 = 0; // count of 6s rolled
```

```
1  5  3  6  2
5  2  6  5  2
4  4  4  2  6
3  1  6  2  2
```

```
6  5  4  2  6
1  2  5  1  3
6  3  2  2  1
6  4  2  6  4
```

6.11 Scope of Declarations

You've seen declarations of various Java entities, such as classes, methods, variables and parameters. Declarations introduce names that can be used to refer to such Java entities. The scope of a declaration is the portion of the program that can refer to the declared entity by its name. Such an entity is said to be "in scope" for that portion of the program. This section introduces several important scope issues.

The basic scope rules are as follows:

1. The scope of a parameter declaration is the body of the method in which the declaration appears.

2. The scope of a local-variable declaration is from the point at which the declaration appears to the end of that block.

3. The scope of a local-variable declaration that appears in the initialization section of a `for` statement's header is the body of the `for` statement and the other expressions in the header.

4. A method or field's scope is the entire body of the class. This enables non-`static` methods of a class to use the fields and other methods of the class.

Any block may contain variable declarations. If a local variable or parameter in a method has the same name as a field of the class, the field is "hidden" until the block terminates execution—this is called shadowing. In Chapter 8, we discuss how to access shadowed fields.

6.12 Method Overloading

Methods of the same name can be declared in the same class, as long as they have different sets of parameters (determined by the number, types and order of the parameters)—this

is called method overloading. When an overloaded method is called, the compiler selects the appropriate method by examining the number, types and order of the arguments in the call. Method overloading is commonly used to create several methods with the *same* name that perform the *same* or *similar* tasks, but on different types or different numbers of arguments. For example, Math methods abs, min and max (summarized in Section 6.3) are overloaded with four versions each:

1. One with two double parameters.
2. One with two float parameters.
3. One with two int parameters.
4. One with two long parameters.

6.13 (Optional) GUI and Graphics Case Study: Colors and filled shapes

Although you can create many interesting designs with just lines and basic shapes, class Graphics provides many more capabilities. The next two features we introduce are colors and filled shapes. Adding color enriches the drawings a user sees on the computer screen. Shapes can be filled with solid colors. Colors displayed on computer screens are defined by their red, green, and blue components (called RGB values) that have integer values from 0 to 255. The higher the value of a component color, the richer that shade will be in the final color. Java uses class **Color** in package java.awt to represent colors using their RGB values. For convenience, class Color (package java.awt) contains 13 predefined static Color objects—BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED, WHITE and YELLOW. Each can be accessed via the class name and a dot (.) as in Color.RED. Class Color also contains a constructor of the form:

```
public Color( int r, int g, int b )
```

Graphics methods **fillRect** and **fillOval** draw filled rectangles and ovals, respectively. These have the same parameters as drawRect and drawOval; the first two are the coordinates for the upper-left corner of the shape, while the next two determine the width and height. The example in Fig. 6.11 and Fig. 6.12 demonstrates colors and filled shapes by drawing and displaying a yellow smiley face on the screen.

```
import java.awt.Color;
import java.awt.Graphics;
 import javax.swing.JPanel;
public class DrawSmiley extends JPanel
 {
 public voi
 {
 super.paintComponent( g );
g.setColor( Color.YELLOW );
g.fillOval( 10, 10, 200, 200 )
g.setColor( Color.BLACK );
g.fillOval( 55, 65, 30, 30 );
g.fillOval( 135, 65, 30, 30 );
g.fillOval( 50, 110, 120, 60 )
g.setColor( Color.YELLOW );
g.fillRect( 50, 110, 120, 30 );
g.fillOval( 50, 120, 120, 40 );
```


The import statements in lines 3–5 of Fig. 6.11 import classes Color, Graphics and JPanel. Class DrawSmiley (lines 7–30) uses class Color to specify drawing colors, and uses class Graphics to draw.

Class JPanel again provides the area in which we draw. Line 14 in method paintComponent uses Graphics method **setColor** to set the current drawing color to Color.YELLOW.

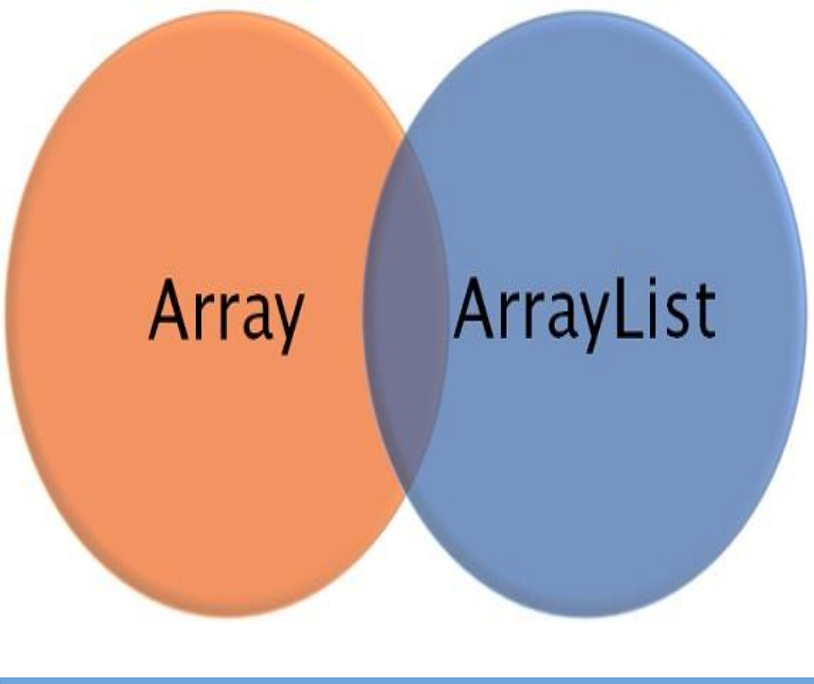
Method setColor requires one argument, the Color to set as the drawing color. In this case, we use the predefined object Color.YELLOW.

```
import javax.swing.JFrame;

public class DrawSmileyTest
 {
 public static void main( String[] args )
 { DrawSmiley panel = new DrawSmiley();
 JFrame application = new JFrame();
 application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 application.add( panel );
 application.setSize( 230, 250 );
 application.setVisible( true );
 }
 }
```

Output:

6.14 Wrap-Up

In this chapter, you learned more about method declarations. You also learned the difference between non-static and static methods and how to call static methods by preceding the method name with the name of the class in which it appears and the dot (.)
separator. You learned how to use operators + and += to perform string concatenations.
We discussed how the method-call stack and activation records keep track of the methods that have been called and where each method must return to when it completes its task.
We also discussed Java's promotion rules for converting implicitly between primitive types and how to perform explicit conversions with cast operators. Next, you learned about some of the commonly used packages in the Java API.



CHAPTER 7

Arrays and ArrayLists

Objectives

In this chapter we'll

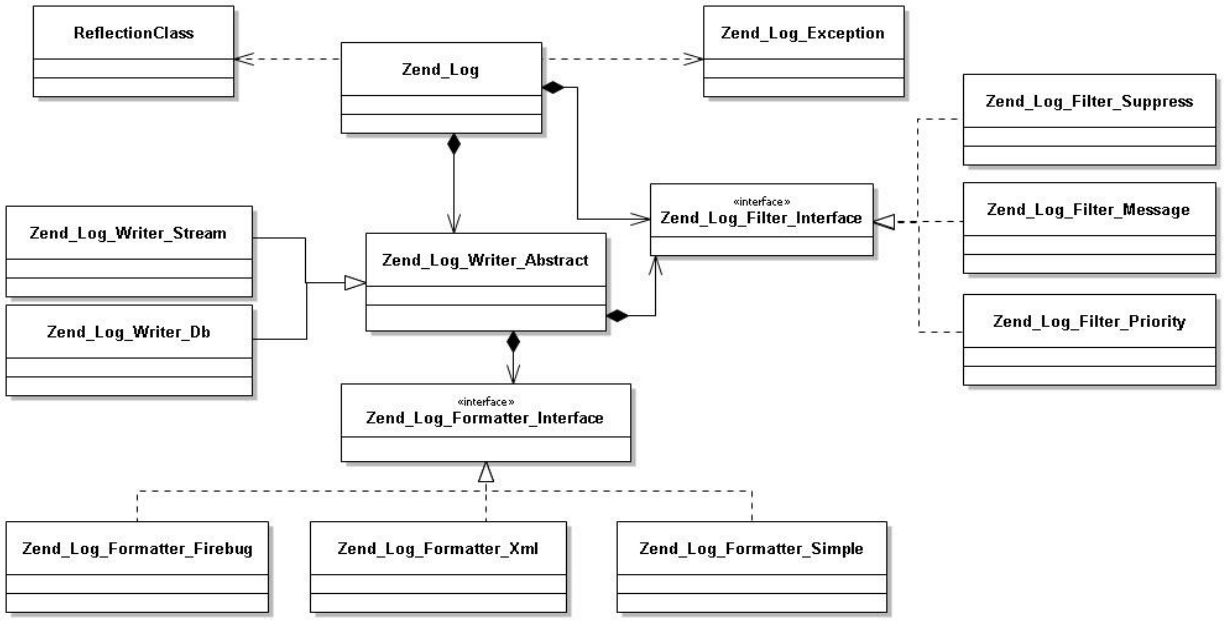
- To use arrays to store data in and retrieve data from lists and tables of values.
- To iterate through arrays with the enhanced for statement.
- To declare and manipulate multidimensional arrays.
- To read command-line arguments into a program.
- To use variable-length argument lists.

An array is a group of variables (called elements or components) containing values that all have the same type. Arrays are *objects*, so they're considered reference types. As you'll soon see, what we typically think of as an array is actually a reference to an array object in memory.

```
public class InitArray
 {
 public static void main( String[] args )
 {
int[] array; // declare array named array
array = new int[ 10 ]; // create the array object

 System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings
 // end main // end class InitArray
// output each array element's value
for ( int counter = 0; counter < array.length; counter++ )
System.out.printf( "%5d%8d\n", counter, array[ counter ] );
```

Calculating the Values to Store in an Array

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


 // calculate value for each array element
 for ( int counter = 0; counter < array.length; counter++ )

// initializer list specifies the value for each element
int[] array = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
final int ARRAY_LENGTH = 10; // declare constant
int[] array = new int[ ARRAY_LENGTH ]; // create array
array[ counter ] = 2 + 2 * counter;

 System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings

 // output each array element's value
 for ( int counter = 0; counter < array.length; counter++ )
 System.out.printf( "%5d%8d\n", counter, array[ counter ] );
 } // end main
 } // end class InitArray
```



Exception Handling: Processing the Incorrect Response

An exception indicates a problem that occurs while a program executes. The name "exception" suggests that the problem occurs infrequently—if the "rule" is that a statement normally executes correctly, then the problem represents the "exception to the rule." Exception handling enables you to create fault-tolerant programs that can resolve (or handle) exceptions. In many cases, this allows a program to continue executing as if no problems were encountered. For example, the StudentPoll application still displays results (Fig. 7.8), even though one of the responses was out of range. More severe problems might prevent a program from continuing normal execution, instead requiring the program to notify the user of the problem, then terminate. When the JVM or a method detects a problem, such as an invalid array index or an invalid method argument, it throws an exception—that is, an exception occurs.

*The **try** Statement*

To handle an exception, place any code that might throw an exception in a **try** statement (lines 17–26). The **try** block (lines 17–20) contains the code that might *throw* an exception, and the **catch** block (lines 21–26) contains the code that *handles* the exception if one occurs. You can have many catch blocks to handle different types of exceptions that might be thrown in the corresponding try block. When line 19 correctly increments an element of the frequency array, lines 21–26 are ignored. The braces that delimit the bodies of the try and catch blocks are required.

*Executing the **catch** Block*

When the program encounters the value 14 in the responses array, it attempts to add 1 to frequency[14], which is *outside* the bounds of the array—the frequency array has only six elements. Because array bounds checking is performed at execution time, the JVM generates

an exception—specifically line 19 throws an **ArrayIndexOutOfBoundsException** to notify the program of this problem. At this point the try block terminates and the catch block begins executing—if you declared any variables in the try block, they're now out of scope and are not accessible in the catch block.
The catch block declares a type (IndexOutOfRangeException) and an exception parameter (e). The catch block can handle exceptions of the specified type. Inside the catch block, you can use the parameter's identifier to interact with a caught exception object.

```
// Fig. 7.9: Card.java
 // Card class represents a playing card.

public class Card
 {
 private String face; // face of card ("Ace", "Deuce", ...)
 private String suit; // suit of card ("Hearts", "Diamonds", ...)
// two-argument constructor initializes card's face and suit
 public Card( String cardFace, String cardSuit )
 {
 face = cardFace; // initialize face of card
 suit = cardSuit; // initialize suit of card
} // end two-argument Card constructor
 } // end class Card
 // Fig. 7.10: DeckOfCards.java
 // DeckOfCards class represents a deck of playing cards. import java.util.Random;

public class DeckOfCards
 {

private int currentCard; // index of next Card to be dealt (0-51)
 private static final int NUMBER_OF_CARDS = 52; // constant # of Cards
 // random number generator
 private static final Random randomNumbers = new Random();

 // constructor fills deck of Cards
 public DeckOfCards()


 currentCard = 0; // set currentCard so first Card dealt is deck[ 0 ]
 // end DeckOfCards constructor

/ return String representation of Card
public String toString()
{
return face + " of " + suit;
} // end method toString
```

add Adds an element to the end of the ArrayList.
clear Removes all the elements from the ArrayList.
contains Returns true if the ArrayList contains the specified element; otherwise, returns false.
get Returns the element at the specified index.
indexOf Returns the index of the first occurrence of the specified element in the ArrayList.
remove Overloaded. Removes the first occurrence of the specified value or the element at the specified index.
size Returns the number of elements stored in the ArrayList.
trimToSize Trims the capacity of the ArrayList to current number of elements.

CLASS AND OBJECTS

CHAPTER 8

Classes and Objects

Objectives

In this chapter we'll

- Encapsulation and data hiding.
- To use static variables and methods.
- To use the enum type to create sets of constants with unique identifiers.
- To declare enum constants with parameters.
- To organize classes in packages to promote reuse.

```
// Fig. 8.5: Time2.java
 // Time2 class declaration with overloaded constructors.

public class Time2
 {
 private int hour; // 0 - 23
 private int minute; // 0 - 59
 private int second; // 0 - 59
```

Fig. 8.5 | Time2 class with overloaded constructors. (Part 1 of 3.)

```
// Time2 no-argument constructor:
// initializes each instance variable to zero
public Time2()
{
this( 0, 0, 0 );                  // invoke Time2 constructor with three arguments
} // end Time2 no-argument constructor
// Time2 constructor: hour supplied, minute and second defaulted to 0
public Time2( int h )
{
this( h, 0, 0 ); // invoke Time2 constructor with three arguments
} // end Time2 one-argument constructor
// Time2 constructor: hour and minute supplied, second defaulted to 0
public Time2( int h, int m )
{
this( h, m, 0 ); // invoke Time2 constructor with three arguments
} // end Time2 two-argument constructor
// Time2 constructor: hour, minute and second supplied
public Time2( int h, int m, int s )
{
setTime( h, m, s ); // invoke setTime to validate time
} // end Time2 three-argument constructor
// Time2 constructor: another Time2 object supplied
public Time2( Time2 time )
{
// invoke Time2 three-argument constructor
this( time.getHour(), time.getMinute(), time.getSecond() );
} // end Time2 constructor with a Time2 object argument
// Set Methods
 // set a new time value using universal time;
 // validate the data
 public void setTime( int h, int m, int s )
 {
 setHour( h ); // set the hour
```

```
 setMinute( m ); // set the minute
 setSecond( s ); // set the second
 } // end method setTime

 // validate and set hour
 public void setHour( int h )
 {
 if ( h >= 0 && h < 24 )
 hour = h;
 else
 throw new IllegalArgumentException( "hour must be 0-23" );
 } // end method setHour

 // validate and set minute
 public void setMinute( int m )
 {
 if ( m >= 0 && m < 60 )
 minute = m;
 else
 throw new IllegalArgumentException( "minute must be 0-59" );
 } // end method setMinute

 // validate and set second
 public void setSecond( int s )
 {
 if ( s >= 0 && s < 60 )
 second = ( ( s >= 0 && s < 60 ) ? s : 0 );
// Time2 constructor: hour and minute supplied, second defaulted to 0
public Time2( int h, int m )
{
this( h, m, 0 ); // invoke Time2 constructor with three arguments
} // end Time2 two-argument constructor
// Time2 constructor: hour, minute and second supplied
public Time2( int h, int m, int s )
{
setTime( h, m, s ); // invoke setTime to validate time
} // end Time2 three-argument constructor
// Time2 constructor: another Time2 object supplied
public Time2( Time2 time )
{
// invoke Time2 three-argument constructor
this( time.getHour(), time.getMinute(), time.getSecond() );
} // end Time2 constructor with a Time2 object argument
```

*Class **Time2**'s Constructors*

Lines 12–15 declare a so-called no-argument constructor that's invoked without arguments. Once you declare any constructors in a class, the compiler will *not* provide a default constructor. This no-argument constructor ensures that class Time2's clients can create Time2 objects with default values. Such a constructor simply initializes the object as specified in the constructor's body. In the body, we introduce a use of the this reference that's allowed only as the *first* statement in a constructor's body. Line 14 uses this in methodcall syntax to invoke the Time2 constructor that takes three parameters (lines 30–33) with values of 0 for the hour, minute and second. Using the this reference as shown here is a popular way to reuse initialization code provided by another of the class's constructors rather than defining similar code in the no-argument constructor's body. We use this syn-

```
                                        else
throw new IllegalArgumentException( "second must be 0-59" );
} // end method setSecond

// Get Methods
// get hour value
public int getHour()
{
return hour;
} // end method getHour

// get minute value
public int getMinute()
{
return minute;
} // end method getMinute

// get second value
public int getSecond()
{
return second; } // end method getSecond

// convert to String in universal-time format (HH:MM:SS)
public String toUniversalString()
{
return String.format(
"%02d:%02d:%02d", getHour(), getMinute(), getSecond() );
} // end method toUniversalString

// convert to String in standard-time format (H:MM:SS AM or PM)
public String toString()
{
return String.format( "%d:%02d:%02d %s",
```

```
( (getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12 ),
getMinute(), getSecond(), ( getHour() < 12 ? "AM" : "PM" ) );
} // end method toString
} // end class Time2
```

8.6 Default and No-Argument Constructors

Every class must have at least one constructor. If you do not provide any in a class's declaration, the compiler creates a default constructor that takes no arguments when it's invoked. The default constructor initializes the instance variables to the initial values
specified in their declarations or to their default values (zero for primitive numeric types, false for boolean values and null for references). In Section 9.4.1, you'll learn that the default constructor performs another task also.
If your class declares constructors, the compiler will *not* create a default constructor. In this case, you must declare a no-argument constructor if default initialization is required. Like a default constructor, a no-argument constructor is invoked with empty parentheses. The Time2 no-argument constructor (lines 12–15 of Fig. 8.5) explicitly initializes a Time2 object by passing to the three-argument constructor 0 for each parameter.
Since 0 is the default value for int instance variables, the no-argument constructor in this example could actually be declared with an empty body. In this case, each instance variable would receive its default value when the no-argument constructor was called. If we omit the no-argument constructor, clients of this class would not be able to create a Time2 object with the expression new Time2().

