UNESCO-NIGERIA TECHNICAL & VOCATIONAL
EDUCATION REVITALISATION PROJECT-PHASE II

NATIONAL DIPLOMA IN

COMPUTER TECHNOLOGY

Introduction to Scientific Programming Using Java

COURSE CODE: COM121

THEORY BOOK

Version 1: December 2008

Table of Contents

Week 1: Java Programming Basics I.....

A Brief History of Java 6

Why Java? 6

Types of Java Programs 7

Introduction to Java Applications 7

Components of a Java Application Program 8

Compilation and Execution of Java Programs 12

Week 2: Java Programming Basics II .....

Using Simple Graphical Interface 15

Week 3: .....

Data Types in Java 18

Integers and Floating Points 19

Arithmetic Operators 20

Precedence of Arithmetic Operators 21

Reference (Non-primitive Data Types) 22

Variable Declaration 24

Using Graphical User Interfaces 28

Week 4: Program Development Techniques .....

Program Development Stages 33

Problem solving 33

Week 5: Understand Insatiable Classes.....

Classes, Objects, Methods and Instance Variables 40

Insatiable Classes 41

Declaring a Class with a Method and Instantiating an Object of a Class 42

Class `Circle` 43

Class `CircleTest` 44

Week 6: Introduction to Applets.....

Week 7: Know the Use of Conditional Statements 49

Algorithms 50

Pseudocode 50
Sequence Structure in Java 52
Selection Statements in Java 52
`if` Single-Selection Statement 53
`if`…`else` Double-Selection Statement 53
Conditional Operator (`?:`) 54
Nested `if`…`else` Statements 55
Dangling-`else` Problem 56
Blocks 57
Week 8: Know the Use of Selection Statements .....
The `while` Repetition Statement 60
Formulating Algorithms: Counter-Controlled Repetition 61
Week 9: Recursion .....
Recursive Concepts 65
Example Using Recursion: Factorials 66
Week10: Characters and Strings .....
Fundamentals of Characters and Strings 70
What are Strings? 70
`String` Constructors 71
`String` Methods `length`, `charAt` and `getChars` 72
Comparing Strings 73
General Learning Objectives for Week11: Arrays 79
Week 11: Arrays .....
Declaring and Creating Arrays 81
Examples Using Arrays 83
Using an Array Initializer 84
Calculating a Value to Store in Each Array Element 86
Summing the Elements of an Array 87
Week12: Event Driven Programs .....
Overview of Swing Components 89
Displaying Text and Images in a Window 90

Labeling GUI Components 90
Text Fields and an Introduction to Event Handling with Nested Classes 92
Creating the GUI 96
Steps Required to Set Up Event Handling for a GUI Component 96
Using a Nested Class to Implement an Event Handler 97
Registering the Event Handler for Each Text Field 98
Details of Class `TextFieldHandler`'s `actionPerformed` Method 98
Week14: Inheritance .....
Introduction to Inheritance 101
Superclasses and Subclasses 102
Relationship between Superclasses and Subclasses 104
Creating and Using a `CommissionEmployee` Class 105
CommissionEmployeeBasePlusCommissionEmployee Inheritance Hierarchy Using private Instance Variables 111
Week15: Polymorphism .....
Polymorphism 116
Polymorphism Examples 118
Demonstrating Polymorphic Behavior 119
Abstract Classes and Methods 122
Creating Abstract Superclass `Employee` 125
Creating Concrete Subclass `SalariedEmployee` 128

WEEK 1

General Learning Objectives for Week 1: Java Programming Basics I

Specific Learning Objectives:

a. Brief History of Java
b. Features of Java Programming Language
c. Identify basics of OOP (Object Oriented Programming)
d. Identify the types of Java Programs
e. Identify the components of a Java Program

A Brief History of Java

Java is an Object Oriented Programming language developed by the team of James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems in 1991. This language was initially called "Oak" but was renamed "Java" in 1995. The name Java came about when some Suns people went for a cup of coffee and the name Java was suggested and it struck.

Java was developed out of the rich experiences of the professionals who came together to design the programming language thus, it is an excellent programming language. It has similar syntax to C/C++ programming languages but without it complexities. Java is an elegant programming language.

Java was initially developed for programming intelligent electronic devices such as TVs, cell phones, pagers, smart cards etc. Unfortunately the expectations of the Suns team in this area did not develop as they envisaged. With the advent of the in the boom of the Internet and the World Wide Web (WWW), the team changed their focus and Java was developed for developing web based applications. It is currently being used to develop a variety of applications.

Why Java?

Thousands of programmers are embracing Java as the programming language of choice and several hundred more will joining before the end of the decade. Why is this so? The basic reasons for these are highlighted below:

a. **Portability:** Java is a highly portable programming language because it is not designed for any specific hardware or software platform. Java programs once written are translated into an intermediate form called bytecode. The bytecode is then translated by the Java Virtual Machine (JVM) into the native object code of the processor that the program is been executed on. JVMs exist for several computer platforms; hence the term Write Once Run Anywhere (WORA).

b. **Memory Management:** Java is very conservative with memory; once a resource is no longer referenced the garbage collector is called to reclaim the resource. This is one of the elegant features that distinguishes Java from C/C++ where the programmer has to "manually" reclaim memory.

c. **Extensibility:** The basic unit of Java programs is the class. Every program written in Java is a class that specifies the attributes and behaviors of objects of that class. Java APIs (Application Programmers Interface) contains a rich set reusable classes that is made available to the programmers. These classes are grouped together as packages from which the programmer can build new enhanced classes. One of the key terms of object oriented programming is reuse.
d. **Secure:** Java is a very secure programming language. Java codes (applets) may not access the memory on the local computer that they are downloaded upon. Thus it provides a secure means of developing internet applications.
e. **Simple:** Java's feature makes it a concise programming language that is easy to learn and understand. It is a serious programming language that easily depicts the skill of the programmer.
f. **Robustness:** Java is a strongly typed programming language and encourages the development of error free applications.

Types of Java Programs

Java programs may be developed in three ways. They will be mentioned briefly here:

a. **Java Applications:** These are stand-alone applications such word processors, inventory control systems etc.
b. **Java Applets:** These programs that are executed within a browser. They are executed on the client computer.
c. **Java Serverlets:** These are server side programs that are executed within a browser.

In this course we will limit ourselves to only the first two mentioned types of Java programs – applications and applets.

Introduction to Java Applications

As earlier described Java applications are stand alone programs that can be executed to solve specific problems. Before delving into the details of writing Java applications (and applets) we will consider the concept on which the language is based upon being: Object Oriented Programming (OOP).

Object Oriented Programming is a methodology which has greatly revolutionized how programs are designed and developed as the complexities involved in programming are increasing. The following are the basic principles of OOP.

a. **Encapsulation:** Encapsulation is a methodology that binds together data and the codes that it manipulates thus keeping it safe from external interference and misuse. An object oriented program contains codes that may have private members that are directly accessible to only the members of that program. Also it may have program codes (methods) that will enable other programs to access these data is a uniform and controlled fashion.
b. **Polymorphism:** Polymorphism is a concept whereby a particular "thing" may be employed in many forms and the exact implementation is determined by the specific nature of the situation (or problem). As an example, consider how a frog, lizard and a fish move ("the interface") from one place to another. A frog may leap ten centimeters, a lizard in a single movement moves two centimeters and a shark may swim three meters in a single movement. All these animals exhibit a common ability – movement – expressed differently.
c. **Inheritance:** Inheritance is the process of building new classes based on existing classes. The new class inherits the properties and attributes of the existing class. Object oriented programs models real world concepts of inheritance. For example children inherit attributes and behaviors from their parents. The attributes such as color of eyes, complexion, facial features etc represent the fields in an java. Behaviors such as being a good dancer, having a good sense of humor etc represent the methods. The child may have other attributes and behaviors that differentiate them from the parents.