8.12 static Import

In Section 6.3, you learned about the static fields and methods of class Math. We invoked class Math's static fields and methods by preceding each with the class name Math and a dot (.). A **static** import declaration enables you to import the static members of a class or interface so you can access them via their unqualified names in your class—the class name and a dot (.) are not required to use an imported static member.
A static import declaration has two forms—one that imports a particular static member (which is known as single **static** import) and one that imports *all* static members of a class (known as **static** import on demand). The following syntax imports a particular static member:
where *packageName* is the package of the class (e.g., java.lang), *ClassName* is the name of the class (e.g., Math) and *staticMemberName* is the name of the static field or method (e.g., PI or abs). The following syntax imports all static members of a class:
The asterisk (*) indicates that *all* static members of the specified class should be available for use in the file. static import declarations import only static class members. Regular import statements should be used to specify the classes used in a program.
Figure 8.14 demonstrates a static import. Line 3 is a static import declaration, which imports all static fields and methods of class Math from package java.lang. Lines 9–12 access the Math class's static fields E (line 11) and PI (line 12) and the static

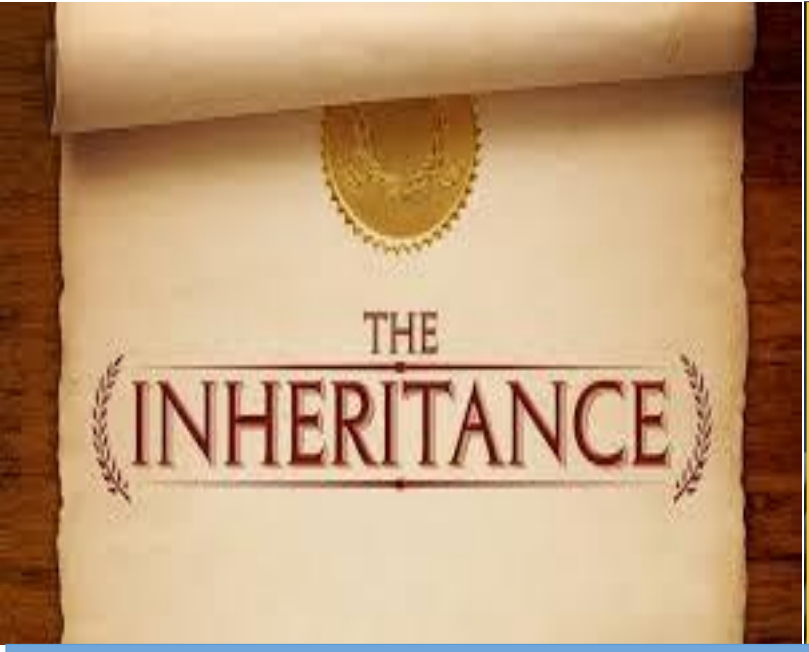
methods sqrt (line 9) and ceil (line 10) without preceding the field names or method names with class name Math and a dot.

```
Employees before instantiation: 0
Employee constructor: Susan Baker; count = 1
Employee constructor: Bob Blue; count = 2
Employees after instantiation:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2
Employee 1: Susan Baker
Employee 2: Bob Blue
```

```
import static packageName.ClassName.staticMemberName;
import static packageName.ClassName.*;
```

8.15 Package Access

If no access modifier (public, protected or private—we discuss protected in Chapter 9) is specified for a method or variable when it's declared in a class, the method or variable is considered to have package access. In a program that consists of one class.



CHAPTER 9

Object Oriented Programming: Inheritance

Objectives

In this chapter we'll

- The notions of superclasses and subclasses and the relationship between them.
- To access superclass members with super.
- To use access modifier protected to give subclass methods access to superclass members.
- The methods of class Object, the direct or indirect superclass of all classes.

9.2 Superclasses and Subclasses

Often, an object of one class *is an* object of another class as well. Figure 9.1 lists several simple examples of superclasses and subclasses—superclasses tend to be "more general" and subclasses "more specific." For example, a CarLoan *is a* Loan as are HomeImprovementLoans and MortgageLoans. Thus, in Java, class CarLoan can be said to inherit from class
Loan. In this context, class Loan is a superclass and class CarLoan is a subclass. A CarLoan *is a* specific type of Loan, but it's incorrect to claim that every Loan *is a* CarLoan—the Loan could be any type of loan.
Because every subclass object *is an* object of its superclass, and one superclass can have many subclasses, the set of objects represented by a superclass is often larger than the set of objects represented by any of its subclasses. For example, the superclass Vehicle represents all vehicles, including cars, trucks, boats, bicycles and so on. By contrast, subclass Car represents a smaller, more specific subset of vehicles.

University Community Member Hierarchy

Inheritance relationships form treelike hierarchical structures. A superclass exists in a hierarchical relationship with its subclasses. Let's develop a sample class hierarchy (Fig. 9.2), also called an inheritance hierarchy. A university community has thousands of members, including employees, students and alumni. Employees are either faculty or staff members. Faculty members are either administrators (e.g., deans and department chairpersons) or teachers. The hierarchy could contain many other classes. For example, students can be graduate or undergraduate students. Undergraduate students can be freshmen, sophomores, juniors or seniors.

Student GraduateStudent, UndergraduateStudent
Shape Circle, Triangle, Rectangle, Sphere, Cube
Loan CarLoan, HomeImprovementLoan, MortgageLoan
Employee Faculty, Staff
BankAccount CheckingAccount, SavingsAccount

9.4 Relationship between Superclasses and Subclasses

We now use an inheritance hierarchy containing types of employees in a company's payroll application to discuss the relationship between a superclass and its subclass. In this
company, commission employees (who will be represented as objects of a superclass) are paid a percentage of their sales, while base-salaried commission employees (who will be represented as objects of a subclass) receive a base salary *plus* a percentage of their sales.
We divide our discussion of the relationship between these classes into five examples.
The first declares class CommissionEmployee, which directly inherits from class Object and declares as private instance variables a first name, last name, social security number, commission rate and gross (i.e., total) sales amount.

The second example declares class BasePlusCommissionEmployee, which also directly inherits from class Object and declares as private instance variables a first name, last name, social security number, commission rate, gross sales amount *and* base salary. We create this class by *writing every line of code* the class requires—we'll soon see that it's much more efficient to create it by inheriting from class CommissionEmployee.
The third example declares a new BasePlusCommissionEmployee class that *extends* class CommissionEmployee (i.e., a BasePlusCommissionEmployee *is a* CommissionEmployee who also has a base salary). This *software reuse lets us write less code* when developing the new subclass. In this example, class BasePlusCommissionEmployee attempts to access class CommissionEmployee's private members—this results in compilation errors, because the subclass cannot access the superclass's private instance variables.
The fourth example shows that if CommissionEmployee's instance variables are declared as protected, the BasePlusCommissionEmployee subclass can access that data directly. Both BasePlusCommissionEmployee classes contain identical functionality, but we show how the inherited version is easier to create and manage.
After we discuss the convenience of using protected instance variables, we create the fifth example, which sets the CommissionEmployee instance variables back to private to enforce good software engineering. Then we show how the BasePlusCommissionEmployee subclass can use CommissionEmployee's public methods to manipulate (in a controlled manner) the private.

```
// Fig. 9.10: CommissionEmployee.java
 // CommissionEmployee class uses methods to manipulate its
 // private instance variables.
 public class CommissionEmployee

 // five-argument constructor
 public CommissionEmployee( String first, String last, String ssn,
 double sales, double rate )
 {
 // implicit call to Object constructor occurs here
 firstName = first;
 lastName = last;
 socialSecurityNumber = ssn;
 setGrossSales( sales ); // validate and store gross sales
 setCommissionRate( rate ); // validate and store commission rate
 } // end five-argument CommissionEmployee constructor

 // set first name
 public void setFirstName( String first )
 {
 firstName = first; // should validate
 } // end method setFirstName

 // return first name
 public String getFirstName()
 {
 return firstName;
 } // end method getFirstName

 // set last name
```

```
 public void setLastName( String last )
 {
variables. (Part 1 of 3.)
private String firstName;
private String lastName;
private String socialSecurityNumber;
private double grossSales; // gross weekly sales
private double commissionRate; // commission percentage
la5stName = last; // should validate
 } // end method setLastName

 // return last name
 public String getLastName()
 {
 return lastName;
 } // end method getLastName

 // set social security number
 public void setSocialSecurityNumber( String ssn )
 {
 socialSecurityNumber = ssn; // should validate
 } // end method setSocialSecurityNumber

 // return social security number
 public String getSocialSecurityNumber()
 {
 return socialSecurityNumber;
 } // end method getSocialSecurityNumber

 // set gross sales amount
 public void setGrossSales( double sales )
 {
 if ( sales >= 0.0 )
 grossSales = sales;
 else
 throw new IllegalArgumentException(
 "Gross sales must be >= 0.0" );
 } // end method setGrossSales

 // return gross sales amount
 public double getGrossSales()
 {
 return grossSales;
 } // end method getGrossSales

 // set commission rate
 public void setCommissionRate( double rate )
 {
 if ( rate > 0.0 && rate < 1.0 )
 commissionRate = rate;
 else
```

```
 throw new IllegalArgumentException(
 "Commission rate must be > 0.0 and < 1.0" );
 } // end method setCommissionRate

 // return commission rate
 public double getCommissionRate()
 {
 return commissionRate;
 } // end method getCommissionRate
 // calculate earnings
 public double earnings()
 {
 return * ;
 } // end method earnings

 // return String representation of CommissionEmployee object
 @Override // indicates that this method overrides a superclass method
 public String toString()
 {
return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
 "commission employee", , ,
 "social security number", ,
 "gross sales", ,
 "commission rate", );
 } // end method toString
} // end class CommissionEmployee
```

CHAPTER 10

Object Oriented Programming: Polymorphism

Objectives:

In this chapter you'll learn:

- The concept of polymorphism.
- To distinguish between abstract and concrete classes.
- To declare abstract methods to create abstract classes.
- How polymorphism makes systems extensible and maintainable.
- To distinguish between Interface and abstract classes.
- To declare and implement interfaces.

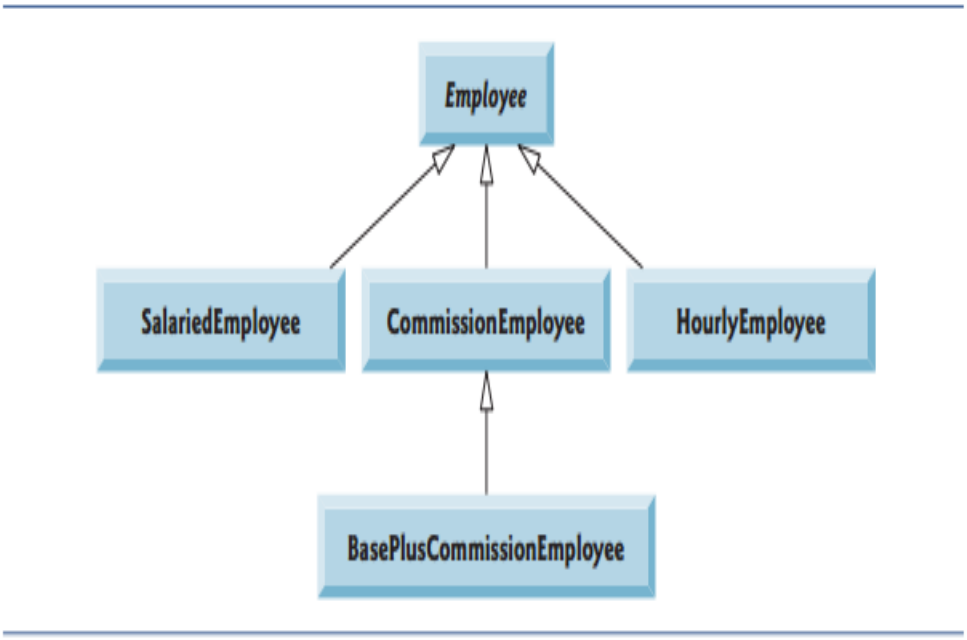
10.1: Polymorphism

*For Polymorphism we use **"extends"** keyword*

"Public class SalariedEmployee extends Employee"

Polymorphism enables you to deal in generalities and let the execution-time environment handle the specifics. You can command objects to behave in manners appropriate to those objects, without knowing their types (as long as the objects belong to the same inheritance hierarchy).

Polymorphism promotes extensibility: Software that invokes polymorphic behavior is independent of the object types to which messages are sent. New object types that can respond to existing method calls can be incorporated into a system without modifying the base system.



10.2 Abstract Class:

10.2.1: Declaring an Abstract Class and Abstract Methods

*We make a class abstract by declaring it with keyword **abstract.** An abstract class normally contains one or more **abstract methods**. An abstract method is one with keyword abstract in its declaration, as in*

"Public abstract void draw(); // abstract method".

Abstract methods don't provide implementations. A class that contains *any* abstract methods must be explicitly declared abstract even if that class contains some concrete (nonabstract) methods. Each concrete subclass of an abstract superclass also must provide concrete implementations of each of the superclass's abstract methods. Constructors and static methods cannot be declared abstract. Constructors are not inherited, so an abstract constructor could never be implemented.

	earnings	toString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklySalary*

Hourly-Employee	if (hours <= 40)
	  wage * hours
	else if (hours > 40)
	{
	  40 * wage +
	  (hours - 40) *
	  wage * 1.5
	}
		hourly employee: *firstName lastName*
		social security number: *SSN*
		hourly wage: *wage*; hours worked: *hours*

Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	(commissionRate * grossSales) + baseSalary	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Abstract Class Employee:

```
public abstract class Employee
{
 private String firstName;
 private String lastName;
 private String socialSecurityNumber;
 public Employee( String first, String last, String ssn ) // three-argument constructor
 {
 firstName = first;
 lastName = last;
 socialSecurityNumber = ssn;
 } // end three-argument Employee constructor
19 public void setFirstName( String first ) // set first name
20 {
21 firstName = first; // should validate
22 }
23 // end method setFirstName
25 public String getFirstName()// return first name
26 {
27 return firstName;
28 } // end method getFirstName
31 public void setLastName( String last ) // set last name
32 {
33 lastName = last; // should validate
34 } // end method setLastName
37 public String getLastName() // return last name
38 {
39 return lastName;
40 } // end method getLastName
43 public void setSocialSecurityNumber( String ssn ) // set social security number
44 {
45 socialSecurityNumber = ssn; // should validate
46 } // end method setSocialSecurityNumber
49 public String getSocialSecurityNumber() // return social security number
50 {
51 return socialSecurityNumber;
52 } // end method getSocialSecurityNumber
54 // return String representation of Employee object
55 @Override
56 public String toString()
57 {
58 return String.format( "%s %s\nsocial security number: %s",
59 getFirstName(), getLastName(), getSocialSecurityNumber() );
60 } // end method toString
62 // abstract method overridden by concrete subclasses
63 public abstract double earnings(); // no implementation here
64 }
```

Class SalariedEmployee:

```
public class SalariedEmployee extends Employee
{
6 private double weeklySalary;
78
// four-argument constructor
9 public SalariedEmployee( String first, String last, String ssn,
10 double salary )
11 {
12 super( first, last, ssn ); // pass to Employee constructor
13 setWeeklySalary( salary ); // validate and store salary
14 } // end four-argument SalariedEmployee constructor
15
16 // set salary
17 public void setWeeklySalary( double salary )
18 {
19 if ( salary >= 0.0 )
20 baseSalary = salary;
21 else
22 throw new IllegalArgumentException(
23 "Weekly salary must be >= 0.0" );
24 } // end method setWeeklySalary
25
// return salary
27 public double getWeeklySalary()
28 {
29 return weeklySalary;
30 } // end method getWeeklySalary
// calculate earnings; override abstract method earnings in Employee
@Override
public double earnings()
{
return getWeeklySalary();
} // end method earnings
```

Class HourlyEmployee:

```
public class HourlyEmployee extends Employee
{
6 private double wage; // wage per hour
7 private double hours; // hours worked for week
// five-argument constructor
10 public HourlyEmployee( String first, String last, String ssn,
11 double hourlyWage, double hoursWorked )
12 {
13 super( first, last, ssn );
14 setWage( hourlyWage ); // validate hourly wage
15 setHours( hoursWorked ); // validate hours worked
```

```
16 } // end five-argument HourlyEmployee constructor
17
18 // set wage
19 public void setWage( double hourlyWage )
20 {
21 if ( hourlyWage >= 0.0 )
22 wage = hourlyWage;
23 else
24 throw new IllegalArgumentException(
25 "Hourly wage must be >= 0.0" );
26 } // end method setWage
27
28 // return wage
29 public double getWage()
30 {
31 return wage;
32 } // end method getWage
34 // set hours worked
35 public void setHours( double hoursWorked )
36 {
37 if ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) )
38 hours = hoursWorked;
39 else
40 throw new IllegalArgumentException(
41 "Hours worked must be >= 0.0 and <= 168.0" );
42 } // end method setHours
43
44 // return hours worked
45 public double getHours()
46 {
47 return hours;
48 } // end method getHours
// calculate earnings; override abstract method earnings in Employee
@Override
public double earnings()
{
if ( getHours() <= 40 ) // no overtime
return getWage() * getHours();
else
return 40 * getWage() + ( getHours() - 40 ) * getWage() * 1.5;
} // end method earnings
// return String representation of HourlyEmployee object
@Override
public String toString()
{
return String.format( "hourly employee: %s\n%s: $%,.2f; %s: %,.2f",
super.toString(), "hourly wage", getWage(),
"hours worked", getHours() );
} // end method toString
} // end class HourlyEmployee
```

Class CommissionEmployee:

```
public class CommissionEmployee extends Employee
{
6 private double grossSales; // gross weekly sales
7 private double commissionRate; // commission percentage
// five-argument constructor
10 public CommissionEmployee( String first, String last, String ssn,
11 double sales, double rate )
12 {
13 super( first, last, ssn );
setGrossSales( sales );
15 setCommissionRate( rate );
16 } // end five-argument CommissionEmployee constructor
17
18 // set commission rate
19 public void setCommissionRate( double rate )
20 {
21 if ( rate > 0.0 && rate < 1.0 )
22 commissionRate = rate;
23 else
24 throw new IllegalArgumentException(
25 "Commission rate must be > 0.0 and < 1.0" );
26 } // end method setCommissionRate
28 // return commission rate
29 public double getCommissionRate()
30 {
31 return commissionRate;
32 } // end method getCommissionRate
34 // set gross sales amount
35 public void setGrossSales( double sales )
36 {
37 if ( sales >= 0.0 )
38 grossSales = sales;
39 else
40 throw new IllegalArgumentException(
41 "Gross sales must be >= 0.0" );
42 } // end method setGrossSales
44 // return gross sales amount
45 public double getGrossSales()
46 {
47 return grossSales;
48 } // end method getGrossSales
// calculate earnings; override abstract method earnings in Employee
@Override
public double earnings()
{
return getCommissionRate() * getGrossSales();
} // end method earnings
// return String representation of CommissionEmployee object
```

```
@Override
public String toString()
{
return String.format( "%s: %s\n%s: $%,.2f; %s: %.2f",
"commission employee", super.toString(),
"gross sales", getGrossSales(),
"commission rate", getCommissionRate() );
} // end method toString
} // end class CommissionEmployee
```

Class BasePlusCommissionEmployee:

```
public class BasePlusCommissionEmployee extends CommissionEmployee
{
6 private double baseSalary; // base salary per week
// six-argument constructor
9 public BasePlusCommissionEmployee( String first, String last,
10 String ssn, double sales, double rate, double salary )
11 {
12 super( first, last, ssn, sales, rate );
13 setBaseSalary( salary ); // validate and store base salary
14 } // end six-argument BasePlusCommissionEmployee constructor
17 public void setBaseSalary( double salary ) // set base salary
18 {
19 if ( salary >= 0.0 )
20 baseSalary = salary;
21 else
22 throw new IllegalArgumentException(
23 "Base salary must be >= 0.0" );
24 } // end method setBaseSalary
27 public double getBaseSalary()// return base salary
28 {
29 return baseSalary;
30 } // end method getBaseSalary
// calculate earnings; override method earnings in CommissionEmployee
@Override
public double earnings()
{
return getBaseSalary() + super.earnings();
} // end method earnings
// return String representation of BasePlusCommissionEmployee object
@Override
public String toString()
{
return String.format( "%s %s; %s: $%,.2f",
"base-salaried", super.toString(),
"base salary", getBaseSalary() );
} // end method toString
} // end class BasePlusCommissionEmployee
```

Class PayrollSystemTest:

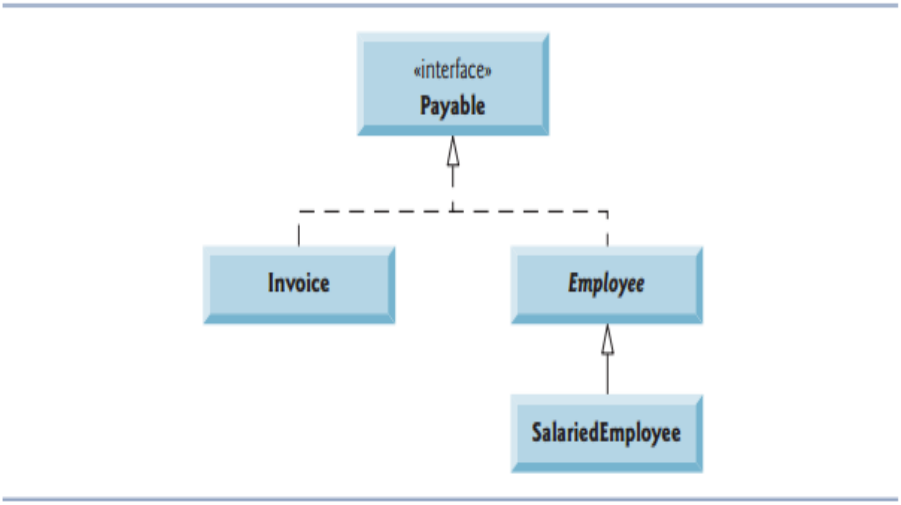
```
public class PayrollSystemTest
 {
public static void main( String[] args )
 {
// create subclass objects
SalariedEmployee salariedEmployee = new SalariedEmployee(
"John", "Smith", "111-11-1111", 800.00 );
HourlyEmployee hourlyEmployee =
new HourlyEmployee( "Karen", "Price", "222-22-2222", 16.75, 40 );
CommissionEmployee commissionEmployee =
new CommissionEmployee("Sue", "Jones", "333-33-3333", 10000, .06 );
BasePlusCommissionEmployee basePlusCommissionEmployee =
new BasePlusCommissionEmployee(
"Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
System.out.println( "Employees processed individually:\n" );
21
22 System.out.printf( "%s\n%s: $%,.2f\n\n",
23 salariedEmployee, "earned", salariedEmployee.earnings() );
24 System.out.printf( "%s\n%s: $%,.2f\n\n",
25 hourlyEmployee, "earned", hourlyEmployee.earnings() );
26 System.out.printf( "%s\n%s: $%,.2f\n\n",
27 commissionEmployee, "earned", commissionEmployee.earnings() );
28 System.out.printf( "%s\n%s: $%,.2f\n\n",
29 basePlusCommissionEmployee,
30 "earned", basePlusCommissionEmployee.earnings() );
31
32 // create four-element Employee array
Employee[] employees = new Employee[ 4 ];
// initialize array with Employees
employees[ 0 ] = salariedEmployee;
employees[ 1 ] = hourlyEmployee;
employees[ 2 ] = commissionEmployee;
employees[ 3 ] = basePlusCommissionEmployee;
System.out.println( "Employees processed polymorphically:\n" );
42
43 // generically process each element in array employees
44 for ( Employee currentEmployee : employees )
45 {
46 System.out.println( ); // invokes toString
47
48 // determine whether element is a BasePlusCommissionEmployee
49 if ( )
50 {
51 // downcast Employee reference to
52 // BasePlusCommissionEmployee reference
53 BasePlusCommissionEmployee employee =
employee.setBaseSalary( 1.10 * employee.getBaseSalary() );
57
```

```
58 System.out.printf(
59 "new base salary with 10%% increase is: $%,.2f\n",
60 employee.getBaseSalary() );
61 } // end if

System.out.printf(
64 "earned $%,.2f\n\n", );
65 } // end for
// get type name of each object in employees array
for ( int j = 0; j < employees.length; j++ )
System.out.printf( "Employee %d is a %s\n", j,
employees[ j ].getClass().getName() );
   } // end main
 } // end class PayrollSystemTest
```

10.5: Interface and Abstract

To use an interface, a concrete class must specify that it **implements** the interface and must declare each method in the interface with the signature specified in the interface declaration. To specify that a class implements an interface add the implements keyword and the name of the interface to the end of your class declaration's first line. A class that does not implement *all* the methods of the interface is an *abstract* class and must be declared abstract.

An **interface** is often used in place of an **abstract** class when there's no default implementation to inherit—that is, no fields and no default method implementations. Like public **abstract** classes, **interfaces** are typically public types. Like a public class, a public **interface** must be declared in a file with the same name as the interface and the .java file-name extension.



Interface:

```
Public interface Payable
{
double getPaymentAmount(); // calculate payment; no implementation
} // end interface Payable
```

Class Invoice:

```
Public class Invoice implements Payable
{
6 private String partNumber;
7 private String partDescription;
8 private int quantity;
9 private double pricePerItem;
10
11 // four-argument constructor
12 public Invoice( String part, String description, int count,
13 double price )
14 {
15 partNumber = part;
16 partDescription = description;
17 setQuantity( count ); // validate and store quantity
18 setPricePerItem( price ); // validate and store price per item
19 } // end four-argument Invoice constructor
20
21 // set part number
22 public void setPartNumber( String part )
23 {
24 partNumber = part; // should validate
25 } // end method setPartNumber
26
27 // get part number
28 public String getPartNumber()
29 {
30 return partNumber;
31 } // end method getPartNumber
32
33 // set description
34 public void setPartDescription( String description )
35 {
36 partDescription = description; // should validate
37 } // end method setPartDescription
38
39 // get description
40 public String getPartDescription()
41 {
```

```
42 return partDescription;
43 } // end method getPartDescription
44
45 // set quantity
46 public void setQuantity( int count )
47 {
48 if ( count >= 0 )
49 quantity = count;
50 else
51 throw new IllegalArgumentException( "Quantity must be >= 0" );
52 } // end method setQuantity
// get quantity
55 public int getQuantity()
56 {
57 return quantity;
58 } // end method getQuantity
59
60 // set price per item
61 public void setPricePerItem( double price )
62 {
63 if ( price >= 0.0 )
64 pricePerItem = price;
65 else
66 throw new IllegalArgumentException(
67 "Price per item must be >= 0" );
68 } // end method setPricePerItem
69
70 // get price per item
71 public double getPricePerItem()
72 {
73 return pricePerItem;
74 } // end method getPricePerItem
75
76 // return String representation of Invoice object
77 @Override
78 public String toString()
79 {
80 return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
81 "invoice", "part number", getPartNumber(), getPartDescription(),
82 "quantity", getQuantity(), "price per item", getPricePerItem() );
83 } // end method toString
// method required to carry out contract with interface Payable
@Override
public double getPaymentAmount()
{
return getQuantity() * getPricePerItem(); // calculate total cost
} // end method getPaymentAmount
} // end class Invoice
```

Abstract Class Employee:

```
public abstract class Employee implements Payable
{
6 private String firstName;
7 private String lastName;
8 private String socialSecurityNumber;
9
10 // three-argument constructor
11 public Employee( String first, String last, String ssn )
12 {
13 firstName = first;
14 lastName = last;
15 socialSecurityNumber = ssn;
16 } // end three-argument Employee constructor
17
18 // set first name
19 public void setFirstName( String first )
20 {
21 firstName = first; // should validate
22 } // end method setFirstName
23
24 // return first name
25 public String getFirstName()
26 {
27 return firstName;
28 } // end method getFirstName
// set last name
31 public void setLastName( String last )
32 {
33 lastName = last; // should validate
34 } // end method setLastName
35
36 // return last name
37 public String getLastName()
38 {
39 return lastName;
40 } // end method getLastName
41
42 // set social security number
43 public void setSocialSecurityNumber( String ssn )
44 {
45 socialSecurityNumber = ssn; // should validate
46 } // end method setSocialSecurityNumber
47
48 // return social security number
49 public String getSocialSecurityNumber()
50 {
```

```
51 return socialSecurityNumber;
52 } // end method getSocialSecurityNumber
53
54 // return String representation of Employee object
55 @Override
56 public String toString()
57 {
58 return String.format( "%s %s\nsocial security number: %s",
59 getFirstName(), getLastName(), getSocialSecurityNumber() );
60 } // end method toString
// Note: We do not implement Payable method getPaymentAmount here so
// this class must be declared abstract to avoid a compilation error
} // end abstract class Employee
```

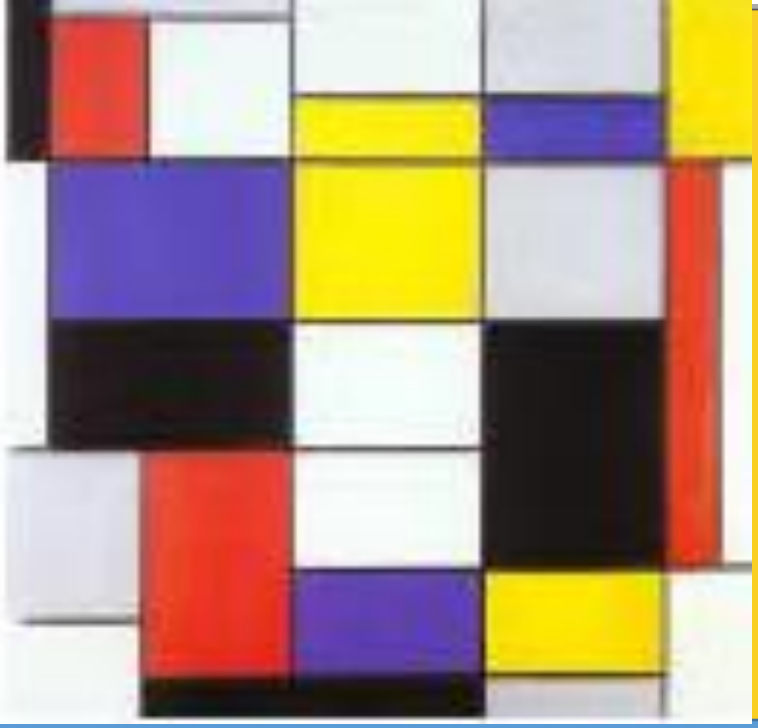
Class SalariedEmployee:

```
public class SalariedEmployee extends Employee
5 {
6 private double weeklySalary;
78
// four-argument constructor
9 public SalariedEmployee( String first, String last, String ssn,
10 double salary )
11 {
super( first, last, ssn ); // pass to Employee constructor
13 setWeeklySalary( salary ); // validate and store salary
14 } // end four-argument SalariedEmployee constructor
15
16 // set salary
17 public void setWeeklySalary( double salary )
18 {
19 if ( salary >= 0.0 )
20 baseSalary = salary;
21 else
22 throw new IllegalArgumentException(
23 "Weekly salary must be >= 0.0" );
24 } // end method setWeeklySalary
25
26 // return salary
27 public double getWeeklySalary()
28 {
29 return weeklySalary;
30 } // end method getWeeklySalary
// calculate earnings; implement interface Payable method that was
// abstract in superclass Employee
@Override
public double getPaymentAmount()
{
return getWeeklySalary();
```

```
} // end method getPaymentAmount


// return String representation of SalariedEmployee object
41 @Override
42 public String toString()
43 {
44 return String.format( "salaried employee: %s\n%s: $%,.2f",
45 super.toString(), "weekly salary", getWeeklySalary() );
46 } // end method toString
47 } // end class SalariedEmployee
```

Class PayableInterfaceTest:

```
public class PayableInterfaceTest
5 {
6 public static void main( String[] args )
7 {

8 // create four-element Payable array
Payable[] payableObjects = new Payable[ 4 ];
// populate array with objects that implement Payable
12 payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00 );
13 payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95 );
14 payableObjects[ 2 ] =
15 new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
16 payableObjects[ 3 ] =
17 new SalariedEmployee( "Lisa", "Barnes", "888-88-8888", 1200.00 );
18
19 System.out.println(
20 "Invoices and Employees processed polymorphically:\n" );
21
22 // generically process each element in array payableObjects
23 for ( Payable currentPayable : payableObjects )
24 {

25 // output currentPayable and its appropriate payment amount
26 System.out.printf( "%s \n%s: $%,.2f\n\n",

"payment due", currentPayable.getPaymentAmount(



} // end for
30 } // end main
31 } // end class PayableInterfaceTest
```



CHAPTER 11

Exception Handling: A Deeper Look

Objectives:

In this chapter you'll learn:

- What exceptions are and how they're handled.
- When to use exception handling.
- To use try blocks to delimit code in which exceptions might occur.
- To throw exceptions to indicate a problem.
- To use catch blocks to specify exception handlers.
- To use the finally block to release resources.
- The exception class hierarchy.
- Assertions

11. 1: Exception Handling:

Exception handling is the process of responding to the occurrence, during computation, of **exceptions** – anomalous or exceptional conditions requiring special processing – often changing the normal flow of program execution.

Exception handling is the process of responding to the occurrence, during computation, of **exceptions** – anomalous or exceptional conditions requiring special processing – often changing the normal flow of program execution.

```
try ( ClassName theObject = new ClassName() )
{
// use theObject here
}
catch ( Exception e )
{
// catch exceptions that occur while using the resource
}
```

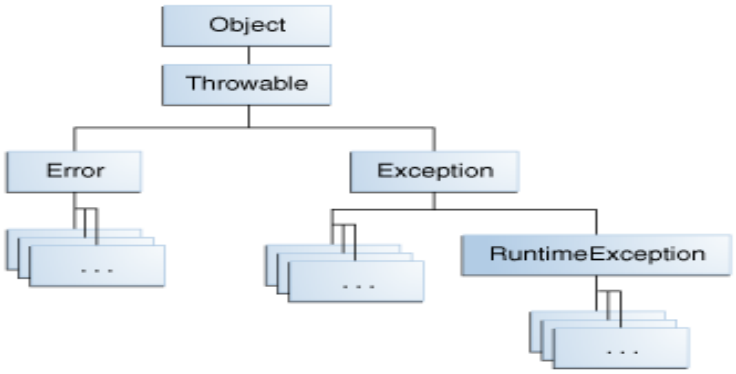
11.4: Throw Exception:

Syntax of Throw exception

```
throw AnyThrowableInstance;
```

Example:

```
//A void method
public void sample()
{
  //Statements
  //if (somethingWrong) then
  IOException e = new IOException();
  throw e;
  //More Statements
 }
```



The Throwable class.

11.5: Catch Exception:

```
MyClass obj =  new MyClass();
try{
     obj.sample();
}catch(IOException ioe)
 {
     //Your error Message here
     System.out.println(ioe);
  }
```

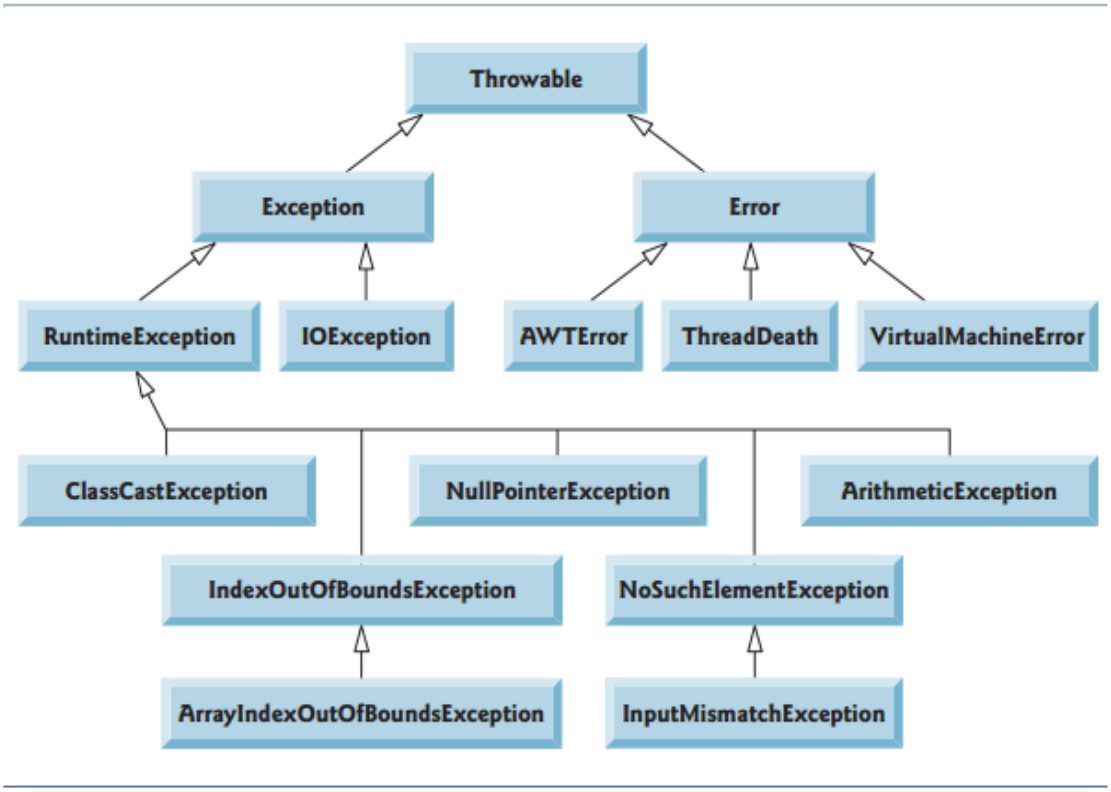
11.6: Finally Exception Block:

```
try {
   line = console.readLine();

   if (line.length() == 0) {
      throw new EmptyLineException("The line read from console was empty!");
   }

   console.printLine("Hello %s!" % line);
   console.printLine("The program ran successfully");
}
catch (EmptyLineException e) {
   console.printLine("Hello!");
```

```
}
catch (Exception e) {
   console.printLine("Error: " + e.message());
}
finally {
   console.printLine("The program terminates now");
}
```

11.7 Java Exception hierarchy:

All Java exception classes inherit directly or indirectly from class **Exception**, forming an **inheritance hierarchy**. Only **Throwable** objects can be used with the exception-handling mechanism. **Class Throwable** has two subclasses: **Exception** and **Error**. Class Exception and its subclasses—for instance, **RuntimeException** (package java.lang) and **IOException** (package java.io)—represent exceptional situations that can occur in a Java program and that can be caught by the application.



Exception Handling:

```
import java.util.InputMismatchException;
import java.util.Scanner;
56
public class DivideByZeroWithExceptionHandling
7 {
8 // demonstrates throwing an exception when a divide-by-zero occurs
9 public static int quotient( int numerator, int denominator )
throws ArithmeticException
{
12 return numerator / denominator; // possible division by zero
13 }
 // end method quotient
14
15 public static void main( String[] args )
16 {
17 Scanner scanner = new Scanner( System.in ); // scanner for input
18 boolean continueLoop = true; // determines if more input is needed
do
{
try // read two numbers and calculate quotient
{
System.out.print( "Please enter an integer numerator: " );
int numerator = scanner.nextInt();
System.out.print( "Please enter an integer denominator: " );
int denominator = scanner.nextInt();
int result = quotient( numerator, denominator );
System.out.printf( "\nResult: %d / %d = %d\n", numerator,
denominator, result );
continueLoop = false; // input successful; end looping
} // end try
catch ( InputMismatchException inputMismatchException )
{
System.err.printf( "\nException: %s\n",
inputMismatchException );
scanner.nextLine(); // discard input so user can try again
System.out.println(
"You must enter integers. Please try again.\n" );
} // end catch
catch ( ArithmeticException arithmeticException )
{
System.err.printf( "\nException: %s\n", arithmeticException );
System.out.println(
"Zero is an invalid denominator. Please try again.\n" );
} // end catch
} while ( continueLoop ); // end do...while
} // end main
} // end class DivideByZeroWithExceptionHandling
```

Finally Block Exception:

```
public class UsingExceptions
5 {
6 public static void main( String[] args )
7 {
8 try
9 {
10 throwException(); // call method throwException
11 } // end try
12 catch ( Exception exception ) // exception thrown by throwException
13 {
14 System.err.println( "Exception handled in main" );
15 } // end catch
16
17 doesNotThrowException();
18 } // end main
20 // demonstrate try...catch...finally
21 public static void throwException() throws Exception
22 {
 try // throw an exception and immediately catch it
24 {
System.out.println( "Method throwException" );
throw new Exception(); // generate exception
} // end try
28 catch ( Exception exception ) // catch exception thrown in try
29 {
30 System.err.println(
31 "Exception handled in method throwException" );
throw exception; // rethrow for further processing
// code here would not be reached; would cause compilation errors
36 } // end catch
finally // executes regardless of what occurs in try...catch
{
System.err.println( "Finally executed in throwException" );
} // end finally
// code here would not be reached; would cause compilation errors
44 } // end method throwException
46 // demonstrate finally when no exception occurs
47 public static void doesNotThrowException()
48 {
49 try // try block does not throw an exception
50 {
51 System.out.println( "Method doesNotThrowException" );
52 } // end try
53 catch ( Exception exception ) // does not execute
54 {
55 System.err.println( exception );
56 } // end catch
finally // executes regardless of what occurs in try...catch
```

```
{ System.err. println(
"Finally executed in doesNotThrowException" );
} // end finally
System.out.println( "End of method doesNotThrowException" );
64 } // end method doesNotThrowException
65 } // end class UsingExceptions
```

11.8: Assertions:

When implementing and debugging a class, it's sometimes useful to state conditions that should be true at a particular point in a method. These conditions, called **assertions**, help ensure a program's validity by catching potential bugs and identifying possible logic errors during development.

assert expression;

assert *expression1* : *expression2*;

```
import java.util.Scanner;
45
public class AssertTest
6 {
7 public static void main( String[] args )
8 {

9 Scanner input = new Scanner( System.in );
System.out.print( "Enter a number between 0 and 10: " );
12 int number = input.nextInt();
13
14 // assert that the value is >= 0 and <= 10
15 assert ( number >= 0 && number <= 10 ) : "bad number: " + number;
16
17 System.out.printf( "You entered %d\n", number );
18 } // end main
19 } // end class AssertTest
```

CHAPTER 12

ATM Case Study, Part I:
Object Oriented Design With UML Diagram:

Objectives

In this chapter you'll learn:

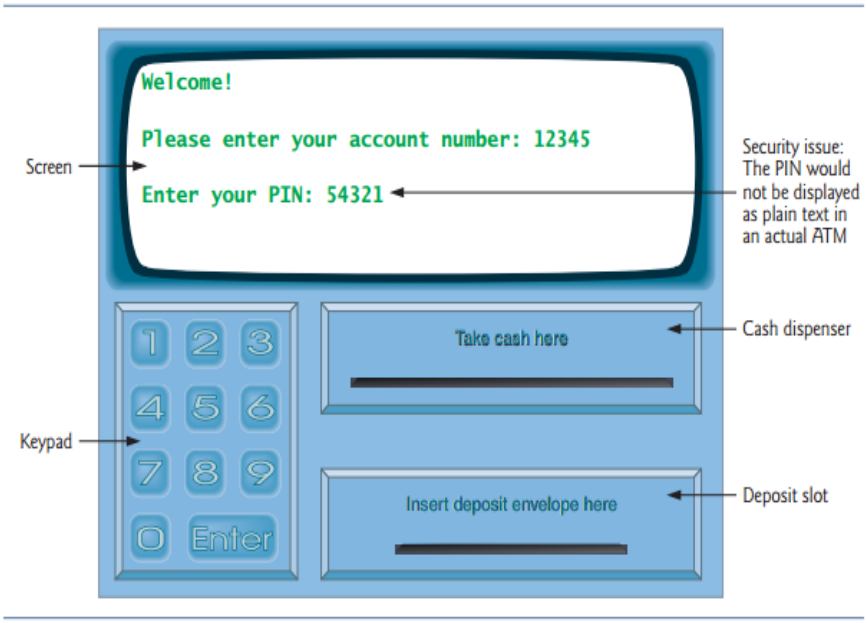
- A simple object-oriented design methodology.
- What a requirements document is.
- To identify classes and class attributes from a requirements document.
- To identify objects' states, activities and operations from a requirements document.
- To work with the UML's use case, class, state, activity, communication and sequence diagrams to graphically model an object oriented system.