Components of a Java Application Program

Every Java application program comprises of a class declaration header, fields (instance variables – which is optional), the main method and several other methods as required for solving the problem. The methods and fields are members of the class. In order to explore these components let us write our first Java program.

```
/*
 * HelloWorld.java
 * Displays Hello world!!! to the output window
 *
 */

public class HelloWorld         // class definition header
{

   public static void main( String[] args )
   {
      System.out.println( “Hello World!!! “ );   // print text

   }  // end method main

}  // end class HelloWorld
```

Listing 1.0 HelloWorld.java

The above program is a simple yet complete program containing the basic features of all Java application programs. We will consider each of these features and explain them accordingly.

The first few lines of the program are comments.

```
/*
 * HelloWorld.java
 * Displays Hello world!!! to the output window
 *
 */
```

The comments are enclosed between the /* */ symbols.

Comments are used for documenting a program, that is, for passing across vital information concerning the program – such as the logic being applied, name of the program and any other relevant information etc. Comments are not executed by the computer.

Comments may also be created by using the // symbols either at the beginning of a line:

// This is a comment

Or on the same line after with an executable statement. To do this the comment must be written after the executable statement and not before else the program statement will be ignored by the computer:

System.out.println(“Hello World!!! “);   **// in-line comment.**

This type of comment is termed as an in-line comment.

The rest of the program is the class declaration, starting with the class definition header: **public class HelloWorld**, followed by a pair of opening and closing curly brackets.

```
{

}
```

The class definition header class definition header starts with the access modifier **public** followed by the keyword **class** then the name of the class **HelloWorld**. The access modifier tells the Java compiler that the class can be accessed outside the program file that it is declared in. The keyword **class** tells Java that we want to define a class using the name HelloWorld.

Note: The file containing this class must be saved using the name HelloWorld.java. The name of the file and the class name must be the same both in capitalization and sequence. Java is very case sensitive thus HelloWorld is different from helloworld and also different from HELLOWORLD.

The next part of the program is the declaration of the main method. Methods are used for carrying out the desired tasks in a Java program, they are akin to functions used in C/C++ programming languages. The listing:

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

}
```

is the main method definition header. It starts with the access modifier public, followed by the keyword static which implies that the method **main( )** may be called before an object of the class has been created. The keyword void implies that the method will not return any value on completion of its task. These keywords public, static, and void should always be placed in the sequenced shown.

Any information that you need to pass to a method is received by variables specified within the set of parentheses that follow the name of the method. These variables are called parameters. If no parameters are required for a given method, you still need to include the empty parentheses. In **main( )** there is only one parameter, **String[] args**, which declares a parameter named **args**.

This is an array of objects of type **String**. (*Arrays* are collections of similar objects.) Objects of type **String** store sequences of characters. In this case, **args** receives any command-line arguments present when the program is executed. Note that the parameters could have been written as **String args[]**. This is perfectly correct.
The instructions (statements) enclosed within the curly braces will be executed once the main method is run. The above program contains the instruction that tells Java to display the output "Hello World!!!" followed by a carriage return. This instruction is:
System.out.println("Hello World!!!"); //print text

This line outputs the string "Java drives the Web." followed by a new line on the screen. Output is actually accomplished by the built-in **println( )** method. In this case, **println( )** displays the string which is passed to it. As you will see, **println( )** can be used to display other types of information, too. The line begins with **System.out**. While too complicated to explain in detail at this time, briefly, **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console. Thus, **System.out** is an object that encapsulates console output. The fact that Java uses an object to define console output is further evidence of its object-oriented nature.

As you have probably guessed, console output (and input) is not used frequently in real-world Java programs and applets. Since most modern computing environments are windowed and graphical in nature, console I/O is used mostly for simple utility programs and for demonstration programs. Later you will learn other ways to generate output using Java, but for now, we will continue to use the console I/O methods. Notice that the **println( )** statement ends with a semicolon. All statements in Java end with a semicolon. The reason that the other lines in the program do not end in a semicolon is that they are not, technically, statements.

The first closing brace -}- in the program ends **main( )**, and the last } ends the **HelloWorld** class definition; it is a good practice to place a comment after the closing curly brace. The opening and close brace are referred to as a block of code.
One last point: Java is case sensitive. Forgetting this can cause you serious problems. For example, if you accidentally type **Main** instead of **main**, or **PrintLn** instead of **println**, the

preceding program will be incorrect. Furthermore, although the Java compiler *will* compile classes that do not contain a **main( )** method, it has no way to execute them. So, if you had mistyped **main**, the compiler would still compile your program. However, the Java interpreter would report an error because it would be unable to find the **main( )** method.

In the above program some lines where left blank, this was done in order to make the program readable. Furthermore, tabs (indentation) were used to within the body of a class or methods as appropriate to spate characters and symbols. The blank spaces, tabs, and newline characters are referred to as white spaces.

Compilation and Execution of Java Programs

As earlier mentioned in this text we will create only two types of Java programs – applications and applets. In the next few paragraphs the steps for editing, compiling and executing a Java programs. The procedures for Java application and Java applets are basically the same. The major difference is that Java applets are executed within a browser.

The basic steps for compiling and executing a Java program are:

a. Enter the source code using a text editor. He file must be saved using the file extension .java.

b. Use the Java compiler to convert the source code to its bytecode equivalent. The byte code will be saved in a file having the same name as the program file with an extension .class. To compile our HelloWorld.java program, type the following instructions at the Windows command prompt (c:\>): javac HelloWorld.java

   The bytecodes (.class file) will be created only if there are no compilation errors.

c. Finally use the Java interpreter to execute the application, to do this at the Windows command prompt (c:\>) type: java HelloWorld. (You need not type the .class extension)

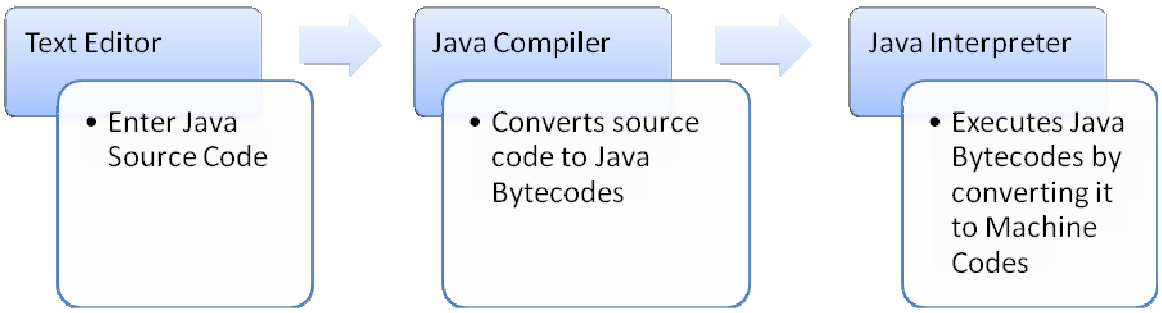


Figure 1.0 Java Compilation and execution process.

Note: Other programs, called *Integrated Development Environments* (IDEs), have been created to support the development of Java programs. IDEs combine an editor, compiler, and other Java support tools into a single application. The specific tools you will use to develop your programs depend on your environment. Examples of IDEs include NetBeans, Eclipse, BlueJ etc.

WEEK 2

General Learning Objectives for Week 2: Java Programming Basics II

Specific Learning Objectives:

Objectives

f. Using Simple Graphical User Interface
g. Apply Graphical Classes

Using Simple Graphical Interface

This week we will employ simple graphical classes – JOptionPane to repeat the same programs which we implemented in week one. In the program presented in week one the output was presented to the windows command prompt.

The JOptionPane class (javax.swing package) enables the user to use its static methods showInputDialog and showMessageDialog to accept data and display information graphically.

The HelloWorldGUI.java which implements JOptionPane static methods for displaying hello world to the user is presented below:

Figure 2.1 HelloWorldGUI.java