12.2 Examining the Requirements Document:

Requirements Document:

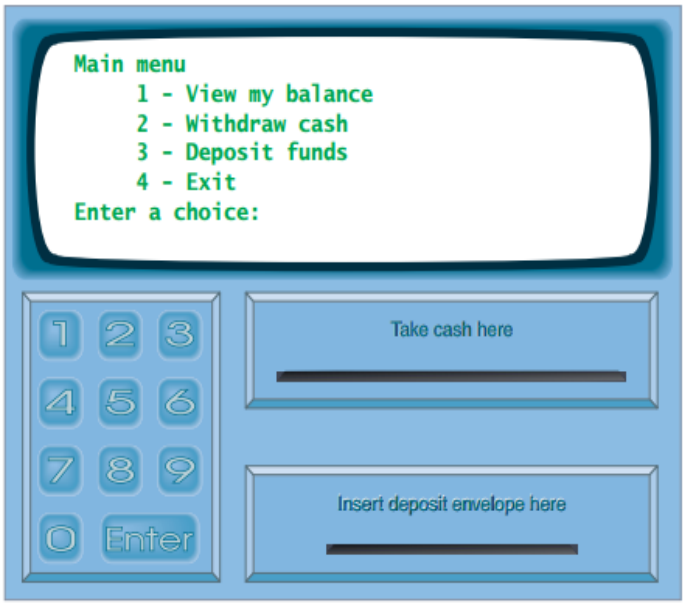
A local bank intends to install a new automated teller machine (ATM) to allow users (i.e., bank customers) to perform basic financial transactions. Each user can have only one account at the bank. **ATM** users should be able to view their account balance and withdraw cash (i.e., take money out of an account) and deposit funds. The user interface of the automated teller machine contains:

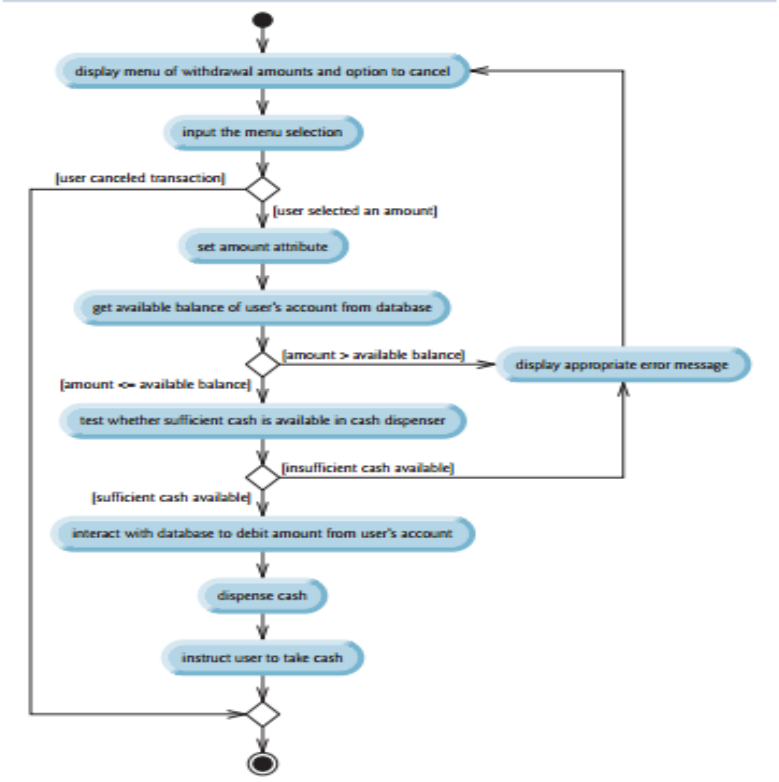
- **a screen that displays messages to the user**
- **a keypad that receives numeric input from the user**
- **a cash dispenser that dispenses cash to the user and**
- **a deposit slot that receives deposit envelopes from the user.**



Functions:

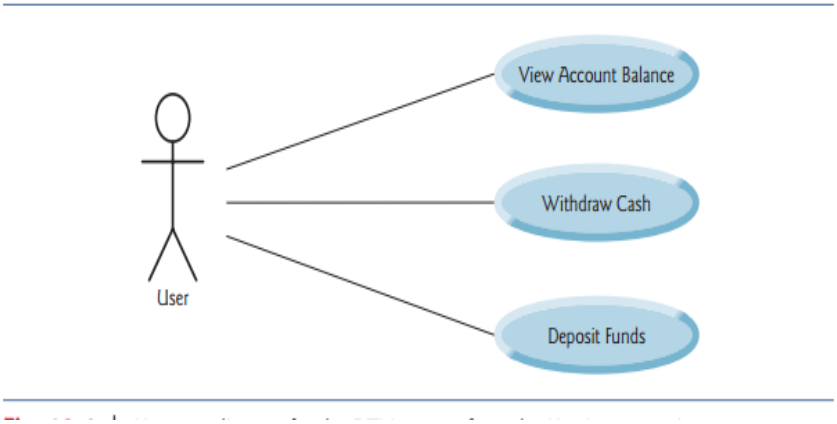
The screen displays Welcome! and prompts the user to enter an account number.
2. The user enters a five-digit account number using the keypad.
3. The screen prompts the user to enter the PIN (personal identification number) associated with the specified account number.
4. The user enters a five-digit PIN using the keypad.1
5. If the user enters a valid account number and the correct PIN for that account, the screen displays the main menu. If the user enters an invalid account number or an incorrect PIN, the screen displays an appropriate message,
then the ATM returns to Step 1 to restart the authentication process.


Main menu
1 - View my balance
2 - Withdraw cash
3 - Deposit funds
4 - Exit
Enter a choice:
1 2 3
4 5 6
7 8 9
0 Enter
Take cash here
Insert deposit envelope here


display menu of withdrawal amounts and option to cancel
input the menu selection
[user canceled transaction]
[user selected an amount]
set amount attribute
get available balance of user's account from database
[amount > available balance]
display appropriate error message
[amount <= available balance]
test whether sufficient cash is available in cash dispenser
[insufficient cash available]
[sufficient cash available]
interact with database to debit amount from user's account
dispense cash
instruct user to take cash

Use Case Diagram:

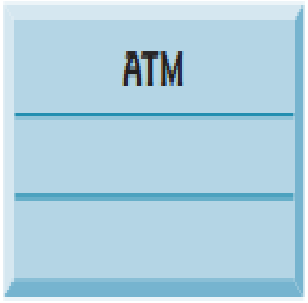
We create a **use case diagram** to model the interactions between a system's clients (in this case study, bank customers) and its use cases. The goal is to show the kinds of interactions users have with a system without providing the details—these are provided in other UML diagrams. Use case diagrams are often accompanied by informal text that gives more detail—like the text that appears in the requirements document.

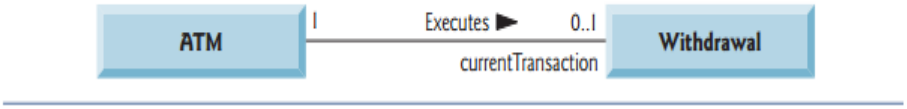


12.3 Identifying the Classes in a Requirements Document:

Modeling Classes:

The UML enables us to model, via **class diagrams**, the classes in the ATM system and their interrelationships. Each class is modeled as a rectangle with three compartments. The top one contains the name of the class centered horizontally in boldface. The middle compartment contains the class's attributes.





Symbol	Meaning
0	None
1	One
m	An integer value
0..1	Zero or one
m, *n*	*m* or *n*
m..*n*	At least *m*, but not more than *n*
*	Any nonnegative integer (zero or more)
0..*	Zero or more (identical to *)
1..*	One or more

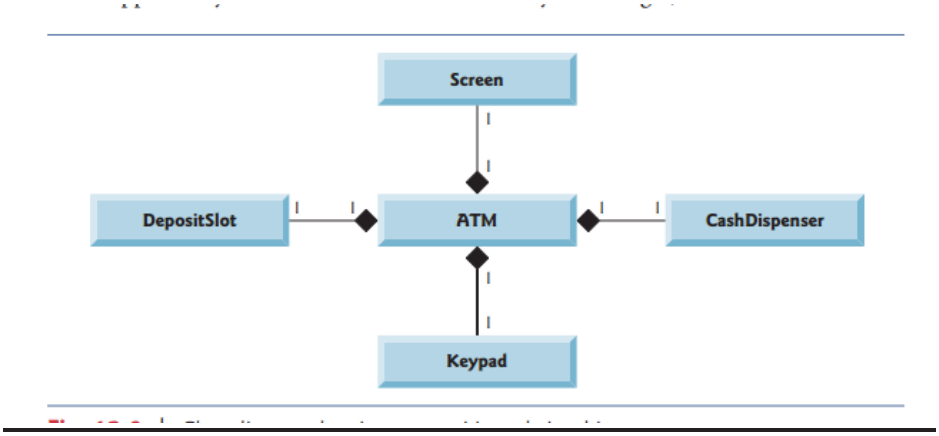
Aggregation:

From **UML specification** composition relationships have properties:

1. Only one class in the relationship can represent the *whole*. For example, either the screen is part of the ATM or the ATM is part of the screen, but the screen and the ATM cannot both represent the whole in the relationship.

2. The *parts* in the composition relationship exist only as long as the whole does, and the whole is responsible for the creation and destruction of its parts.

3. A *part* may belong to only one *whole* at a time, although it may be removed and attached to another whole, which then assumes responsibility for the part.

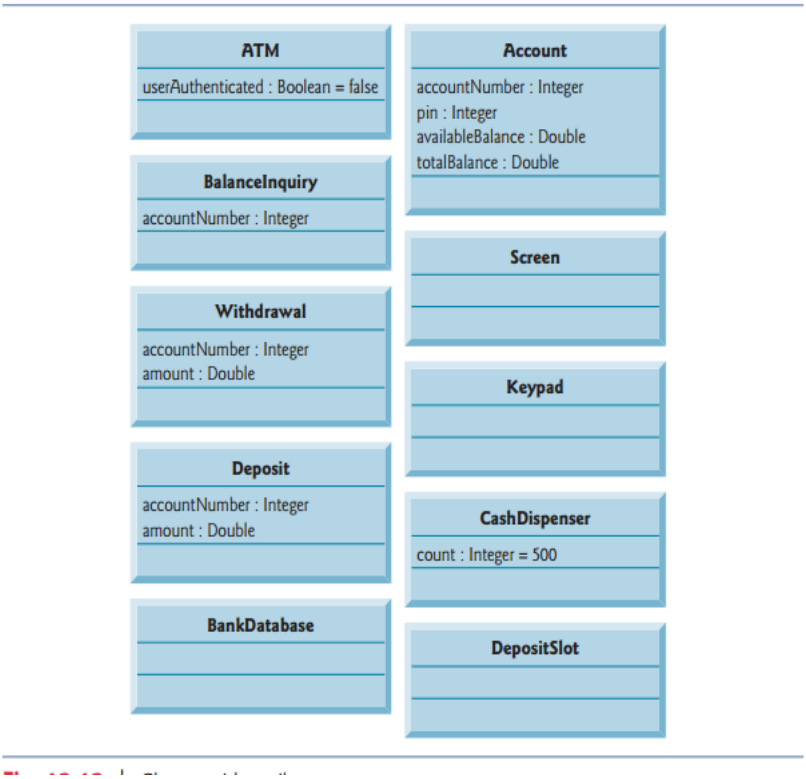


12.4 Identifying Class Attributes:

Identifying Attributes:

Class	Descriptive words and phrases
ATM	user is authenticated
BalanceInquiry	account number
Withdrawal	account number amount
Deposit	account number amount
BankDatabase	*[no descriptive words or phrases]*
Account	account number PIN balance
Screen	*[no descriptive words or phrases]*
Keypad	*[no descriptive words or phrases]*
CashDispenser	begins each day loaded with 500 $20 bills
DepositSlot	*[no descriptive words or phrases]*

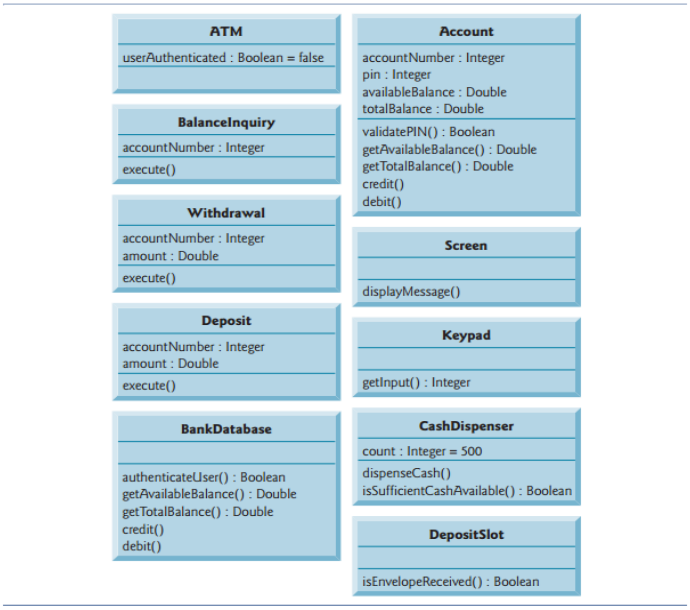
UML Diagram:



12.5 Identifying Class Operations:

Class	Verbs and verb phrases
`ATM`	executes financial transactions
`BalanceInquiry`	*[none in the requirements document]*
`Withdrawal`	*[none in the requirements document]*
`Deposit`	*[none in the requirements document]*
`BankDatabase`	authenticates a user, retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account
`Account`	retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account
`Screen`	displays a message to the user
`Keypad`	receives numeric input from the user
`CashDispenser`	dispenses cash, indicates whether it contains enough cash to satisfy a withdrawal request
`DepositSlot`	receives a deposit envelope

Modeling Operations:

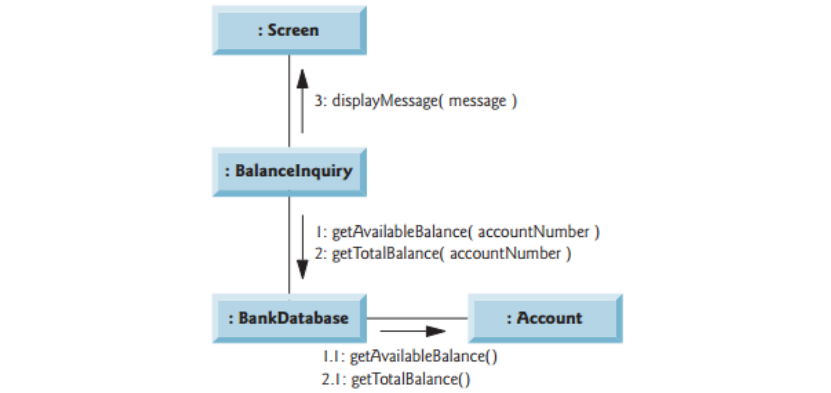


12.7 Indicating Collaboration among Objects:

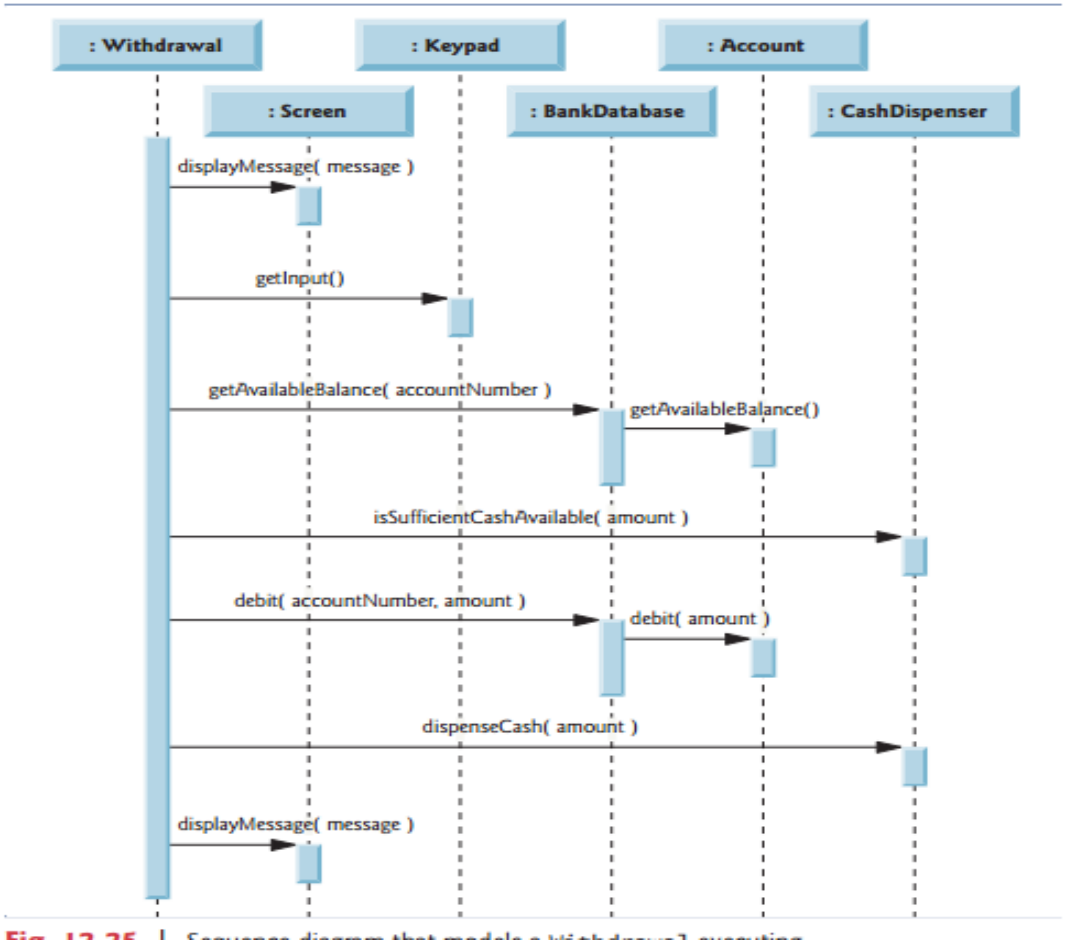
When two objects communicate with each other to accomplish a task, they're said to **collaborate**—objects do this by invoking one another's operations. A **collaboration** consists of an object of one class sending a **message** to an object of another class.

An object of class...	sends the message...	to an object of class...
ATM	displayMessage	Screen
	getInput	Keypad
	authenticateUser	BankDatabase
	execute	BalanceInquiry
	execute	Withdrawal
	execute	Deposit
BalanceInquiry	getAvailableBalance	BankDatabase
	getTotalBalance	BankDatabase
	displayMessage	Screen
Withdrawal	displayMessage	Screen
	getInput	Keypad
	getAvailableBalance	BankDatabase
	isSufficientCashAvailable	CashDispenser
	debit	BankDatabase
	dispenseCash	CashDispenser
Deposit	displayMessage	Screen
	getInput	Keypad
	isEnvelopeReceived	DepositSlot
	credit	BankDatabase
BankDatabase	validatePIN	Account
	getAvailableBalance	Account
	getTotalBalance	Account
	debit	Account
	credit	Account

Sequence of Messages in a Communication Diagram:



Sequence Diagrams:



CHAPTER 13

ATM Case Study Part2

Objectives

In this chapter we'll

- Incorporate inheritance into the design of the ATM.
- Incorporate polymorphism into the design of the ATM.
- Fully implement in Java the UML-based object-oriented design of the ATM software.
- Study a detailed code walkthrough of the ATM software system that explains the implementation issues.

13.1 Introduction

In Chapter 12, we developed an object-oriented design for our ATM system. We now implement our object-oriented design in Java. In Section 13.2, we show how to convert class diagrams to Java code. In Section 13.3, we tune the design with inheritance and polymorphism. Then we present a full Java code implementation of the ATM software in Section 13.4.

13.2 Starting to program the Classes of ATM System

Visibility

We now apply access modifiers to the members of our classes. We've introduced access modifiers `public` and `private`. Access modifiers determine the visibility or accessibility of an object's attributes and methods to other objects. Before we can begin implementing our design, we must consider which attributes and methods of our classes should be `public` and which should be `private`. We've observed that attributes normally should be `private` and that methods invoked by clients of a given class should be `public`. Methods that are called as "utility methods" only by other methods of the same class normally should be `private`. The UML employs visibility markers for modeling the visibility of attributes and operations. Public visibility is indicated by placing a plus sign (+) before an operation or an attribute, whereas a minus sign (–) indicates private visibility. Figure 13.1 shows our updated class diagram with visibility markers included.

Navigability

Before we begin implementing our design in Java, we introduce an additional UML notation. The class diagram in Fig. 13.2 further refines the relationships among classes in the ATM system by adding navigability arrows to the association lines. Navigability arrows (represented as arrows with stick ( ) arrowheads in the class diagram) indicate in the direction which an association can be traversed. When implementing a system designed using the UML, you use navigability arrows to determine which objects need references to other objects. For example, the navigability arrow pointing from class `ATM` to class `BankDatabase` indicates that we can navigate from the former to the latter, thereby enabling the `ATM` to invoke the `BankDatabase`'s operations.

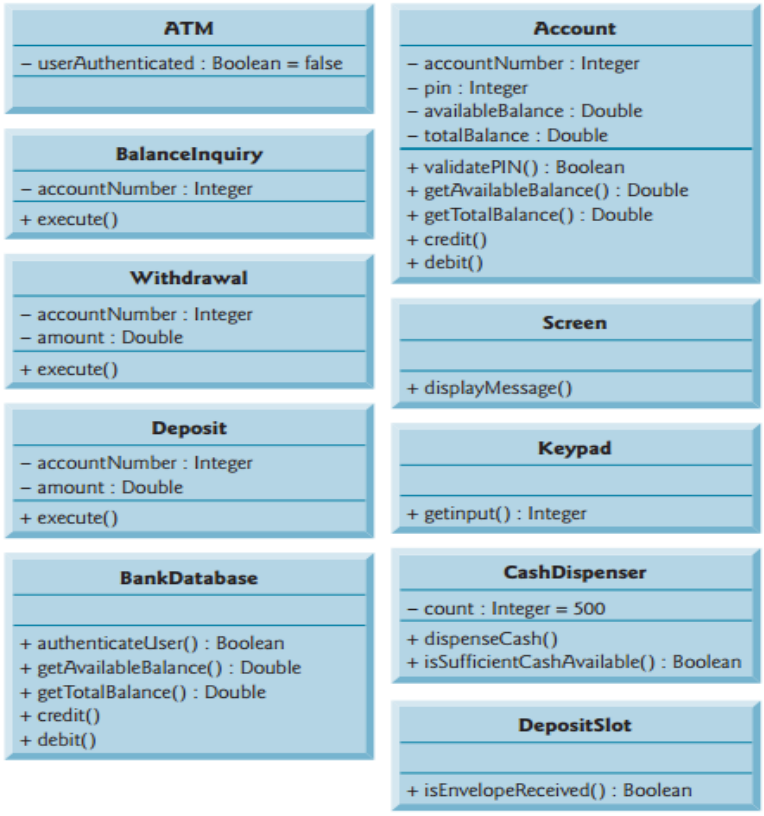


Fig. 13.1 | Class diagram with visibility markers.

However, since Fig. 13.2 does *not* contain a navigability arrow pointing from class BankDatabase to class ATM, the BankDatabase cannot access the ATM's operations. Associations in a class diagram that have navigability arrows at both ends or have none at all indicate bidirectional navigability—navigation can proceed in either direction across the association.

Like the class diagram of Fig. 12.10, that of Fig. 13.2 omits classes BalanceInquiry and Deposit for simplicity. The navigability of the associations in which these classes participate closely parallels that of class Withdrawal. Recall from Section 12.3 that BalanceInquiry has an association with class Screen. We can navigate from class BalanceInquiry to class Screen along this association, but we cannot navigate from class Screen to class BalanceInquiry. Thus, if we were to model class BalanceInquiry in Fig. 13.2, we would place a navigability arrow at class Screen's end of this association. Also recall that class Deposit associates with classes Screen, Keypad and DepositSlot. We can navigate from class Deposit to each of these classes, but *not* vice versa. We therefore would place navigability arrows at the Screen, Keypad and DepositSlot ends of these associations.

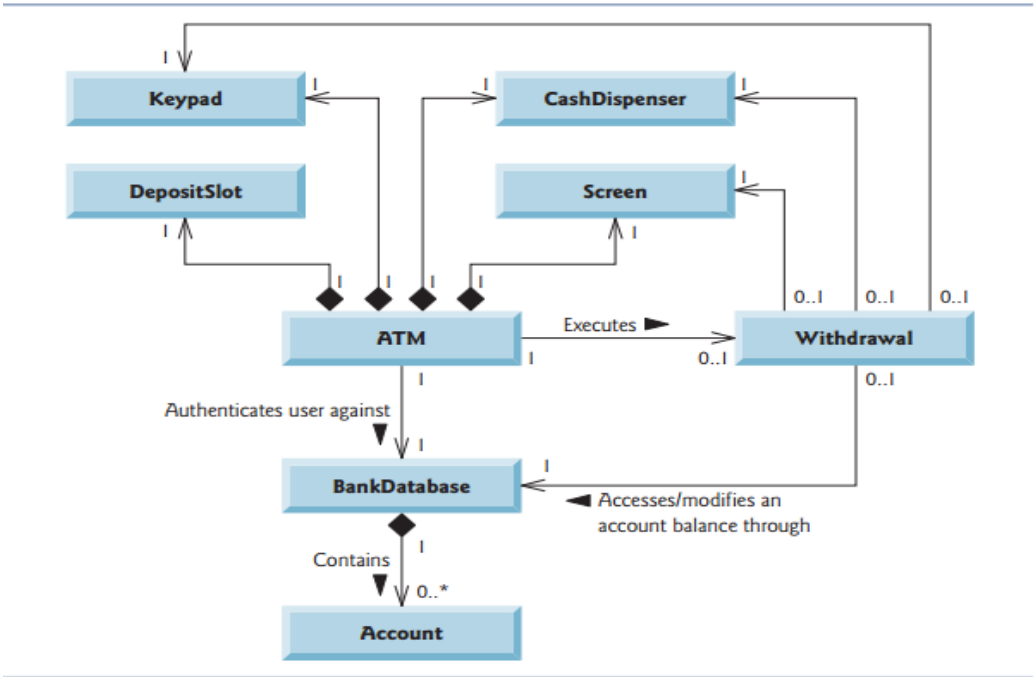


Fig. 13.2 | Class diagram with navigability arrows.

13.3 Incorporating Inheritance and Polymorphism into the ATM System

We now revisit our ATM system design to see how it might benefit from inheritance. To apply inheritance, we first look for *commonality among classes* in the system. We create an inheritance hierarchy to model similar (yet not identical) classes in a more elegant and efficient manner. We then modify our class diagram to incorporate the new inheritance relationships. Finally, we demonstrate how our updated design is translated into Java code.

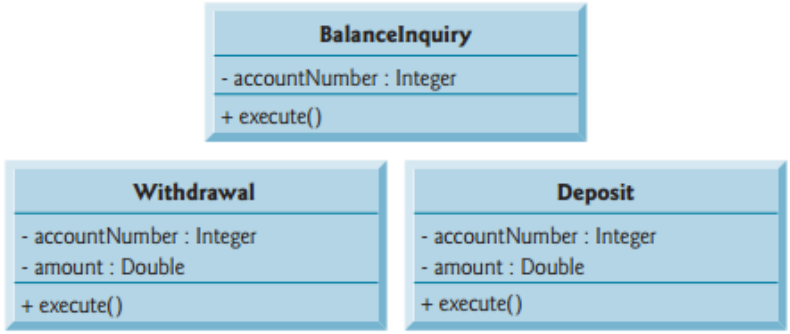


Fig. 13.3 | Attributes and operations of BalanceInquiry, Withdrawal and Deposit.

Generalization

The UML specifies a relationship called a generalization to model inheritance. Figure 13.8 is the class diagram that models the generalization of superclass Transaction

and subclasses BalanceInquiry, Withdrawal and Deposit. The arrows with triangular hollow arrowheads indicate that classes BalanceInquiry, Withdrawal and Deposit extend class Transaction. Class Transaction is said to be a generalization of classes BalanceInquiry, Withdrawal and Deposit. Class BalanceInquiry, Withdrawal and Deposit are said to be specializations of class Transaction.

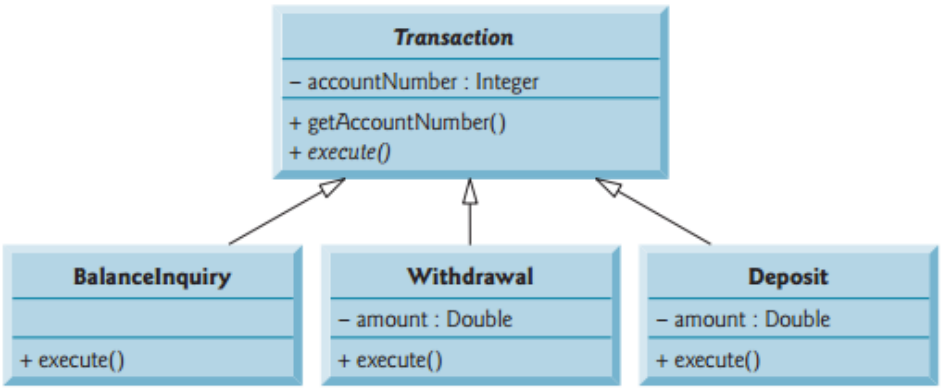


Fig. 13.4 | Class diagram modeling generalization of superclass Transaction and subclasses BalanceInquiry, Withdrawal and Deposit. Abstract class names (e.g., Transaction) and method names (e.g., execute in class Transaction) appear in italics.

*Processing **Transaction**s Polymorphically*

Polymorphism provides the ATM with an elegant way to execute all transactions "in the general." For example, suppose a user chooses to perform a balance inquiry. The ATM sets a Transaction reference to a new BalanceInquiry object. When the ATM uses its Transaction reference to invoke method execute, BalanceInquiry's version of execute is called.

*Class Diagram with **Transaction** Hierarchy Incorporated*

Figure 13.9 presents an updated class diagram of our model that incorporates inheritance and introduces class Transaction. We model an association between class ATM and class Transaction to show that the ATM, at any given moment, either is executing a transaction or is not (i.e., zero or one objects of type Transaction exist in the system at a time). Because a Withdrawal is a type of Transaction, we no longer draw an association line directly between class ATM and class Withdrawal. Subclass Withdrawal inherits superclass Transaction's association with class ATM. Subclasses BalanceInquiry and Deposit inherit this association, too, so the previously omitted associations between ATM and classes BalanceInquiry and Deposit no longer exist either.

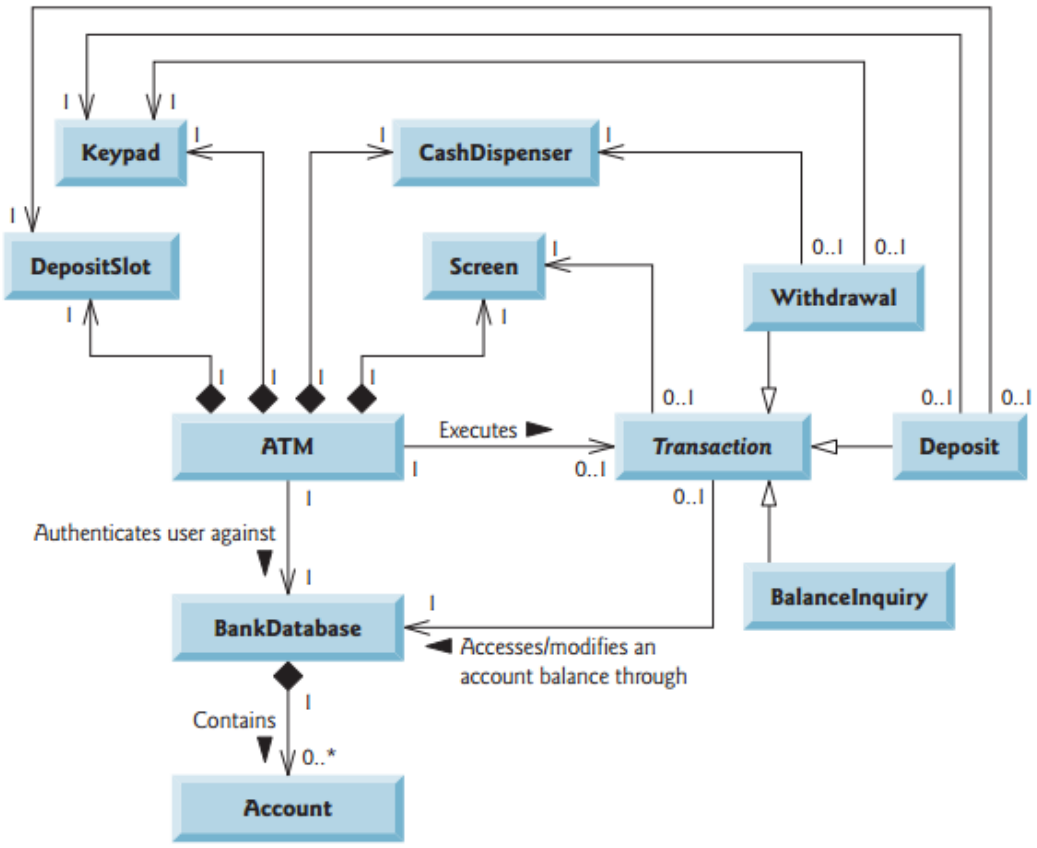


Fig. 13.9 | Class diagram of the ATM system (incorporating inheritance). The abstract class name Transaction appears in italics.

We also add an association between class Transaction and the BankDatabase (Fig. 13.9). All Transactions require a reference to the BankDatabase so they can access and modify account information. Because each Transaction subclass inherits this reference, we no longer model the association between class Withdrawal and the BankDatabase. Similarly, the previously omitted associations between the BankDatabase and classes BalanceInquiry and Deposit no longer exist. We show an association between class Transaction and the Screen. All Transactions display output to the user via the Screen. Thus, we no longer include the association previously modeled between Withdrawal and the Screen, although Withdrawal still participates in associations with the CashDispenser and the Keypad. Our class diagram incorporating inheritance also models Deposit and BalanceInquiry. We show associations between Deposit and both the DepositSlot and the Keypad. Class BalanceInquiry takes part in no associations other than those inherited from class Transaction—a BalanceInquiry needs to interact only with the BankDatabase and with the Screen.

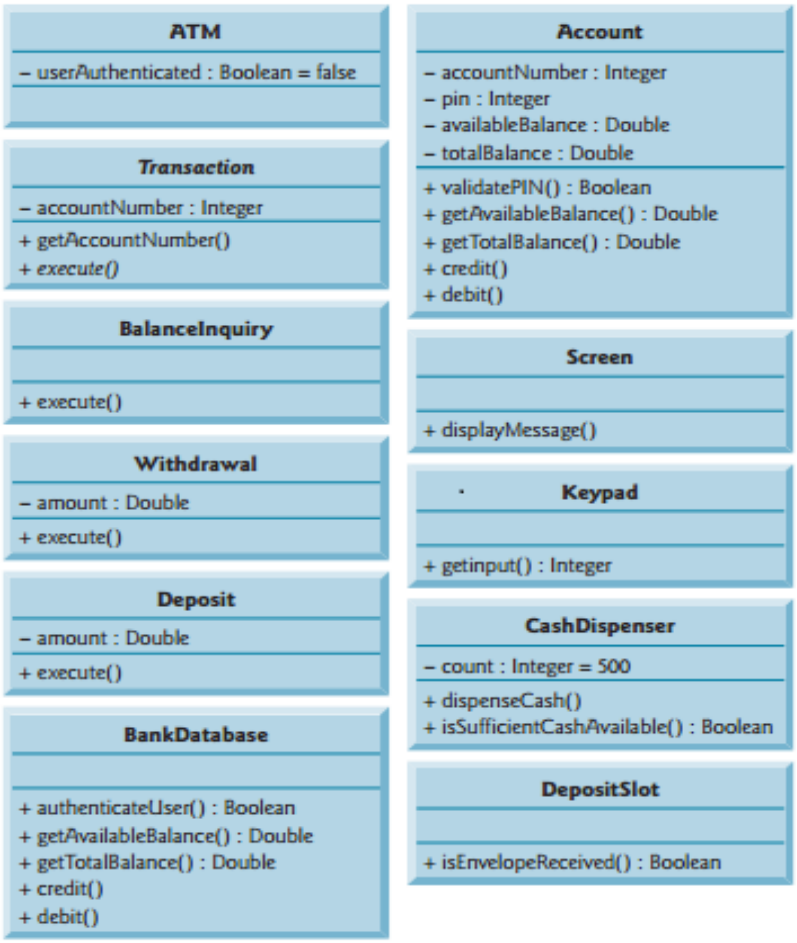


Fig. 13.10 | Class diagram with attributes and operations (incorporating inheritance) The abstract class name Transaction and the abstract method name execute in class Transaction appear in italics.

13.4 ATM Case Study Implementation

This section contains the complete working 673-line implementation of the ATM system. We consider the classes in the order in which we identified them in Section 12.3—ATM, Screen, Keypad, CashDispenser, DepositSlot, Account, BankDatabase, Transaction, BalanceInquiry, Withdrawal and Deposit.