```
 1 /*
 2  * HelloWolrdGUI.java
 3  *
 4  */
 5
 6
 7 import javax.swing.JOptionPane;
 8
 9 public class HelloWorldGUI {
10     public static void main(String[] args) {
11         String msg = "Hello Wolrd";
12         String ans ="";
13
14         JOptionPane.showMessageDialog(null, msg );
15
16         // accept the users name
17         ans = JOptionPane.showInputDialog( null, "Enter your Name Please"
);
18
19         // say hello to the user
20         JOptionPane.showMessageDialog(null, "Hello " + ans );
21
22     }  // end method main
23
24 }  // end of class HelloWorldGUI
```

Line 7 we imported he JOptionPane class so that the JVM will ensure that we use it promperly. The class definition header is presented in line 9. This is followed by the main method header which must be mus be written this way it is presented in line 10. Two string variables are used, one for displaying output – msg – and the other for input –ans-. The Graphical message is

displayed with “Hello World” and a command button labeled ok shown. The user is requested to enter his/her name (line 17) and a hello message with the name enter is displayed –(line 20).

The outputs of the program are presented below.

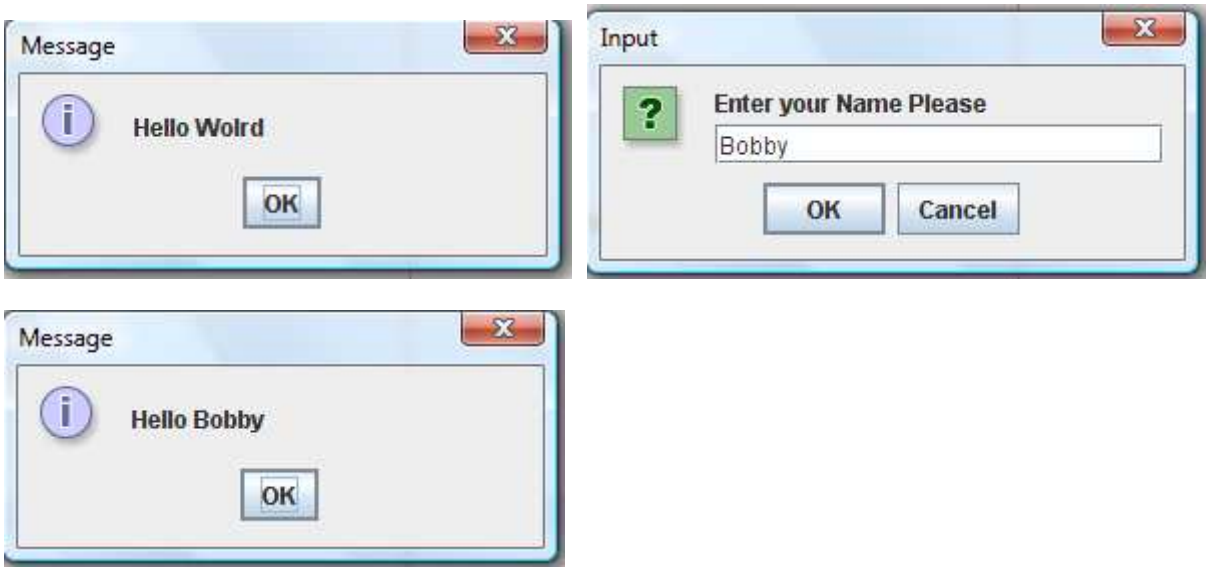


Figure 2.2 sample output of HelloWorldGUI.java program.

WEEK 3

General Learning Objectives for Week 3:

Specific Learning Objectives:

h. Know Java Data Types.
i. Know Java Identifiers and Reserved Words.
j. Know Memory Allocation Concepts.
k. Give the general format of arithmetic expression.
l. Know operator precedence rules.
m. Be able to evaluate simple and complex arithmetic expression.
n. Understand the concept of Data Conversion.

Data Types in Java

A data type defines a set of values and the operations that can be defined on those values. Data types in Java can be divided into two groups:

a. Primitive Data Types

b. Reference Data Types (or Non-Primitives)

Data types are especially important in Java because it is a strongly typed language. This means that all operations are type checked by the compiler for type compatibility. Illegal operations will not be compiled. Thus, strong type checking helps prevent errors and enhances reliability. To enable strong type checking, all variables, expressions, and values have a type. There is no concept of a "type-less" variable, for example. Furthermore, the type of a value determines what operations are allowed on it. An operation allowed on one type might not be allowed on another.

Primitive Data Types

The term primitive is used here to indicate that these types are not objects in an object-oriented sense, but rather, normal binary values. These primitive types are not objects because of efficiency concerns. All of Java's other data types are constructed from these primitive types.

Java strictly specifies a range and behavior for each primitive type, which all implementations of the Java Virtual Machine must support. Because of Java's portability requirement, Java is uncompromising on this account. For example, an **int** is the same in all execution environments. This allows programs to be fully portable. There is no need to rewrite code to fit a specific platform. Although strictly specifying the size of the primitive types may cause a small loss of performance in some environments, it is necessary in order to achieve portability.

There are eight primitive data types in Java: four subsets of integers, two subsets of floating point numbers, a character data type, and a boolean data type. Everything else is represented using objects. Let's examine these eight primitive data types in some detail.

Integers and Floating Points

Java has two basic kinds of numeric values: integers, which have no fractional part, and floating points, which do. There are four integer data types (byte, short, int, and long) and two floating point data types (float and double). All of the numeric types differ by the amount of memory space used to store a value of that type, which determines the range of values that can be represented. The size of each data type is the same for all hardware platforms. All numeric types are *signed*, meaning that both positive and negative values can be stored in them. Figure 3.0 summarizes the numeric primitive types.

Type	Storage	Minimum Value	Maximum Value
byte	8 bits	-128	127
short	16 bits	–32,768	32,767
int	32 bits	–2,147,483,648	2,147,483,647
long	64 bits	–9,223,372,036,854,775,808	9,223,372,036,854,775,807
float	32 bits	Approximately –3.4E+38 with 7 significant digits	Approximately 3.4E+38 with 7 significant digits
double	64 bits	Approximately –1.7E+308 with 15 significant digits	Approximately 1.7E+308 with 15 significant digits

Table 3.0 List of Java's in-built numeric primitive data types.

When designing a program, we sometimes need to be careful about picking variables of appropriate size so that memory space is not wasted. For example, if a value will not vary outside of a range of 1 to 1000, then a two-byte integer (short) is large enough to accommodate it. On the other hand, when it's not clear what the range of a particular variable will be, we should provide a reasonable, even generous, amount of space. In most situations memory space is not a serious restriction, and we can usually afford generous assumptions. Note that even though a float value supports very large (and very small) numbers, it only has seven significant digits. Therefore if it is important to accurately maintain a value such as 50341.2077, we need to use a double.

A *literal* is an explicit data value used in a program. The various numbers used in programs such as Facts and Addition and Piano Keys are all *integer literals*.

Java assumes all integer literals are of type int, unless an L or l is appended to the end of the value to indicate that it should be considered a literal of type long, such as 45L.

Likewise, Java assumes that all *floating point literals* are of type double. If we need to treat a floating point literal as a float, we append an F or f to the end of the value, as in 2.718F or 23.45f. Numeric literals of type double can be followed by a D or d if desired.

The following are examples of numeric variable declarations in Java:

```
int marks = 100;
byte smallNo1, smallNo2;
long totalStars = 86827263927L;
float ratio = 0.2363F;
double mega = 453.523311903;
```

Arithmetic Operators

Arithmetic operators are special symbols for carrying out calculations. These operators enable programmers to write arithmetic expressions. An expression is an algebraic like term that evaluates to a value; it comprises of one or more operands (values) joined together by one or more operators. Below is a summary of Java arithmetic operators in their order of precedence, that is, the order in which the arithmetic expression are evaluated.

Order of Precedence	Operator	Symbol	Algebraic Expression	Java Expression	Association
First	Multiplication	*	a x c	a * c	Left to Right
	Division	/	x/y or x ÷ y or $\frac{x}{y}$	x/y	Left to Right
	Modulus or Remainder	%	w mod 3	w % 3	Left to Right
Second	Addition	+	d+ p	d + p	Right to Left
	Subtraction	-	j – 2	j - 2	Right to Left

Table 3.1 Operators, precedence and association of operators.

Precedence of Arithmetic Operators

The order in which arithmetic operators are applied on data values (operand) is termed rules of operator precedence. These rules are similar to that of algebra. They enable Java to evaluate arithmetic expressions consistently and correctly.

The rules can be summarized thus:

a. Multiplication, division and modulus are applied first. Arithmetic expressions with several of these operators are evaluated from the left to the right.

b. Addition and subtraction are applied next. In the situation that an expression contains several of these operators they are evaluated from right to left.

The order in which the expressions are evaluated is referred to as their association. Now let us consider some examples in the light of the rules of operator precedence; we will list both the algebraic expression and the equivalent java expression.

Algebra : $\frac{x + y + z}{3}$

Java: (x + y + z)/3;

This expression calculates the average of three values. The parentheses is required so that the values represented by x, y and z will be added and the result divided by three. If the parentheses is omitted only z will be divided by three because division has a higher precedence over addition.

Algebra: y = mx + c

Java: y = m * x + c;

In this case the parentheses is not required because multiplication has higher precedence over addition.

Algebra: z = pr % q + w/x – y

Java: z = p * r % q + w/x – y;

In this example, the expression contains the operators *, % followed by +, / and -. The order of execution is listed below:

Note: the order of precedence may be overwritten by using parentheses, that is to say if we desire addition before multiplication or division for example we can include the that part of the expression in parentheses. In the above expression, if x - y is written as (x – y), then the value represented by the y will be subtracted from that of x then the result will be divided by w.

Exercises

Show the order of execution the following arithmetic expressions and write their Java equivalents:

i. $a = \frac{4}{2} - (4 + 1x)$

ii. $c = w - 2d^4 - \frac{k}{f + 2}$

iii. $r = 2c + wx_y^2$

iv. $y = mx + 3b$

v. T= 4r + d mod 4

Reference (Non-primitive Data Types)

Reference or non-primitive type data is used to represent objects. An object is defined by a class, which can be thought of as the data type of the object. The operations that can be

performed on the object are defined by the methods in the class. The attributes or qualities of the objects of a class are defined by the fields – which in essence are primitive type data values. Every object belongs to a class and can be referenced using identifiers. An identifier is a name which is used to identify programming elements such as memory location, names of classes, Java statements and so on. The names used for identifying memory locations are commonly referred to as memory variables or variables for short.

Variable names are created by the programmer for representing values to be stored in the computer memory. Each memory location is associated with a type, a value, and a name. primitive data such as int (integer) can hold a single value and that value must correspond to the data type specified by the programmer. Reference data types on the other hand contain not the objects in memory but the addresses of where the objects (their method and fields etc) are stored in memory. Examples of reference data types include arrays, strings and objects of any class declared by the programmer.

Pertinent data about any object can be gathered and used to represent attributes (fields) and tasks the objects can perform (methods) by using a well defined interface. Once a class has been declared several objects can be created from it. The objects protect their own data and it cannot be directly accessed by other objects.

In the next section we will summarized the rules for creating variables in Java.

a. Variables names may start with and alphabet (a-z / A-Z) and he remaining characters may be, an underscore (_) or a dollar sign, or a number for example sum, counter, firstName, bar2x amount_Paid are all valid variable names. 9x, 0value are invalid.

b. Embedded blank spaces may not be included in variable names though an underscore may be used to join variable names that comprises of compound words. Example x 10 is not valid, it could be written as x_10 or x10.

c. Reserved words (words defined for specific use in Java) may not be employed as variables names. Example loop, do, for, while, switch are reserved.

d. Special symbols such as arithmetic operators, comma (,), ?, /, ! are not allowed.

e. Variable name may be of any length.

It is a good programming practice to use names that indicate the meaning of the value it represents. For amountPaid, or amt_paid can be easily remembered that it represent an amount paid value. Though if the programmer had used x or y as the variable name it would still have been valid.

Java is a case sensitive programming language thus the programmer must be very careful and consistent when giving and using variable names. Java distinguishes between upper case (capital) letters and lower case letters (small letters) hence 'a' is different from 'A' as far as Java is concerned.

Exercises: Study the variable names below and indicate whether they are valid or not. If invalid give reasons.

a. &maxium _____

b. X _____

c. Absolute temperature _____

d. Money _____

e. 30yearold _____

f. initVolume _____

Variable Declaration

Variable may represent values that are expected to change or not during the execution of a computer program. When declaring variable names the scope (visibility) of variables from other part of the program may be specified, the type of data that should be stored in the area of memory.

Variable names may also represent either primitive data or reference data. The general form for creating or declaring variable is presented below.

accessModifier dataType variableList

Where:

accessModifier determines the visibility of the variable to other part of the program e.g. public or private.

dataType represents either primitive (int, char, float) or reference type data (array, String).
variableList is one or more valid variables separated using commas.
Examples:

a. private int x, y, z;

In this example the access modifier is private, the data type is int and the variables that are permitted to hold integer values are the identifiers x, y and z. similar pattern is applied in other examples below.

b. private float balance, initTemperature;

c. private boolean alreadyPaid;

d. public long population;

Alternatively the initial values to be stored in the memory may be specified when declaring the variables. The general form for declaring and initializing the variables is presented below:

accessModifier dataType variable1= value1, variable2 = value2, … , variable = valuen;
We will illustrate with examples.

```
private int x = 10;
private int a = 0, b= 0, c = 0;
public double amountLoaned = 100;
```

Note: we declaring variables in a method (e.g. main method) do not include the access modifiers because all variables declared in methods are implicitly private hence localized to that method. The examples given above can be used to create instance variables.

Now let us write a program to put together all that we have learnt. The program listing below demonstrates how to create primitive variables (int) and non-primitive variable (of type Scanner). It demonstrates how to write arithmetic expressions, input and output data from the user.

```
/*
 * AddTwoNo.java
 * Add any two integer numbers
 */
```

```
import java.util.Scanner;

public class AddTwoNo {

   public static void main(String[] args) {
      // declare primitive variables
      int firstNumber = 10;
      int secondNumber = 20;
      int sum = 0;

      sum = firstNumber + secondNumber;

      system.out.println( "Sum of " + firstNumber + " and " +
            secondNumber + " is " + sum );

      // declare reference variable of type Scanner
      Scanner input = new Scanner( System.in );

      // Accept values from the user
      System.out.println( "Enter first integer number please ");
      firstNumber = input.nextInt();

      System.out.println( "Enter second integer number please ");
      secondNumber = input.nextInt();

      // calculate sum and display result
      sum = firstNumber + secondNumber;

      System.out.println( "Sum of " + firstNumber + " and " +
            secondNumber + " is " + sum );

   }  // end of method main

}  // end of class AddTwoNos
```

Listing 3.1 – AddTwoNos.java

Now let us dissect the program. We will only pay particular attention to parts of the program that were introduced.

After the main comments at the beginning of the program, just before the class declaration we have the statement import statement. This statement is always placed before the class declaration:

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

This statement instructs Java to include the Scanner as part of our program so that it will be able to ensure that we use the elements of this class properly. A collection of classes are grouped together in Java for ease of usage and to facilitate software reuse. A collection of classes grouped together are referred to as a package. The Scanner class is a member of the java.util package. By importing classed and using them in our classes makes Java a robust programming language.

Next is the class definition header, then the main method definition header both of which we have discussed earlier. Within the main method three local variables of type int (integer) are declared. It is advisable to declare individual variables on separate tine as this enhances readability, a plus during debugging.