13.4.1 Class ATM

```
// ATM.java
// Represents an automated teller machine

public class ATM
{
private boolean userAuthenticated; // whether user is authenticated
private int currentAccountNumber; // current user's account number
private Screen screen; // ATM's screen
private Keypad keypad; // ATM's keypad
private CashDispenser cashDispenser; // ATM's cash dispenser
private DepositSlot depositSlot; // ATM's deposit slot
private BankDatabase bankDatabase; // account information database

// constants corresponding to main menu options
private static final int BALANCE_INQUIRY = 1;
private static final int WITHDRAWAL = 2;
private static final int DEPOSIT = 3;
private static final int EXIT = 4;

// no-argument ATM constructor initializes instance variables
public ATM()
{
userAuthenticated = false; // user is not authenticated to start
currentAccountNumber = 0; // no current account number to start
screen = new Screen(); // create screen
keypad = new Keypad(); // create keypad
cashDispenser = new CashDispenser(); // create cash dispenser
depositSlot = new DepositSlot(); // create deposit slot
bankDatabase = new BankDatabase(); // create acct info database
} // end no-argument ATM constructor

// start ATM
public void run()
{
// welcome and authenticate user; perform transactions
while ( true )
{
// loop while user is not yet authenticated
while ( !userAuthenticated )
{
screen.displayMessageLine( "\nWelcome!" );
authenticateUser(); // authenticate user
} // end while

performTransactions(); // user is now authenticated
userAuthenticated = false; // reset before next ATM session
currentAccountNumber = 0; // reset before next ATM session
screen.displayMessageLine( "\nThank you! Goodbye!" );
} // end while
} // end method run

// attempts to authenticate user against database
private void authenticateUser()
```

```
{
screen.displayMessage( "\nPlease enter your account number: " );
int accountNumber = keypad.getInput(); // input account number
screen.displayMessage( "\nEnter your PIN: " ); // prompt for PIN
int pin = keypad.getInput(); // input PIN

// set userAuthenticated to boolean value returned by database
userAuthenticated =
bankDatabase.authenticateUser( accountNumber, pin );

// check whether authentication succeeded
if ( userAuthenticated )
{
currentAccountNumber = accountNumber; // save user's account #
} // end if
else
screen.displayMessageLine(
"Invalid account number or PIN. Please try again." );
} // end method authenticateUser

// display the main menu and perform transactions
private void performTransactions()
{
// local variable to store transaction currently being processed
Transaction currentTransaction = null;

boolean userExited = false; // user has not chosen to exit

// loop while user has not chosen option to exit system
while ( !userExited )
{
// show main menu and get user selection
int mainMenuSelection = displayMainMenu();

// decide how to proceed based on user's menu selection
switch ( mainMenuSelection )
{
// user chose to perform one of three transaction types
case BALANCE_INQUIRY:
case WITHDRAWAL:
case DEPOSIT:

// initialize as new object of chosen type
currentTransaction =
createTransaction( mainMenuSelection );
currentTransaction.execute(); // execute transaction
break;
case EXIT: // user chose to terminate session
screen.displayMessageLine( "\nExiting the system..." );
userExited = true; // this ATM session should end
break;
default: // user did not enter an integer from 1-4
screen.displayMessageLine(
"\nYou did not enter a valid selection. Try again." );
break;
```

```
} // end switch
} // end while
} // end method performTransactions

// display the main menu and return an input selection
private int displayMainMenu()
{
screen.displayMessageLine( "\nMain menu:" );
screen.displayMessageLine( "1 - View my balance" );
screen.displayMessageLine( "2 - Withdraw cash" );
screen.displayMessageLine( "3 - Deposit funds" );
screen.displayMessageLine( "4 - Exit\n" );
screen.displayMessage( "Enter a choice: " );
return keypad.getInput(); // return user's selection
} // end method displayMainMenu

// return object of specified Transaction subclass
private Transaction createTransaction( int type )
{
Transaction temp = null; // temporary Transaction variable

// determine which type of Transaction to create
switch ( type )
{
case BALANCE_INQUIRY: // create new BalanceInquiry transaction
temp = new BalanceInquiry(
currentAccountNumber, screen, bankDatabase );
break;
case WITHDRAWAL: // create new Withdrawal transaction
temp = new Withdrawal( currentAccountNumber, screen,
bankDatabase, keypad, cashDispenser );
break;
case DEPOSIT: // create new Deposit transaction
temp = new Deposit( currentAccountNumber, screen,
bankDatabase, keypad, depositSlot );
break;
} // end switch

return temp; // return the newly created object
} // end method createTransaction
} // end class ATM
```

13.4.2 Class Screen

```
// Screen.java
// Represents the screen of the ATM

public class Screen
{
// display a message without a carriage return
public void displayMessage( String message )
{
System.out.print( message );
} // end method displayMessage
```

```
// display a message with a carriage return
public void displayMessageLine( String message )
{
System.out.println( message );
} // end method displayMessageLine

// displays a dollar amount
public void displayDollarAmount( double amount )
{
System.out.printf( "$%,.2f", amount );
} // end method displayDollarAmount
} // end class Screen
```

13.4.3 Class Keypad

```
// Keypad.java
// Represents the keypad of the ATM
import java.util.Scanner; // program uses Scanner to obtain user input

public class Keypad
{
private Scanner input; // reads data from the command line

// no-argument constructor initializes the Scanner
public Keypad()
{
input = new Scanner( System.in );
} // end no-argument Keypad constructor

// return an integer value entered by user
public int getInput()
{
return input.nextInt(); // we assume that user enters an integer
} // end method getInput
} // end class Keypad
```

13.4.4 Class CashDispenser

```
// CashDispenser.java
// Represents the cash dispenser of the ATM
public class CashDispenser
{
// the default initial number of bills in the cash dispenser
private final static int INITIAL_COUNT = 500;
private int count; // number of $20 bills remaining

// no-argument CashDispenser constructor initializes count to default
```

```
public CashDispenser()
{
count = INITIAL_COUNT; // set count attribute to default
} // end CashDispenser constructor

// simulates dispensing of specified amount of cash
public void dispenseCash( int amount )
{
int billsRequired = amount / 20; // number of $20 bills required
count -= billsRequired; // update the count of bills
} // end method dispenseCash

// indicates whether cash dispenser can dispense desired amount
public boolean isSufficientCashAvailable( int amount )
{
int billsRequired = amount / 20; // number of $20 bills required

if ( count >= billsRequired )
return true; // enough bills available
else
return false; // not enough bills available
} // end method isSufficientCashAvailable
} // end class CashDispenser
```

13.4.5 Class DipositSlot

```
// DepositSlot.java
// Represents the deposit slot of the ATM

public class DepositSlot
{
// indicates whether envelope was received (always returns true,
// because this is only a software simulation of a real deposit slot)
public boolean isEnvelopeReceived()
{
return true; // deposit envelope was received
} // end method isEnvelopeReceived
} // end class DepositSlot
```

13.4.6 Class Account

```
// Account.java
// Represents a bank account

public class Account
{
private int accountNumber; // account number
private int pin; // PIN for authentication
private double availableBalance; // funds available for withdrawal
private double totalBalance; // funds available + pending deposits
```

```
// Account constructor initializes attributes
public Account( int theAccountNumber, int thePIN,
double theAvailableBalance, double theTotalBalance )
{
accountNumber = theAccountNumber;
pin = thePIN;
availableBalance = theAvailableBalance;
totalBalance = theTotalBalance;
} // end Account constructor

// determines whether a user-specified PIN matches PIN in Account
public boolean validatePIN( int userPIN )
{
if ( userPIN == pin )
return true;
else
return false;
} // end method validatePIN

// returns available balance
public double getAvailableBalance()
{
return availableBalance;
} // end getAvailableBalance

// returns the total balance
public double getTotalBalance()
{
return totalBalance;
} // end method getTotalBalance

// credits an amount to the account
public void credit( double amount )
{
totalBalance += amount; // add to total balance
} // end method credit

// debits an amount from the account
public void debit( double amount )
{
availableBalance -= amount; // subtract from available balance
totalBalance -= amount; // subtract from total balance
} // end method debit

// returns account number
public int getAccountNumber()
{
return accountNumber;
} // end method getAccountNumber
} // end class Account
```

13.4.7 Class BankDataBase

```
// BankDatabase.java
// Represents the bank account information database

public class BankDatabase
{
private Account[] accounts; // array of Accounts

// no-argument BankDatabase constructor initializes accounts
public BankDatabase()
{
accounts = new Account[ 2 ]; // just 2 accounts for testing
accounts[ 0 ] = new Account( 12345, 54321, 1000.0, 1200.0 );
accounts[ 1 ] = new Account( 98765, 56789, 200.0, 200.0 );
} // end no-argument BankDatabase constructor

// retrieve Account object containing specified account number
private Account getAccount( int accountNumber )
{
// loop through accounts searching for matching account number
for ( Account currentAccount : accounts )
{
// return current account if match found
if ( currentAccount.getAccountNumber() == accountNumber )
return currentAccount;
} // end for

return null; // if no matching account was found, return null
} // end method getAccount

// determine whether user-specified account number and PIN match
// those of an account in the database
public boolean authenticateUser( int userAccountNumber, int userPIN )
{
// attempt to retrieve the account with the account number
Account userAccount = getAccount( userAccountNumber );

// if account exists, return result of Account method validatePIN
if ( userAccount != null )
return userAccount.validatePIN( userPIN );
else
return false; // account number not found, so return false
} // end method authenticateUser

// return available balance of Account with specified account number
public double getAvailableBalance( int userAccountNumber )
{
return getAccount( userAccountNumber ).getAvailableBalance();
} // end method getAvailableBalance

// return total balance of Account with specified account number
public double getTotalBalance( int userAccountNumber )
{
return getAccount( userAccountNumber ).getTotalBalance();
```

```
 } // end method getTotalBalance

 // credit an amount to Account with specified account number
 public void credit( int userAccountNumber, double amount )
 {
 getAccount( userAccountNumber ).credit( amount );
 } // end method credit

 // debit an amount from Account with specified account number
 public void debit( int userAccountNumber, double amount )
 {
 getAccount( userAccountNumber ).debit( amount );
 } // end method debit
 } // end class BankDatabase
```

13.4.8 Class Transaction

```
 // Transaction.java
 // Abstract superclass Transaction represents an ATM transaction

public abstract class Transaction
 {
 private int accountNumber; // indicates account involved
 private Screen screen; // ATM's screen
 private BankDatabase bankDatabase; // account info database

 // Transaction constructor invoked by subclasses using super()
 public Transaction( int userAccountNumber, Screen atmScreen,
 BankDatabase atmBankDatabase )
 {
 accountNumber = userAccountNumber;
 screen = atmScreen;
 bankDatabase = atmBankDatabase;
 } // end Transaction constructor

 // return account number
 public int getAccountNumber()
 {
 return accountNumber;
 } // end method getAccountNumber

 // return reference to screen
 public Screen getScreen()
 {
 return screen;
 } // end method getScreen

 // return reference to bank database
 public BankDatabase getBankDatabase()
 {
 return bankDatabase;
 } // end method getBankDatabase
```

```
 // perform the transaction (overridden by each subclass)
 abstract public void execute();
 } // end class Transaction
```

13.4.9 Class BalanceInquiry

```
// BalanceInquiry.java
 // Represents a balance inquiry ATM transaction

public class BalanceInquiry extends Transaction
 {
 // BalanceInquiry constructor
 public BalanceInquiry( int userAccountNumber, Screen atmScreen,
 BankDatabase atmBankDatabase )
 {
 super( userAccountNumber, atmScreen, atmBankDatabase );
 } // end BalanceInquiry constructor

 // performs the transaction
 @Override
 public void execute()
 {
 // get references to bank database and screen
 BankDatabase bankDatabase = getBankDatabase();
 Screen screen = getScreen();

 // get the available balance for the account involved
 double availableBalance =
 bankDatabase.getAvailableBalance( getAccountNumber() );

 // get the total balance for the account involved
 double totalBalance =
 bankDatabase.getTotalBalance( getAccountNumber() );

 // display the balance information on the screen
 screen.displayMessageLine( "\nBalance Information:" );
 screen.displayMessage( " - Available balance: " );
 screen.displayDollarAmount( availableBalance );
 screen.displayMessage( "\n - Total balance: " );
 screen.displayDollarAmount( totalBalance );
 screen.displayMessageLine( "" );
 } // end method execute
 } // end class BalanceInquiry
```

13.4.10 Class Withdrawal

```
 // Withdrawal.java
 // Represents a withdrawal ATM transaction

public class Withdrawal extends Transaction
 {
 private int amount; // amount to withdraw
```

```
private Keypad keypad; // reference to keypad
private CashDispenser cashDispenser; // reference to cash dispenser

// constant corresponding to menu option to cancel
private final static int CANCELED = 6;

// Withdrawal constructor
public Withdrawal( int userAccountNumber, Screen atmScreen,
BankDatabase atmBankDatabase, Keypad atmKeypad,
CashDispenser atmCashDispenser )
{
// initialize superclass variables
super( userAccountNumber, atmScreen, atmBankDatabase );

// initialize references to keypad and cash dispenser
keypad = atmKeypad;
cashDispenser = atmCashDispenser;
} // end Withdrawal constructor

// perform transaction
@Override
public void execute()
{
boolean cashDispensed = false; // cash was not dispensed yet
double availableBalance; // amount available for withdrawal

// get references to bank database and screen
BankDatabase bankDatabase = getBankDatabase();
Screen screen = getScreen();

// loop until cash is dispensed or the user cancels
do
{
// obtain a chosen withdrawal amount from the user
amount = displayMenuOfAmounts();

// check whether user chose a withdrawal amount or canceled
if ( amount != CANCELED )
{
// get available balance of account involved
availableBalance = bankDatabase.getAvailableBalance( getAccountNumber() );

// check whether the user has enough money in the account
if ( amount <= availableBalance )
{
// check whether the cash dispenser has enough money
if ( cashDispenser.isSufficientCashAvailable( amount ) )
{
// update the account involved to reflect the withdrawal
bankDatabase.debit( getAccountNumber(), amount );

cashDispenser.dispenseCash( amount ); // dispense cash
cashDispensed = true; // cash was dispensed

// instruct user to take cash
```

```
screen.displayMessageLine( "\nYour cash has been" +
" dispensed. Please take your cash now." );
} // end if
else // cash dispenser does not have enough cash
screen.displayMessageLine(
"\nInsufficient cash available in the ATM." +
"\n\nPlease choose a smaller amount." );
} // end if
else // not enough money available in user's account
{
screen.displayMessageLine(
"\nInsufficient funds in your account." +
"\n\nPlease choose a smaller amount." );
} // end else
} // end if
else // user chose cancel menu option
{
screen.displayMessageLine( "\nCanceling transaction..." );
return; // return to main menu because user canceled
} // end else
} while ( !cashDispensed );

} // end method execute

// display a menu of withdrawal amounts and the option to cancel;
// return the chosen amount or 0 if the user chooses to cancel
private int displayMenuOfAmounts()
{
int userChoice = 0; // local variable to store return value

Screen screen = getScreen(); // get screen reference

// array of amounts to correspond to menu numbers
int[] amounts = { 0, 20, 40, 60, 100, 200 };

// loop while no valid choice has been made
while ( userChoice == 0 )
{
// display the withdrawal menu
screen.displayMessageLine( "\nWithdrawal Menu:" );
screen.displayMessageLine( "1 - $20" );
screen.displayMessageLine( "2 - $40" );
screen.displayMessageLine( "3 - $60" );
screen.displayMessageLine( "4 - $100" );
screen.displayMessageLine( "5 - $200" );
screen.displayMessageLine( "6 - Cancel transaction" );
screen.displayMessage( "\nChoose a withdrawal amount: " );

int input = keypad.getInput(); // get user input through keypad

// determine how to proceed based on the input value
switch ( input )
{
case 1: // if the user chose a withdrawal amount
case 2: // (i.e., chose option 1, 2, 3, 4 or 5), return the
```

```
case 3: // corresponding amount from amounts array
case 4:
case 5:
userChoice = amounts[ input ]; // save user's choice
break;
case CANCELED: // the user chose to cancel
userChoice = CANCELED; // save user's choice
break;
default: // the user did not enter a value from 1-6
screen.displayMessageLine(
"\nInvalid selection. Try again." );
} // end switch
} // end while

return userChoice; // return withdrawal amount or CANCELED
} // end method displayMenuOfAmounts
} // end class Withdrawal
```

13.4.11 Class Deposit

```
// Deposit.java
// Represents a deposit ATM transaction

public class Deposit extends Transaction
{
private double amount; // amount to deposit
private Keypad keypad; // reference to keypad
private DepositSlot depositSlot; // reference to deposit slot
private final static int CANCELED = 0; // constant for cancel option

// Deposit constructor
public Deposit( int userAccountNumber, Screen atmScreen,
BankDatabase atmBankDatabase, Keypad atmKeypad,
DepositSlot atmDepositSlot )
{
// initialize superclass variables
super( userAccountNumber, atmScreen, atmBankDatabase );

// initialize references to keypad and deposit slot
keypad = atmKeypad;
depositSlot = atmDepositSlot;
} // end Deposit constructor

// perform transaction
@Override
public void execute()
{
BankDatabase bankDatabase = getBankDatabase(); // get reference
Screen screen = getScreen(); // get reference

amount = promptForDepositAmount(); // get deposit amount from user

// check whether user entered a deposit amount or canceled
```

```
if ( amount != CANCELED )
{
// request deposit envelope containing specified amount
screen.displayMessage(
"\nPlease insert a deposit envelope containing " );
screen.displayDollarAmount( amount );
screen.displayMessageLine( "." );

// receive deposit envelope
boolean envelopeReceived = depositSlot.isEnvelopeReceived();

// check whether deposit envelope was received
if ( envelopeReceived )
{
screen.displayMessageLine( "\nYour envelope has been " +
"received.\nNOTE: The money just deposited will not " +
"be available until we verify the amount of any " +
"enclosed cash and your checks clear." );

// credit account to reflect the deposit
bankDatabase.credit( getAccountNumber(), amount );
} // end if
else // deposit envelope not received
{
screen.displayMessageLine( "\nYou did not insert an " +
"envelope, so the ATM has canceled your transaction." );
} // end else
} // end if
else // user canceled instead of entering amount
{
screen.displayMessageLine( "\nCanceling transaction..." );
} // end else
} // end method execute

// prompt user to enter a deposit amount in cents
private double promptForDepositAmount()
{
Screen screen = getScreen(); // get reference to screen

// display the prompt
screen.displayMessage( "\nPlease enter a deposit amount in " + "CENTS (or 0 to cancel): " );
int input = keypad.getInput(); // receive input of deposit amount

// check whether the user canceled or entered a valid amount
if ( input == CANCELED )
return CANCELED;
else
{
return ( double ) input / 100; // return dollar amount
} // end else
} // end method promptForDepositAmount
} // end class Deposit
```

13.4.12 Class ATMCaseStudy

```
// ATMCaseStudy.java
// Driver program for the ATM case study

public class ATMCaseStudy
{
// main method creates and runs the ATM
public static void main( String[] args )
{
ATM theATM = new ATM();
theATM.run();
} // end main
} // end class ATMCaseStudy
```

CHAPTER 14

GUI Components: Part I

Objectives

In this chapter we'll

- How to use Java's elegant, cross-platform Nimbus look and-feel.
- To build GUIs and handle events generated by user interactions with GUIs.
- To understand the packages containing GUI components, event-handling classes and interfaces.
- To create and manipulate buttons, labels, lists, text fields and panels.
- To handle mouse events and keyboard events.

14.1 Introduction

A graphical user interface (GUI) presents a user-friendly mechanism for interacting with an application. A GUI (pronounced "GOO-ee") gives an application a distinctive "look and feel." GUIs are built from GUI components. These are sometimes called controls or widgets—short for window gadgets. A GUI component is an object with which the user interacts via the mouse, the keyboard or another form of input, such as voice recognition.

Sample GUI:

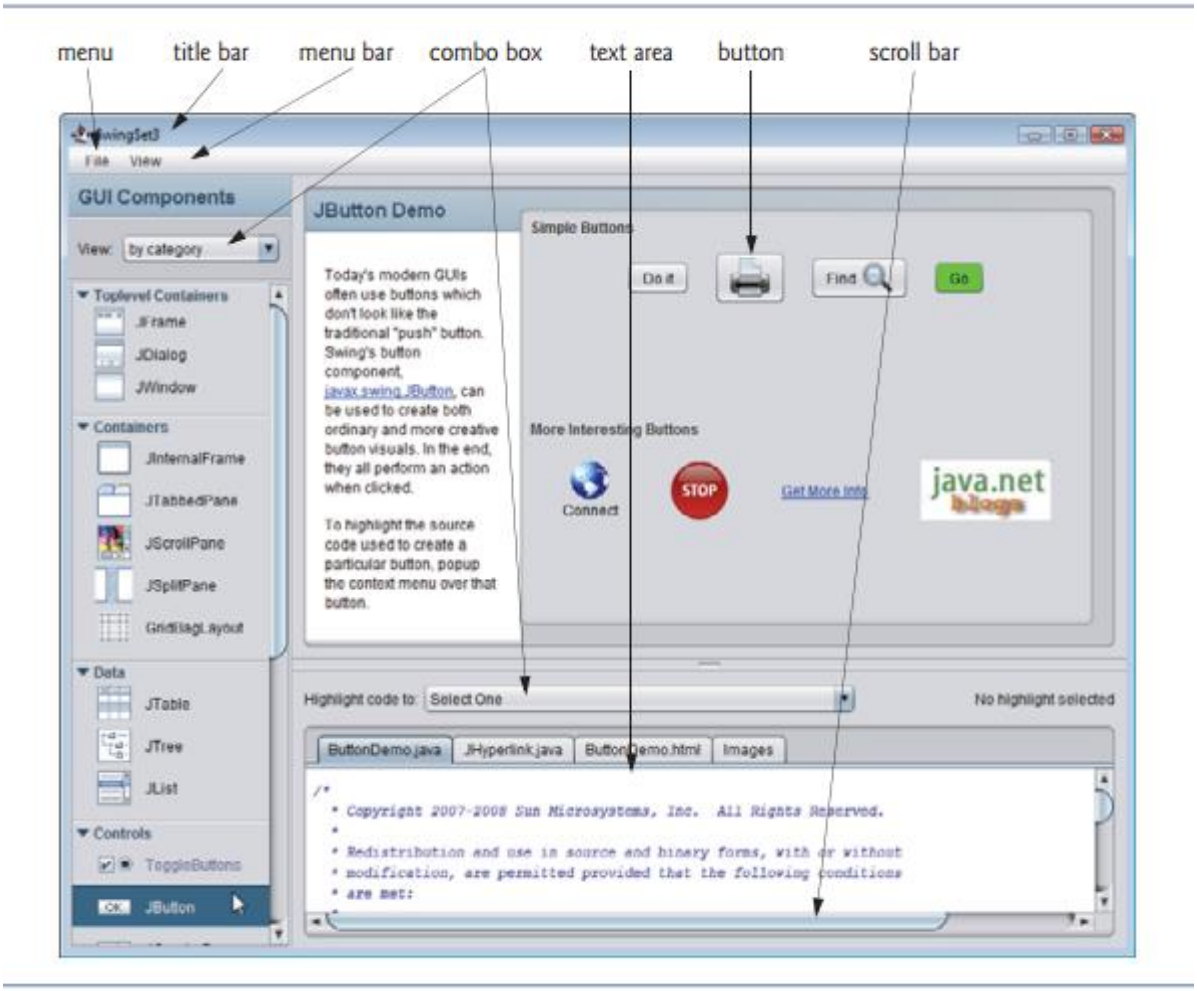


Fig. 14.1 | SwingSet3 application demonstrates many of Java's Swing GUI components.