```
    // declare primitive variables
    int firstNumber = 10;
    int secondNumber = 20;
    int sum = 0;
```

The next instruction is an arithmetic expression that calculates the sum of the first and second numbers and assigns the resulting values to sum.

```
sum = firstNumber + secondNumber;
```

In order for our program to permit the user to enter values via the standard input stream using an object the Scanner class (input).

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

Let us tarry a little while and study this statement in depth. The first part of the expression: Scanner input; declares an object reference input of class Scanner; the second part of the expression instructs Java to create the object in memory using the new keyword that calls a special method (constructor) to initialize fields of the class to initial values. This single statement may be broken down into two:

Scanner input; // declares the variable

input = new Scanner(System.in); // creates an instance (object) of the Scanner class.

Before the user enters any value, he is prompted accordingly; this is achieved through the use of the println statement. The first and second println statements prompts the user for the first and second integer numbers respectively.

The statement

firstNumber = input.nextInt();

instructs the computer to read an integer value from the keyboard. To achieve this, the nextInt() method of the Scanner class was invoked using an object reference input.

After accepting the integer numbers from the user the sum is calculated as before and then displayed to the output window.

Using Graphical User Interfaces

The entire program we have written so far has inputted or displayed prompts to the user via the output window. In most real world programs, this will not be the case. Most modern applications display messages to the user using windows – dialog boxes – and use same to accept data from the user. In our next example, we will improve on the addition program by using dialog boxes.

```
/*
 * AddTwoNoDialog.java
 * Add any two integer numbers
 */

import javax.swing.JOptionPane;

public class AddTwoNoDialog {

    public static void main(String[] args) {
        // declare primitive variables
        int firstNumber = 10;
        int secondNumber = 20;
        int sum = 0;
```

```
        String input;   // for accepting input from the user
        String output;  // for displaying output

        // Accept values from the user
        input = JOptionPane.showInputDialog( null, "Enter first integer number",
                "Adding Integers", JOptionPane.QUESTION_MESSAGE );

        firstNumber = Integer.parseInt( input );

        input = JOptionPane.showInputDialog(null, "Enter second integer number",
                "Adding Integers", JOptionPane.QUESTION_MESSAGE );

        secondNumber = Integer.parseInt( input );

        // calculate sum and display result
        sum = firstNumber + secondNumber;

        // build output string
        output = "Sum of " + firstNumber + " and " + secondNumber + " is "
                + sum;

        // display output
        JOptionPane.showMessageDialog( null, output, "Adding two integers",
                JOptionPane.INFORMATION_MESSAGE );


    }  // end of method main

}  // end of class AddTwoNos
```

Listing 3.2 AddTwoNoDialog.java

In this program, we imported the JOptionPane class (package javax.swing). JOptionPane contains several static methods and static fields, some of which was implemented in the listing 3.2.

Let us go through the program and see how it works. We will only emphasize new concepts that were introduced. In the program we imported the JOptionPane class (javax.swing package). This will ensure Java loads the class and that we make use of the features of the class properly. This class enables us to create objects that displays dialog boxes for both input and output.

Two string variables for handling both input and output were declared using the statements:

```
        String input;   // for accepting input from the user
```

```
String output;  // for displaying output
```

To accept the input from the user we employed the static method **showInputDialog()** of the class JOptionPane. The code is represented below:

```
// Accept values from the user
input = JOptionPane.showInputDialog( null, "Enter first integer number",
    "Adding Integers", JOptionPane.QUESTION_MESSAGE );
```

showInputDialog method always returns a string value thus we have to assign it return value to the string variable input. The first parameter has a null value which implies that the dialog box will be displayed in the middle of the computer screen. The next parameter is the prompt-message- in this case “Enter first integer number”, followed by the title that will be displayed as the title of the dialog box see figures 3.1a and 3.1b. the final parameter specifies the icon to be displayed.

Each value entered by the user and assigned to the variable input is the wrapped into an integer using the wrapper class Integer. This is one of the wrapper classes that is used to convert primitives to their object equivalent and vice versa were entered by the user is assigned

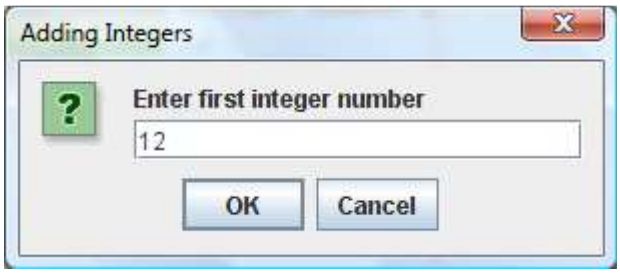


Figure 3.1a

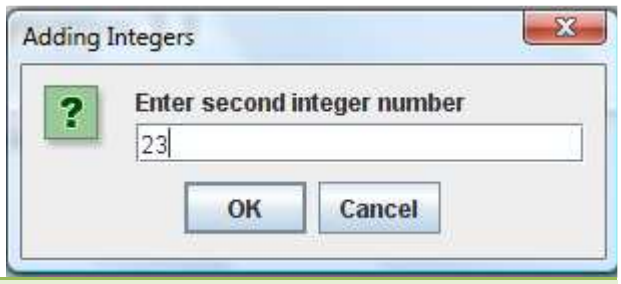


Figure 3.1b

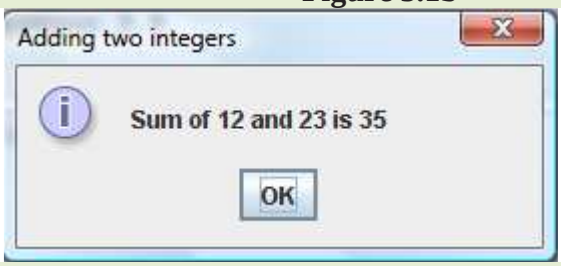


Figure 3.3c

```
// display output
JOptionPane.showMessageDialog( null, output, "Adding two integers",
    JOptionPane.INFORMATION_MESSAGE );
```

WEEK 4

General Learning Objectives for Week 4: Program Development Techniques

Specific Learning Objectives:

o. Understand the concept developmental techniques of program development.
p. Know how to input and output data using graphical user interfaces
q. Apply arithmetic operators in manipulating input data

Program Development Stages

Programming in any programming language is not a task that should be trivialized as mere entry of code into the computer. In order to develop robust and efficient applications the developer/programmer must follow certain steps. Most beginning programmers simply start keying in code, for simple applications this may be ok, for most real life applications that may include hundreds of classes and codes spanning thousands of line such an approach to programming is as good as a Mission Impossible.

In this chapter we will introduce the basic steps that are to be employed when developing a Java program. Programming basically involves problem solving – that is we write programs to efficiently and effectively meet the needs of the users.

Problem solving

The purpose of writing a program is to solve a problem. Problem solving, in general, consists of multiple steps:

a. Understanding the problem.
b. Breaking the problem into manageable pieces.
c. Designing a solution.
d. Considering alternatives to the solution and refining the solution.
e. Implementing the solution.
f. Testing the solution and fixing any problems that exist.

The first step, understanding the problem, may sound obvious, but a lack of attention to this step has been the cause of many misguided efforts. If we attempt to solve a problem we don't completely understand, we often end up solving the wrong problem or at least going off on improper tangents. We must understand the needs of the people who will use the solution. These needs often include subtle nuances that will affect our overall approach to the solution.

After we thoroughly understand the problem, we then break the problem into manageable pieces and design a solution. These steps go hand in hand. A solution to any problem can rarely be expressed as one big activity. Instead, it is a series of small cooperating tasks that interact to perform a larger task. When developing software, we don't write one big program. We design separate pieces that are responsible for certain parts of the solution, subsequently integrating them with the other parts.

Our first inclination toward a solution may not be the best one. We must always consider alternatives and refine the solution as necessary. The earlier we consider alternatives, the easier it is to modify our approach.

Implementing the solution is the act of taking the design and putting it in a usable form. When developing a software solution to a problem, the implementation stage is the process of actually writing the program. Too often programming is thought of as writing code. But in most cases, the final implementation of the solution is one of the last and easiest steps. The act of designing the program should be more interesting and creative than the process of implementing the design in a particular programming language.

Finally, we test our solution to find any errors that exist so that we can fix them and improve the quality of the software. Testing efforts attempt to verify that the program correctly represents the design, which in turn provides a solution to the problem.

Throughout this text we explore programming techniques that allow us to elegantly design and implement solutions to problems. Although we will often delve into these specific techniques in detail, we should not forget that they are just tools to help us solve problems.

Let us consider a simple problem and use it to explain these concepts in some detail. Let us design and write a program to calculate the sum of any three integer numbers and calculate their average.

The task before us is quite simple enough to understand. To know what we are to do we can begin by asking ourselves the 'what' question. What are we (or in effect the program) expected to do? What are the major processes involved in calculating the sum and average of any three numbers? These could be summarized as follows:

a. First we must be able to accept any three numbers from the user.
b. Calculate the average.
c. Display average.

Next, we look at each of these steps and see if we break them and refine them as necessary; from there onwards we plan how we will implement our solution. This is sometimes described as asking the how question. How can we achieve the tasks we have identified and if necessary refine the step.

Considering step 'a' we do not need to break it down. Let us implement this in our program using dialog boxes.

Then step 'b' - calculate the average – how do we calculate average of three integer numbers? This can be broken down into two steps that is:

i. Add up the three numbers to calculate their sum.

$$sum = firstNumber + secondNumber + thirdNumber$$

ii. Divide the sum by three to calculate the average.

$$average = \frac{sum}{3}$$

The resulting value – average – may be an integer value or a floating point value. This being the case we would declare average as double.

Finally, we consider the step c, that is, displaying the result. We will display the numbers added and then the average using dialog boxes also.

The entire processing steps (algorithm) are rewritten as:

a. First we must be able to accept any three numbers from the user.
b. Calculate the average.
    i. Add up the three numbers to calculate their sum.
    ii. Divide the sum by three to calculate the average.
c. Display average.

Now we proceed to write the Java program, correct any errors and test with sample data. The source code for the program is presented below:

```
/*
 * AverageThreeIntegers
 * Calculates the sumand average of any three integer numbers
 */

import javax.swing.JOptionPane;

public class AverageThreeIntegers
{
   public static void main( String args[] )
  {
      int firstNumber;    // first integer number
      int secondNumber;   // second integer number
      int thirdNumber;    // third integer number
      int sum;            // sum of the three numbers

      double average;     // average of the three numbers

      String input;       // input values
      String result;      // output generating string

      // Accept inteher numbers from he user
      input = JOptionPane.showInputDialog( null, "Enter first number: " );
      firstNumber = Integer.parseInt( input );    // wrap input to integer

      input = JOptionPane.showInputDialog( null, "Enter second number: " );
      secondNumber = Integer.parseInt( input );    // wrap input to integer

      input = JOptionPane.showInputDialog( null, "Enter third number: " );
      thirdNumber = Integer.parseInt( input );    // wrap input to integer

      // Calculate sum
      sum = firstNumber + secondNumber + thirdNumber;

      // Calculate average
      average = sum/3.0;

      // Build output string and display output
      result = "Average of " + firstNumber + ", " + secondNumber + " and " +
            thirdNumber + " is = " + average;

      JOptionPane.showMessageDialog( null, result, "Average of 3 Integers",
            JOptionPane.INFORMATION_MESSAGE );
```