14.2 Java's New Nimbus Look-and-Feel

In Java SE 6 update 10, Java's elegant, cross-platform look-and-feel known as Nimbus was introduced. For GUI screen captures like Fig. 14.1, we've configured our systems to use Nimbus as the default look-and-feel. There are three ways that you can use Nimbus:

1. Set it as the default for all Java applications that run on your computer.
2. Set it as the look-and-feel at the time that you launch an application by passing a command-line argument to the java command.
3. Set it as the look-and-feel programatically in your application (see Section 25.6). To set Nimbus as the default for all Java applications, you must create a text file named swing.properties in the lib folder of both your JDK installation folder and your JRE installation folder. Place the following line of code in the file:

swing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel

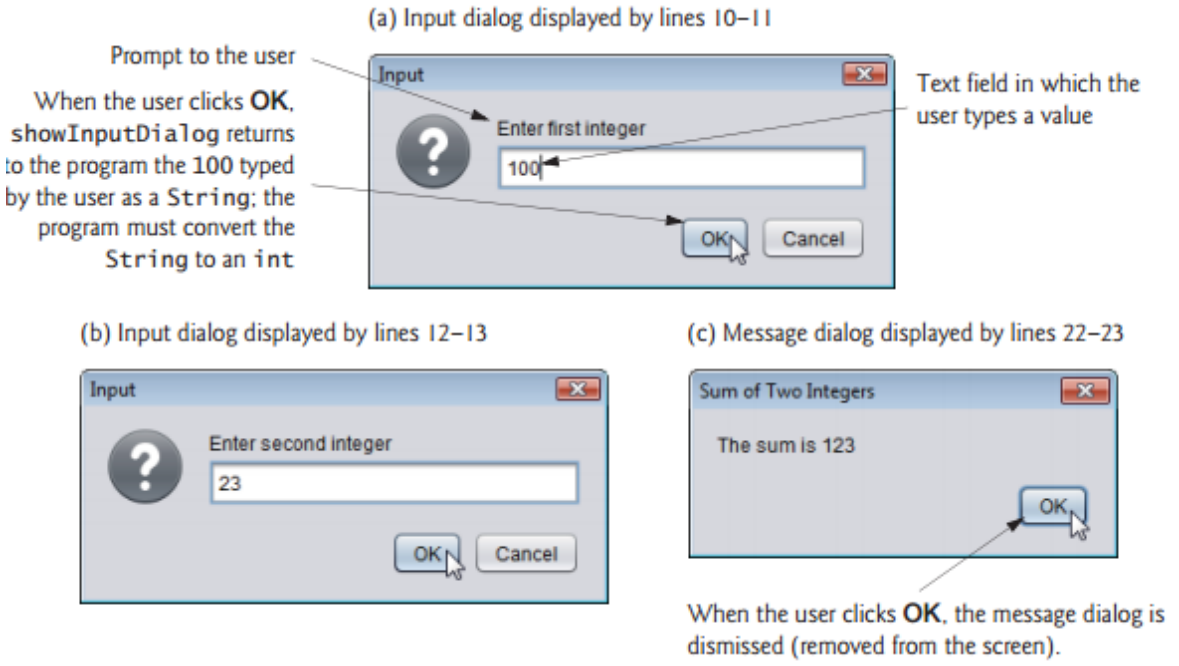
14.3 Simple GUI_Based Input/Output with JOption Pane

Dialog boxes are windows in which programs display important messages to the user or obtain information from the user. Java's **JOptionPane** class (package javax.swing) provides prebuilt dialog boxes for both input and output. These are displayed by invoking static JOptionPane methods. Figure 14.2 presents a simple addition application that uses two input dialogs to obtain integers from the user and a message dialog to display the sum of the integers the user enters.

Implementation:

```
// Fig. 14.2: Addition.java
// Addition program that uses JOptionPane for input and output.
import javax.swing.JOptionPane; // program uses JOptionPane

public class Addition
{
public static void main( String[] args )
{
// obtain user input from JOptionPane input dialogs
String firstNumber =
JOptionPane.showInputDialog( "Enter first integer" );
String secondNumber =
JOptionPane.showInputDialog( "Enter second integer" );

// convert String inputs to int values for use in a calculation
int number1 = Integer.parseInt( firstNumber );
int number2 = Integer.parseInt( secondNumber );

int sum = number1 + number2; // add numbers
// display result in a JOptionPane message dialog
JOptionPane.showMessageDialog( null, "The sum is " + sum,
"Sum of Two Integers", JOptionPane.PLAIN_MESSAGE );
} // end method main
} // end class Addition
```

Output:



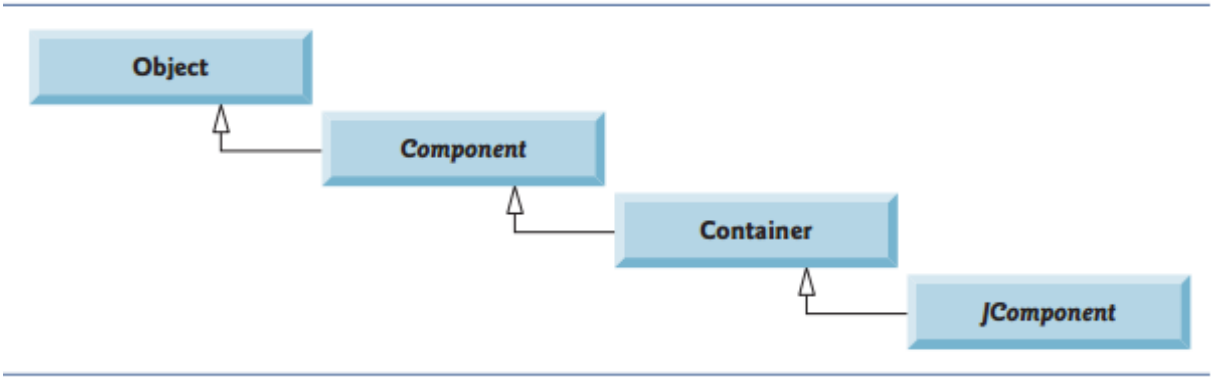
14.4 Overview of Swing Components

Though it's possible to perform input and output using the JOptionPane dialogs, most GUI applications require more elaborate user interfaces. The remainder of this chapter discusses many GUI components that enable application developers to create robust GUIs. The list of several basic Swing GUI components is as follows.

Component	Description
`JLabel`	Displays uneditable text and/or icons.
`JTextField`	Typically receives input from the user.
`JButton`	Triggers an event when clicked with the mouse.
`JCheckBox`	Specifies an option that can be selected or not selected.
`JComboBox`	A drop-down list of items from which the user can make a selection.
`JList`	A list of items from which the user can make a selection by clicking on any one of them. Multiple elements can be selected.
`JPanel`	An area in which components can be placed and organized.

Superclasses of Swing's Lightweight GUI Components:

The UML class diagram below shows an inheritance hierarchy of classes from which lightweight Swing components inherit their common attributes and behaviors.



14.5 Displaying Text and Images in a Window

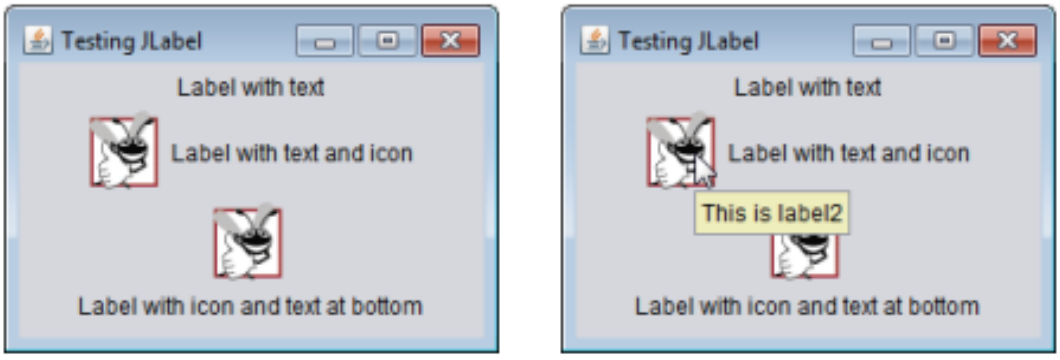
Our next example introduces a framework for building GUI applications. Several concepts in this framework will appear in many of our GUI applications. This is our first example in which the application appears in its own window.

Implementation:

```
// Fig. 14.6: LabelFrame.java
// Demonstrating the JLabel class.
import java.awt.FlowLayout; // specifies how components are arranged
import javax.swing.JFrame; // provides basic window features
import javax.swing.JLabel; // displays text and images
import javax.swing.SwingConstants; // common constants used with Swing
import javax.swing.Icon; // interface used to manipulate images
import javax.swing.ImageIcon; // loads images

public class LabelFrame extends JFrame
{
private JLabel label1; // JLabel with just text
private JLabel label2; // JLabel constructed with text and icon
private JLabel label3; // JLabel with added text and icon

// LabelFrame constructor adds JLabels to JFrame
public LabelFrame()
{
super( "Testing JLabel" );
setLayout( new FlowLayout() ); // set frame layout
// JLabel constructor with a string argument
label1 = new JLabel( "Label with text" );
label1.setToolTipText( "This is label1" );
add( label1 ); // add label1 to JFrame
// JLabel constructor with string, Icon and alignment arguments
Icon bug = new ImageIcon( getClass().getResource( "bug1.png" ) );
label2 = new JLabel( "Label with text and icon", bug,
SwingConstants.LEFT );
label2.setToolTipText( "This is label2" );
```

```
add( label2 ); // add label2 to JFrame
label3 = new JLabel(); // JLabel constructor no arguments
label3.setText( "Label with icon and text at bottom" );
label3.setIcon( bug ); // add icon to JLabel
label3.setHorizontalTextPosition( SwingConstants.CENTER );
label3.setVerticalTextPosition( SwingConstants.BOTTOM );
label3.setToolTipText( "This is label3" );
add( label3 ); // add label3 to JFrame
 } // end LabelFrame constructo
public static void main( String[] args )
{
 LabelFrame labelFrame = new LabelFrame(); // create LabelFrame
 labelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 labelFrame.setSize( 260, 180 ); // set frame size
 labelFrame.setVisible( true ); // display frame
 } // end main
} // end class LabelFrame
```

Output:



14.6 Text Fields and an Introduction to event handling with Nested Classes

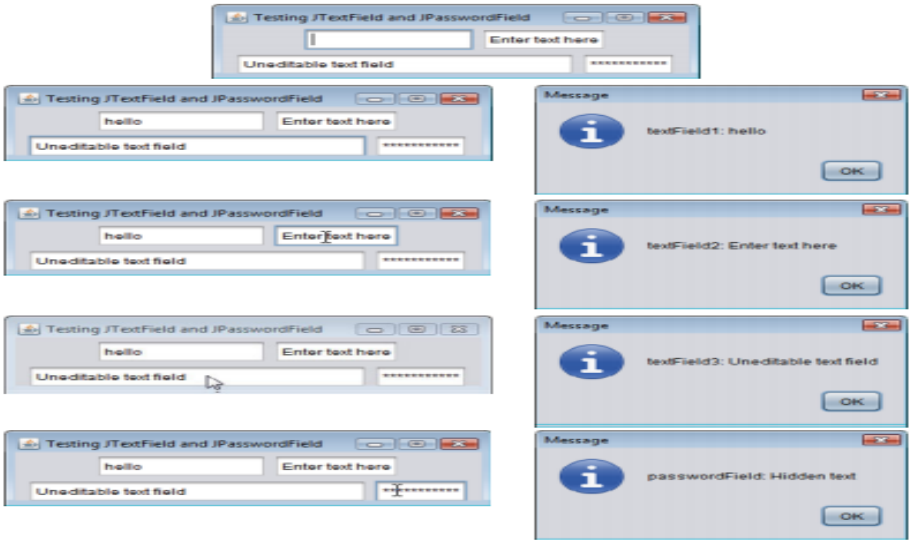
Normally, a user interacts with an application's GUI to indicate the tasks that the application should perform. For example, when you write an e-mail in an e-mail application, clicking the Send button tells the application to send the e-mail to the specified e-mail addresses. GUIs are event driven. When the user interacts with a GUI component, the interaction—known as an event—drives the program to perform a task.

Implementation:

```
// Fig. 14.9: TextFieldFrame.java
// Demonstrating the JTextField class.
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.Jframe;
import javax.swing.JtextField;
import javax.swing.JpasswordField;
import javax.swing.JoptionPane;

public class TextFieldFrame extends Jframe
{
private JtextField textField1; // text field with set size
private JtextField textField2; // text field constructed with text
private JtextField textField3; // text field with text and size
private JpasswordField passwordField; // password field with text

// TextFieldFrame constructor adds JtextFields to Jframe
public TextFieldFrame()
{
super( “Testing JtextField and JpasswordField” );
setLayout( new FlowLayout() ); // set frame layout
// construct textfield with 10 columns
textField1 = new JTextField( 10 );
add( textField1 ); // add textField1 to JFrame
// construct textfield with default text
textField2 = new JTextField( "Enter text here" );
add( textField2 ); // add textField2 to JFrame
// construct textfield with default text and 21 columns
textField3 = new JTextField( "Uneditable text field", 21 );
textField3.setEditable( false ); // disable editing
add( textField3 ); // add textField3 to JFrame
passwordField = new JPasswordField( "Hidden text" );
add( passwordField ); // add passwordField to JFrame
TextFieldHandler handler = new TextFieldHandler();
textField1.addActionListener( handler );
textField2.addActionListener( handler );
textField3.addActionListener( handler );
passwordField.addActionListener( handler );
} // end TextFieldFrame constructor
private class TextFieldHandler implements ActionListener
{
String string = ""; // declare string to display
if ( string = String.format( "textField1: %s", );
// user pressed Enter in JTextField textField2
else if ( string = String.format( "textField2: %s", );

// user pressed Enter in JTextField textField3
else if ( string = String.format( "textField3: %s" );
```

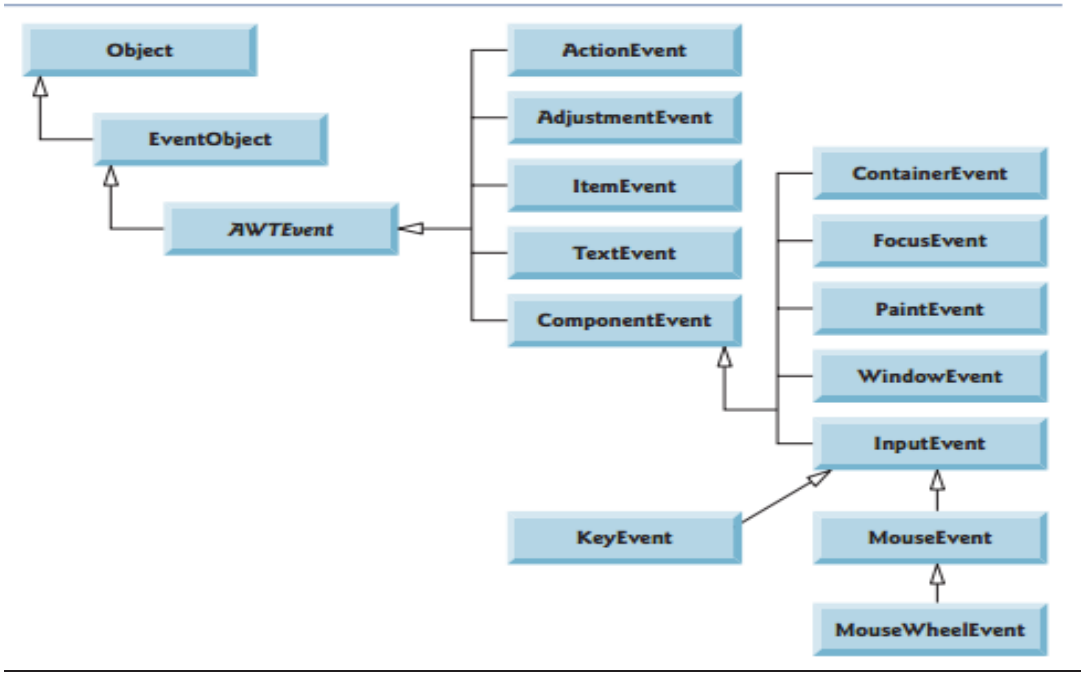
```
 // user pressed Enter in JTextField passwordField
 else if ( )
 string = String.format( "passwordField: %s",
 );

 // display JTextField content
 JOptionPane.showMessageDialog( null, string );
 } // end method actionPerformed
 } // end private inner class TextFieldHandler
public static void main( String[] args )
 {
 TextFieldFrame textFieldFrame = new TextFieldFrame();
 textFieldFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 textFieldFrame.setSize( 350, 100 ); // set frame size
 textFieldFrame.setVisible( true ); // display frame
 } // end main
 } // end class TextFieldFrame
```

Output:

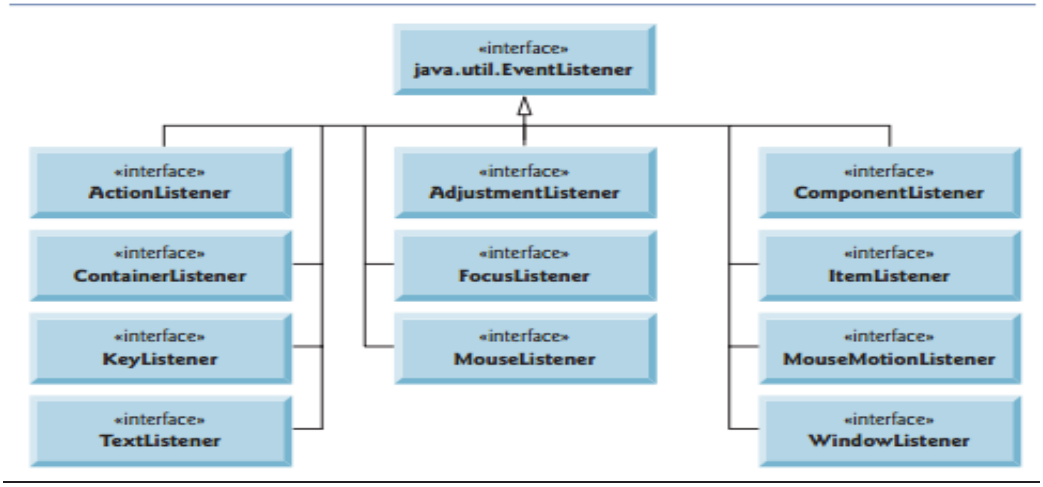


14.7 Common GUI Event Types and Listener Interfaces

Many different types of events can occur when the user interacts with a GUI. The event information is stored in an object of a class that extends AWTEvent (from package java.awt). Figure 14.11 illustrates a hierarchy containing many event classes from the package **java.awt.event**. Some of these are discussed in this chapter and Chapter 25. These event types are used with both AWT and Swing components. Additional event types that are specific to Swing GUI components are declared in package **javax.swing.event**. Let's summarize the three parts to the event-handling mechanism that you saw in Section 14.6—the *event source*, the *event object* and the *event listener*.



Each event-listener interface specifies one or more event-handling methods that *must* be declared in the class that implements the interface. Recall from Section 10.7 that any class which implements an interface must declare *all* the abstract methods of that interface; otherwise, the class is an abstract class and cannot be used to create objects.