```
    }  // end method main

}  // end class AverageThreeIntegers
```

Listing 4.1 AverageThreeIntegers.java

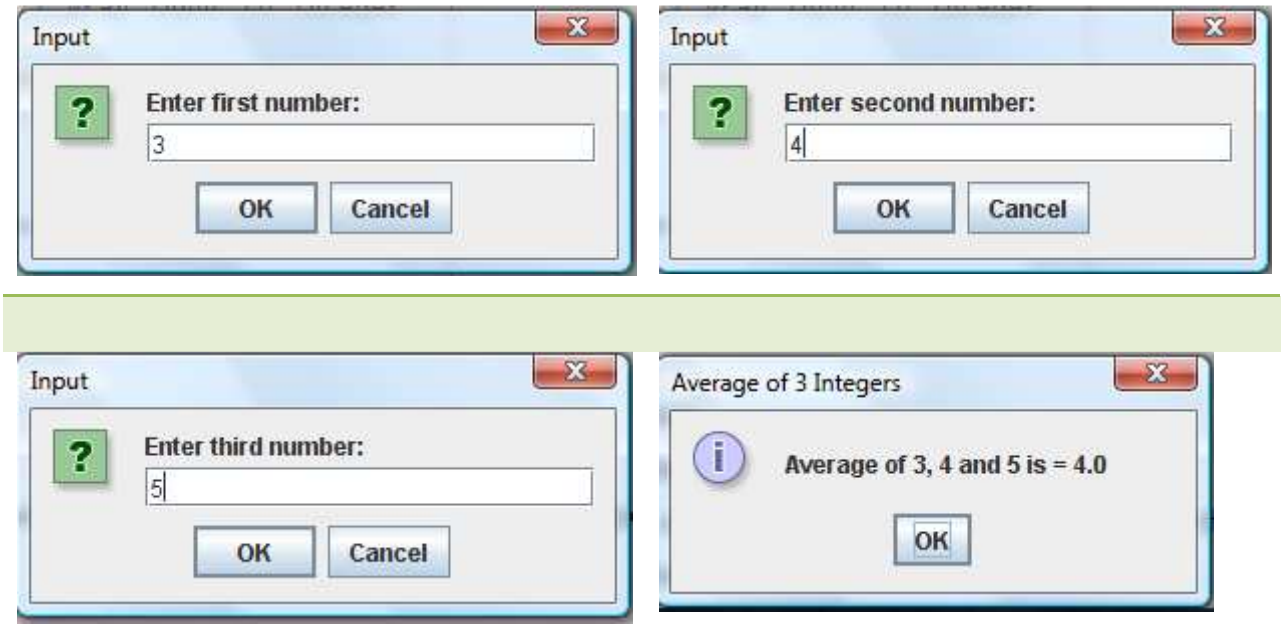


Figure 4.1

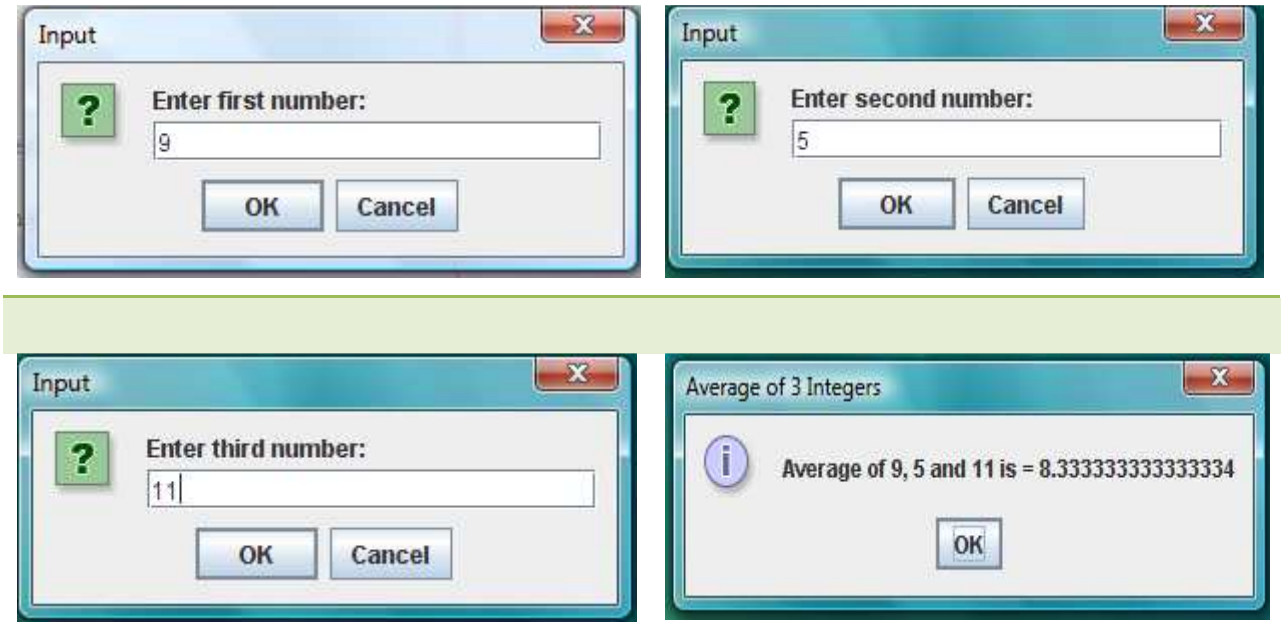


Figure 4.2

The program listing 4.1 needs no elaborate explanation; all the material presented has been explained earlier. Figures 4.1 and 4.2 shows the sample input and output results obtained when the program is executed.

WEEK 5

General Learning Objectives for Week5: Understand Insatiable Classes

Specific Objectives:

a. Define Insatiable Classes.
b. Understand the concepts of Class Members
c. Differentiate between Instance and Local Variables
d. Understand the concepts of declaring methods
e. Describe parameter passing in method definitions
f. Differentiate between public and private data

Classes, Objects, Methods and Instance Variables

Let's begin with a simple analogy to help you understand classes and their contents. Suppose you want to drive a car and make it go faster by pressing down on its accelerator pedal. What must happen before you can do this? Well, before you can drive a car, someone has to design the car. A car typically begins as engineering drawings, similar to the blueprints used to design a house. These engineering drawings include the design for an accelerator pedal to make the car go faster. The pedal "hides" the complex mechanisms that actually make the car go faster, just as the brake pedal "hides" the mechanisms that slow the car and the steering wheel "hides" the mechanisms that turn the car. This enables people with little or no knowledge of how engines work to drive a car easily.

Unfortunately, you cannot drive the engineering drawings of a car. Before you can drive a car, the car must be built from the engineering drawings that describe it. A completed car will have an actual accelerator pedal to make the car go faster, but even that's not enough the car will not accelerate on its own, so the driver must press the accelerator pedal.

Now let's use our car example to introduce the key programming concepts of this section. Performing a task in a program requires a method. The method describes the mechanisms that actually perform its tasks. The method hides from its user the complex tasks that it performs, just as the accelerator pedal of a car hides from the driver the complex mechanisms of making the car go faster. In Java, we begin by creating a program unit called a class to house a method, just as a car's engineering drawings house the design of an accelerator pedal. In a class, you provide one or more methods that are designed to perform the class's tasks. For example, a class that represents a bank account might contain one method to deposit money to an account, another to withdraw money from an account and a third to inquire what the current balance is.

Just as you cannot drive an engineering drawing of a car, you cannot "drive" a class. Just as someone has to build a car from its engineering drawings before you can actually drive a car, you must build an object of a class before you can get a program to perform the tasks the class

describes how to do. That is one reason Java is known as an object-oriented programming language.

When you drive a car, pressing its gas pedal sends a message to the car to perform a taskthat is, make the car go faster. Similarly, you send **messages** to an object each message is known as a **method call** and tells a method of the object to perform its task.

Thus far, we have used the car analogy to introduce classes, objects and methods. In addition to the capabilities a car provides, it also has many attributes, such as its color, the number of doors, the amount of gas in its tank, its current speed and its total miles driven (i.e., its odometer reading). Like the car's capabilities, these attributes are represented as part of a car's design in its engineering diagrams. As you drive a car, these attributes are always associated with the car. Every car maintains its own attributes. For example, each car knows how much gas is in its own gas tank, but not how much is in the tanks of other cars. Similarly, an object has attributes that are carried with the object as it is used in a program. These attributes are specified as part of the object's class. For example, a bank account object has a balance attribute that represents the amount of money in the account. Each bank account object knows the balance in the account it represents, but not the balances of the other accounts in the bank. Attributes are specified by the class's **instance variables**.

The remainder of this chapter presents examples that demonstrate the concepts we introduced in the context of the car analogy.

Insatiable Classes

Insatiable classes can be permits users to create instances – objects of the class. Each object of the class will have its own set of variables – instance variables that represent the current state of the object and methods that defines the task that object can perform. Technically each method should perform only a single task, and it is permitted to call other methods to assist it.

Instance variables are declared outside any method inside the class and usually immediately after the class definition header. See line 8 of figure 5.1. the access modifier is used to ensure that each object maintains it own set of variables and thus not visible to any other object of the class

or any other class. Variables declared inside a method are not visible to any other member of that class or outside that class. Such variables are termed local variables. Instance variables are visible to all members of the class, but not to members of another class. To enable other classes to access instance variables in order to reflect a change in the state of the objects we use public service methods – set and get – methods to facilitate this through a well defined mechanism. The set methods enables changes while get methods gives the current state of the object.

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

We begin with an example that consists of classes `Circle` (Fig. 5.1) and `CircleTest` (Fig. 5.2). Class `Circle` (declared in file `Circle.java`) will be used to define the properties of a Circle object and tasks which each object of the class will be able to perform – in this case calculate the area of a Circle instance. Class `CircleTest` (declared in file `CircleTest.java`) is an application class in which the `main` method will use class `Circle`. Each class declaration that begins with keyword `public` must be stored in a file that has the same name as the class and ends with the `.java` file-name extension. Thus, classes `Circle` and `CircleTest` must be declared in separate files, because each class is declared `public`.