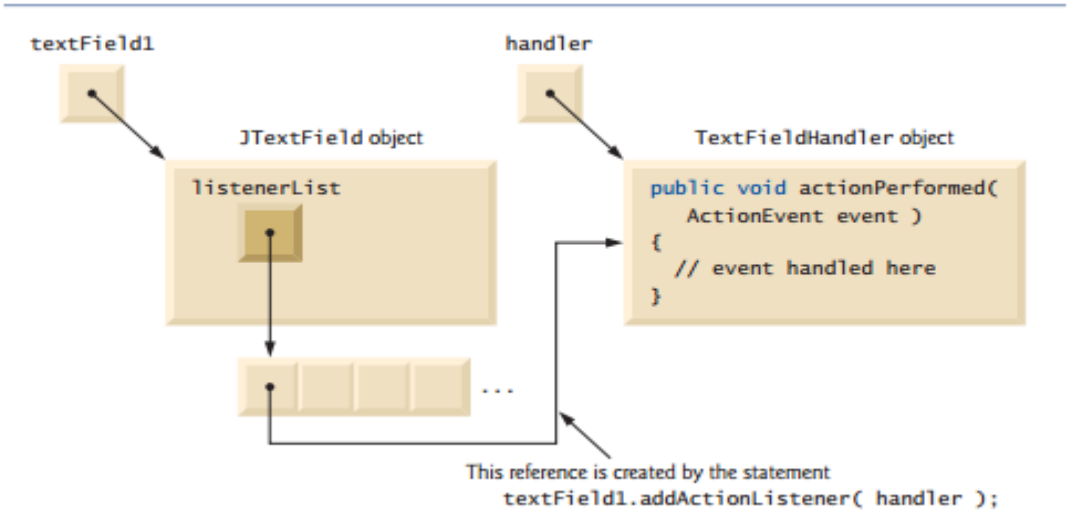


14.8 How Event Handling Works

Registering Events

Every JComponent has an instance variable called listenerList that refers to an object of class **EventListenerList** (package javax.swing.event). Each object of a JComponent subclass maintains references to its registered listeners in the listenerList. For simplicity, we've diagramed listenerList as an

array below the JTextField object is:



Event-Handler Invocation

The event-listener type is important in answering the second question: How does the GUI component know to call actionPerformed rather than another method? Every GUI component supports several *event types*, including mouse events, key events and others. When an event occurs, the event is dispatched only to the *event listeners* of the appropriate type. Dispatching is simply the process by which the GUI component calls an event-handling method on each of its listeners that are registered for the event type that occurred.

14.9 **JButton**

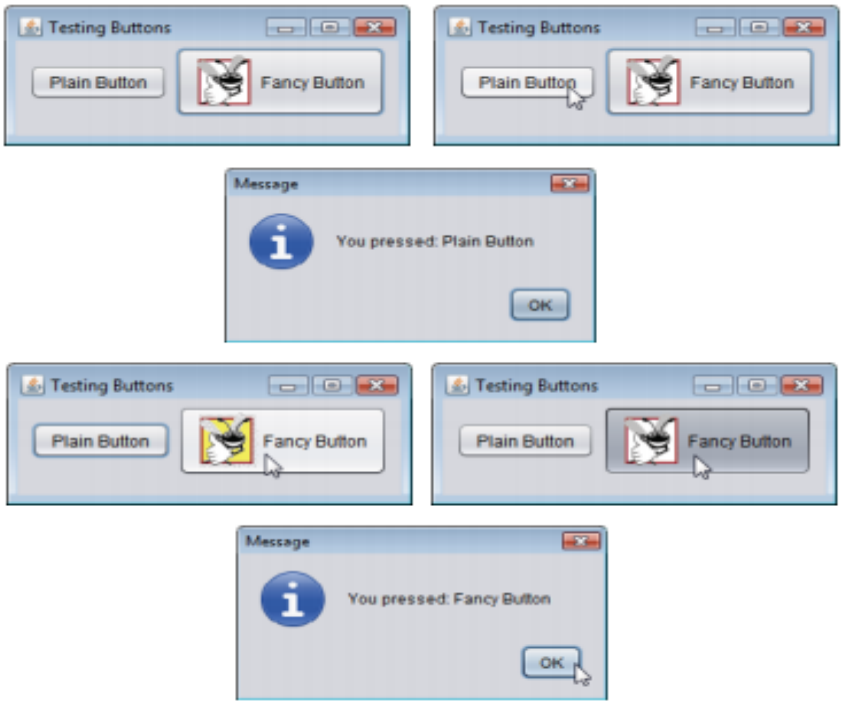
A button is a component the user clicks to trigger a specific action. A Java application can use several types of buttons, including command buttons, checkboxes, toggle buttons and radio buttons. Figure 14.14 shows the inheritance hierarchy of the Swing buttons we cover in this chapter. As you can see, all the button types are subclasses of **AbstractButton** (package javax.swing), which declares the common features of Swing buttons. In this section, we concentrate on buttons that are typically used to initiate a command.

Implementation:

```
// Creating JButtons.
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JOptionPane;

public class ButtonFrame extends JFrame
```

```
 {
private JButton plainJButton; // button with just text
private JButton fancyJButton; // button with icons
 // ButtonFrame adds JButtons to JFrame
 public ButtonFrame()
 {
 super( "Testing Buttons" );
 setLayout( new FlowLayout() ); // set frame layout
  plainJButton = new JButton( "Plain Button" ); // button with text
 add( plainJButton ); // add plainJButton to JFrame
    Icon bug1 = new ImageIcon( getClass().getResource( "bug1.gif" ) );
Icon bug2 = new ImageIcon( getClass().getResource( "bug2.gif" ) );
fancyJButton = new JButton( "Fancy Button", bug1 ); // set image
fancyJButton.setRolloverIcon( bug2 ); // set rollover image
 add( fancyJButton ); // add fancyJButton to JFrame
// create new ButtonHandler for button event handling
ButtonHandler handler = new ButtonHandler();
fancyJButton.addActionListener( handler );
plainJButton.addActionListener( handler );
 } // end ButtonFrame constructor

 // inner class for button event handling
private class ButtonHandler implements ActionListener
 {
 // handle button event
 public void actionPerformed( ActionEvent event )
 {
 JOptionPane.showMessageDialog( , String.format(
 "You pressed: %s", ) );
 } // end method actionPerformed
 } // end private inner class ButtonHandler
public static void main( String[] args )
 {
 ButtonFrame buttonFrame = new ButtonFrame(); // create ButtonFrame
 buttonFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 buttonFrame.setSize( 275, 110 ); // set frame size
 buttonFrame.setVisible( true ); // display frame
 } // end main
 } // end class ButtonFrame
```

Output:



14.11 Conclusion

In this chapter, you learned many GUI components and how to implement event handling. You also learned about nested classes, inner classes and anonymous inner classes. You saw the special relationship between an inner-classobject and an object of its top-level class. You learned how to use JOptionPane dialogs to obtain text input from the user and how to display messages to the user. You also learned how to create applications that execute in their own windows. We discussed class JFrame and components that enable a user to interact with an application.

CHAPTER 15

Graphics and Java 2D

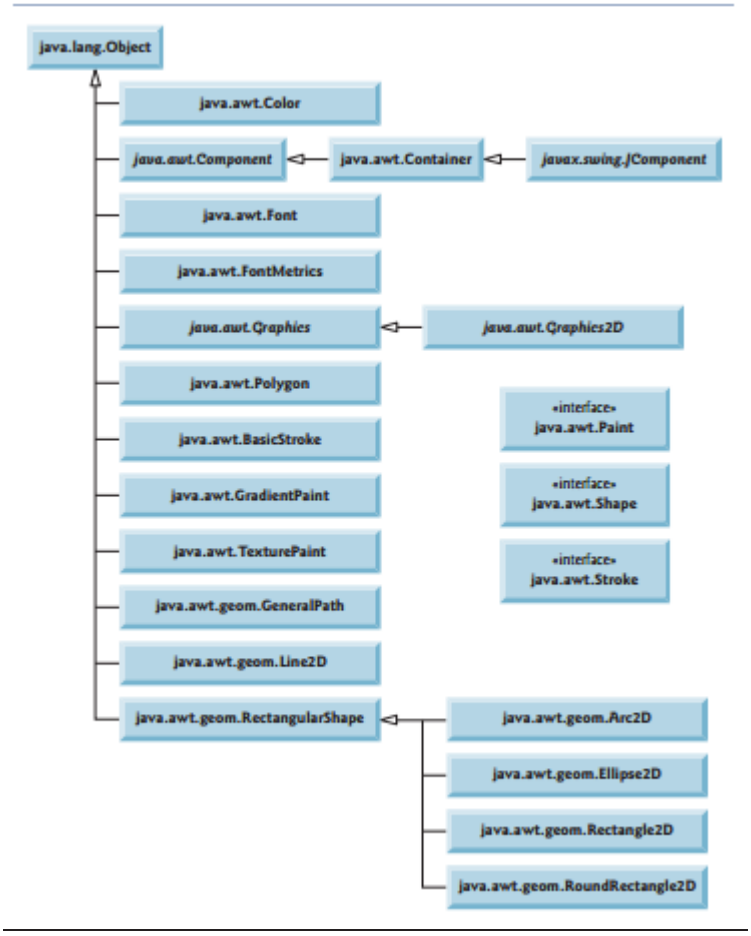
Objectives

In this chapter we'll

- To understand graphics contexts and graphics objects.
- To manipulate colors and fonts.
- To use methods of class Graphics to draw various shapes.
- To use methods of class Graphics2D from the Java 2D API to draw various shapes.

15.1 Introduction

In this chapter, we overview several of Java's capabilities for drawing two-dimensional shapes, controlling colors and controlling fonts. Part of Java's initial appeal was its support for graphics that enabled programmers to visually enhance their applications. Java now contains many more sophisticated drawing capabilities as part of the Java 2D A



15.2 Graphics Contexts and Graphics Objects

A graphics context enables drawing on the screen. A Graphics object manages a graphics context and draws pixels on the screen that represent text and other graphical objects (e.g., lines, ellipses, rectangles and other polygons). Graphics objects contain methods for drawing, font manipulation, color manipulation and the like. This contributes to Java's portability. Because drawing is performed differently on every platform that supports Java, there cannot be only one implementation of the drawing capabilities across all systems. For example, the graphics capabilities that enable a PC running Microsoft Windows to draw a rectangle are different from those that enable a Linux

workstation to draw a rectangle—and they're both different from the graphics capabilities that enable a Macintosh to draw a rectangle. When Java is implemented on each platform, a subclass of Graphics is created that implements the drawing capabilities. This implementation is hidden by class Graphics, which supplies the interface that enables us to use graphics in a platform-independent manner.

15.3 Color Control

Class Color declares methods and constants for manipulating colors in a Java program. The predeclared color constants are summarized here, and several color methods and constructors are summarize

Color constant	RGB value
`public final static Color RED`	255, 0, 0
`public final static Color GREEN`	0, 255, 0
`public final static Color BLUE`	0, 0, 255
`public final static Color ORANGE`	255, 200, 0
`public final static Color PINK`	255, 175, 175
`public final static Color CYAN`	0, 255, 255
`public final static Color MAGENTA`	255, 0, 255
`public final static Color YELLOW`	255, 255, 0
`public final static Color BLACK`	0, 0, 0
`public final static Color WHITE`	255, 255, 255
`public final static Color GRAY`	128, 128, 128
`public final static Color LIGHT_GRAY`	192, 192, 192
`public final static Color DARK_GRAY`	64, 64, 64

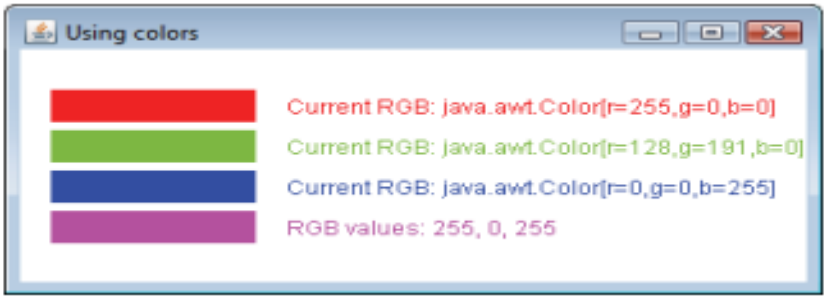
Method	Description
Color constructors and methods	
`public Color( int r, int g, int b )`	Creates a color based on red, green and blue components expressed as integers from 0 to 255.
`public Color( float r, float g, float b )`	Creates a color based on red, green and blue components expressed as floating-point values from 0.0 to 1.0.

Implementation:

```
 //  ColorJPanel.java
 // Demonstrating Colors.
 import java.awt.Graphics;
import java.awt.Color;

import javax.swing.JPanel;

public class ColorJPanel extends JPanel
 {
```

```
 // draw rectangles and Strings in different colors
 public void paintComponent( Graphics g )
 {
 super.paintComponent( g ); // call superclass's paintComponent

 this.setBackground( Color.WHITE );

 // set new drawing color using integers
g.setColor( new Color( 255, 0, 0 ) );
g.fillRect( 15, 25, 100, 20 );
g.drawString( "Current RGB: " + , 130, 40 );

 // set new drawing color using floats
g.setColor( new Color( 0.50f, 0.75f, 0.0f ) );
g.fillRect( 15, 50, 100, 20 );
 g.drawString( "Current RGB: " + , 130, 65 );
g.setColor( Color.BLUE );
g.fillRect( 15, 75, 100, 20 );
 g.drawString( "Current RGB: " + g.getColor(), 130, 90 );
Color color = Color.MAGENTA;
g.setColor( color );
g.fillRect( 15, 100, 100, 20 );
 g.drawString( "RGB values: " + + ", " ++ ", " + , 130, 115 );
}
public static void main( String[] args )
 {
 // create frame for ColorJPanel
 JFrame frame = new JFrame( "Using colors" );
 frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

 ColorJPanel colorJPanel = new ColorJPanel(); // create ColorJPanel
 frame.add( colorJPanel ); // add colorJPanel to frame
 frame.setSize( 400, 180 ); // set frame size
 frame.setVisible( true ); // display frame
 } // end main
}
```

Output:



15.4 **Manipulating Fonts**

This section introduces methods and constants for manipulating fonts. Most font methods and font constants are part of class Font. Some methods of class Font and class Graphics are summarized as

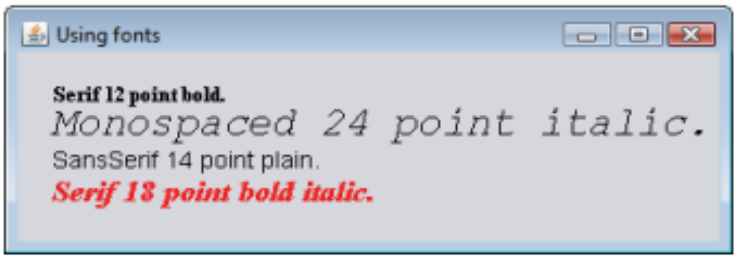
Method or constant	Description
Font constants, constructors and methods	
`public final static int PLAIN`	A constant representing a plain font style.
`public final static int BOLD`	A constant representing a bold font style.
`public final static int ITALIC`	A constant representing an italic font style.
`public Font( String name, int style, int size )`	Creates a `Font` object with the specified font name, style and size.
`public int getStyle()`	Returns an `int` indicating the current font style.
`public int getSize()`	Returns an `int` indicating the current font size.
`public String getName()`	Returns the current font name as a string.
`public String getFamily()`	Returns the font's family name as a string.
`public boolean isPlain()`	Returns `true` if the font is plain, else `false`.
`public boolean isBold()`	Returns `true` if the font is bold, else `false`.
`public boolean isItalic()`	Returns `true` if the font is italic, else `false`.

Implementation:

```
 //  FontJPanel.java
 // Display strings in different fonts and colors.
import java.awt.Font;

import java.awt.Color;
 import java.awt.Graphics;
 import javax.swing.JPanel;

public class FontJPanel extends JPanel
 {
 // display Strings in different fonts and colors
 public void paintComponent( Graphics g )
 {
 super.paintComponent( g ); // call superclass's paintComponent
g.setFont( new Font( "Serif", Font.BOLD, 12 ) );
 g.drawString( "Serif 12 point bold.", 20, 30 );
g.setFont( new Font( "Monospaced", Font.ITALIC, 24 ) );
g.drawString( "Monospaced 24 point italic.", 20, 50 );
g.setFont( new Font( "SansSerif", Font.PLAIN, 14 ) );
g.drawString( "SansSerif 14 point plain.", 20, 70 );
g.setColor( Color.RED );
g.setFont( new Font( "Serif", Font.BOLD + Font.ITALIC, 18 ) );
```

```
g.drawString( + " " + +
 " point bold italic.", 20, 90 );
 } // end method paintComponent
public static void main( String[] args )
 {
 // create frame for FontJPanel
 JFrame frame = new JFrame( "Using fonts" );
 frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 FontJPanel fontJPanel = new FontJPanel(); // create FontJPanel
 frame.add( fontJPanel ); // add fontJPanel to frame
 frame.setSize( 420, 150 ); // set frame size
 frame.setVisible( true ); // display frame
 } // end main
 } // end class Fonts
 } // end class FontJPanel
```

Output:



15.5 **Drawing Lines, Rectangles and Ovals**

This section presents Graphics methods for drawing lines, rectangles and ovals. The methods and their parameters are summarized below. For each drawing method that requires a width and height parameter, the width and height must be nonnegative values. Otherwise, the shape will not display.

Method	Description
`public void drawLine( int x1, int y1, int x2, int y2 )`	Draws a line between the point (x1, y1) and the point (x2, y2).
`public void drawRect( int x, int y, int width, int height )`	Draws a rectangle of the specified `width` and `height`. The rectangle's top-left corner is located at (x, y). Only the outline of the rectangle is drawn using the `Graphics` object's color—the body of the rectangle is not filled with this color.
`public void fillRect( int x, int y, int width, int height )`	Draws a filled rectangle in the current color with the specified `width` and `height`. The rectangle's top-left corner is located at (x, y).
`public void clearRect( int x, int y, int width, int height )`	Draws a filled rectangle with the specified `width` and `height` in the current background color. The rectangle's top-left corner is located at (x, y). This method is useful if you want to remove a portion of an image.
`public void drawRoundRect( int x, int y, int width, int height, int arcWidth, int arcHeight )`	Draws a rectangle with rounded corners in the current color with the specified `width` and `height`. The `arcWidth` and `arcHeight` determine the rounding of the corners (see Fig. 15.20). Only the outline of the shape is drawn.
`public void fillRoundRect( int x, int y, int width, int height, int arcWidth, int arcHeight )`	Draws a filled rectangle in the current color with rounded corners with the specified `width` and `height`. The `arcWidth` and `arcHeight` determine the rounding of the corners (see Fig. 15.20).
`public void draw3DRect( int x, int y, int width, int height, boolean b )`	Draws a three-dimensional rectangle in the current color with the specified `width` and `height`. The rectangle's top-left corner is located at (x, y). The rectangle appears raised when `b` is true and lowered when `b` is false. Only the outline of the shape is drawn.
`public void fill3DRect( int x, int y, int width, int height, boolean b )`	Draws a filled three-dimensional rectangle in the current color with the specified `width` and `height`. The rectangle's top-left corner is located at (x, y). The rectangle appears raised when `b` is true and lowered when `b` is false.
`public void drawOval( int x, int y, int width, int height )`	Draws an oval in the current color with the specified `width` and `height`. The bounding rectangle's top-left corner is located at (x, y). The oval touches all four sides of the bounding rectangle at the center of each side (see Fig. 15.21). Only the outline of the shape is drawn.
`public void fillOval( int x, int y, int width, int height )`	Draws a filled oval in the current color with the specified `width` and `height`. The bounding rectangle's top-left corner is located at (x, y). The oval touches the center of all four sides of the bounding rectangle (see Fig. 15.21).

15.5 Drawing Arcs

An arc is drawn as a portion of an oval. Arc angles are measured in degrees. Arcs sweep (i.e., move along a curve) from a starting angle through the number of degrees specified by their arc angle. The starting angle indicates in degrees where the arc begins. The arc angle specifies the total number of degrees through which the arc sweeps. It illustrates two arcs. The left set of axes shows an arc sweeping from zero degrees to approximately 110 degrees. Arcs that sweep in a counterclockwise direction are measured in positive degrees. The set of axes on the right shows an arc sweeping from zero degrees to approximately –110 degrees. Arcs that sweep in a clockwise direction are measured in negative degrees.

Method	Description
`public void drawArc( int x, int y, int width, int height, int startAngle, int arcAngle )`	Draws an arc relative to the bounding rectangle's top-left x- and y-coordinates with the specified `width` and `height`. The arc segment is drawn starting at `startAngle` and sweeps `arcAngle` degrees.
`public void fillArc( int x, int y, int width, int height, int startAngle, int arcAngle )`	Draws a filled arc (i.e., a sector) relative to the bounding rectangle's top-left x- and y-coordinates with the specified `width` and `height`. The arc segment is drawn starting at `startAngle` and sweeps `arcAngle` degrees.

15.6 Conclusion

In this chapter, you learned how to use Java's graphics capabilities to produce colorful drawings. You learned how to specify the location of an object using Java's coordinate system, and how to draw on a window using the paintComponent method. You were introduced to class Color, and you learned how to use this class to specify different colors using their RGB components.