```
 1 /*
 2  * Circle.Java
 3  * Create a Circle Object and Calculate the Area of the Circle
 4  */
 5
 6 public class Circle {
 7     // Create Instance Variables
 8     private double radius;
 9
10     public Circle() {
11         radius = 0;
12     }
13
14     // User defined Constructor for Creating Circle Object with Specified
15     // radius - r
16     public Circle(double r){
17         setRadius( r );
18     }
19
20     // setRadius initializes radius and ensuees that it is always in a
21     // consistent state
22     public void setRadius(double r){
23         radius = r;
24
25     }  // end method setRadius
```

```
26
27      // getRadius returns radius to clients of Circle class
28      public double getRadius(){
29          return radius;
30
31      }  // end method getRadius
32
33      // Calculates the area of the circle
34      public double calcArea(){
35          return Math.PI * Math.pow(radius, 2);

36
37      }  // end method calcArea
38  }  // end class Circle
```

Figure 5.1

Class `Circle`

The `Circle` class declaration (Fig. 3.1) contains a two constructor methods, a no argument constructor `Circle()`method (lines 10-12) that initializes the radius of a Circle object to its default value of zero (0. The second constructor method is a programmer declared constructor `Circle(double r)` method (lines 16-18) that that initializes a Circle object using the values specified by the user.

The methods `setRadius()`and `setRadius()` are public service methods that enables the instance variable **radius** (declared in line 8) to be visible to other classes. The class declaration also contains calcArea`()`that calculates the area of the circle.

So far, each class we declared had one method named `main` (a special method that is always called automatically by the Java Virtual Machine (JVM) when you execute an application). Most methods do not get called automatically. As you will soon see, you must call method `calcArea` to tell it to perform its task.

The method declaration begins with keyword `public` to indicate that the method is "available to the public" that is, it can be called from outside the class declaration's body by methods of other classes. Keyword `double` indicates that this method will perform a task and will return (i.e., give back) a value of type double to its **calling method** when it completes its task. Method `setRadius` on the other hand does not return any value to its calling method thus the keyword `void`.

The name of the method, `calcArea`, follows the return type. By convention, method names begin with a lowercase first letter and all subsequent words in the name begin with a capital letter. The parentheses after the method name indicate that this is a method. An empty set of parentheses, as shown in line 34, indicates that this method does not require additional information to perform its task. Line 7 is commonly referred to as the **method header**. Every method's body is delimited by left and right braces (`{` and `}`), as in lines 34 and 37.

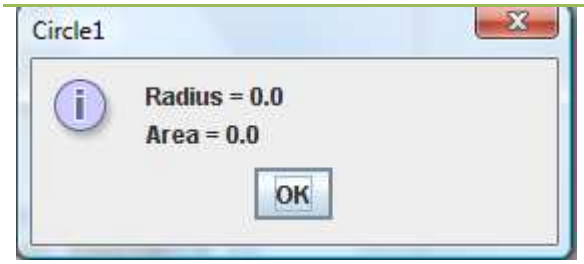
The body of a method contains statement(s) that perform the method's task. In this case, the method contains one statement (line 35) that calculates the area of a cicle. After this statement executes, the method has completed its task.

Next, we'd like to use class `Circle` in an application. As earlier stated method `main` begins the execution of every application. A class that contains method `main` is a Java application. Such a class is special because the JVM can use `main` to begin execution. Class `Circle` is not an application because it does not contain `main`. Therefore, if you try to execute `Circle` by typing `java Circle` in the command window, you will get the error message:

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Class `CircleTest`

The `CircleTest` class declaration (Fig. 5.2) contains the `main` method that will control our application's execution. Any class that contains `main` declared as shown on line 7 can be used to execute an application. This class declaration begins at line 4 and ends at line 16. The class contains only a `main` method, which is typical of many classes that begin an application's execution.



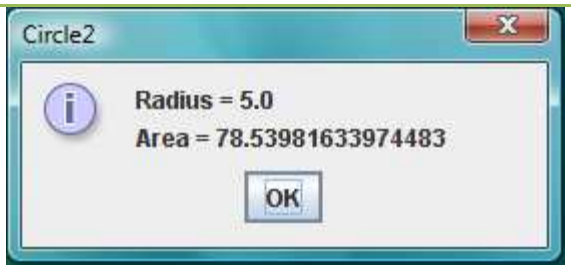


Figure 5.1a Figure 5.1b

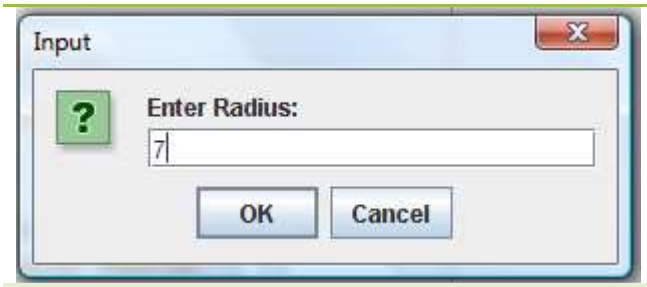


Figure 5.1c

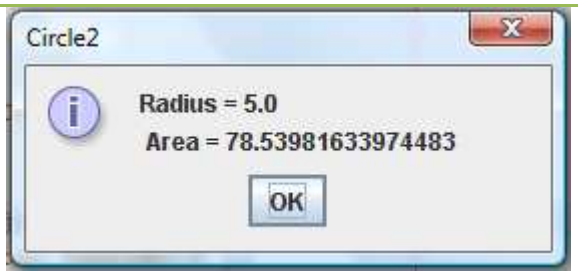


Figure5.1d

```
1  /*
2   * CircleTest.java
3   *
4   */
5
6  package MyTrig;
7
8  import javax.swing.JOptionPane;
9
10 public class CircleTest
11 {
12
13     public static void main(String[] args)
14     {
15         // Declare local variables
16         String input;
17         String output;
18
19         double newRadius = 0;
20
21         // Create Circle object using the no-argument constructor
22         Circle circle1 = new Circle();
23
24         // create another Circle instance class using the Programmer
25         // declared constructor
26         Circle circle2 = new Circle( 5 );   // circle has a radius of 5
27
28         // display state of Circle objects
29         output = "Radius = " + circle1.getRadius() +
30                 "\nArea = " + circle1.calcArea();
31         JOptionPane.showMessageDialog( null, output, "Circle1",
32                 JOptionPane.INFORMATION_MESSAGE );
33
34         output = "Radius = " + circle2.getRadius() +
35                 "\nArea = " + circle2.calcArea();
36         JOptionPane.showMessageDialog( null, output, "Circle2",
37                 JOptionPane.INFORMATION_MESSAGE );
38
39         // Allow user to alter the state of circle1 object
40         input = JOptionPane.showInputDialog( null, "Enter Radius: " );
41         newRadius = Double.parseDouble( input );
42
43         circle1.setRadius( newRadius );     // reset radius variable
44
45         // display current radius and area of circle1
```

```
46          output = "Radius = " + circle2.getRadius() +
47                   "\n Area = " + circle2.calcArea();
48          JOptionPane.showMessageDialog( null, output, "Circle2",
49                   JOptionPane.INFORMATION_MESSAGE );
50
51      }  // end method main
52
53 }  // end class CircleTest
```

Figure 5.2

Lines 13-51 declare method `main`. Recall that the `main` header must appear as shown in line 13; otherwise, the application will not execute. A key part of enabling the JVM to locate and call method `main` to begin the application's execution is the `static` keyword (line 13), which indicates that `main` is a `static` method. A `static` method is special because it can be called without first creating an object of the class in which the method is declared.

In this application, we'd like to call class `Circle`'s `calcArea` method to calculate the area of any Circle object. Typically, you cannot call a method that belongs to another class until you create an object of that class, as shown in lines 30 and 47. We begin by declaring two variables `circle1` and `circle2`. Note that the variable's type is `Circle` the class we declared in Fig. 5.1. Each new class you create becomes a new type in Java that can be used to declare variables and create objects. Programmers can declare new class types as needed; this is one reason why Java is known as an extensible language.

Variables `circle1 and circle2` are initialized with the result of the **class instance creation expression** `new Circle()and circle2(5) respectively`. Keyword **`new`** creates a new object of the class specified to the right of the keyword (i.e., `Circle`). The parentheses to the right of the `Circle` are required. Those parentheses in combination with a class name represent a call to a constructor, which is similar to a method, but is used only at the time an object is created to initialize the object's data.

Just as we can use object `System.out` to call methods `print`, `printf` and `println`, we can now use `Circle objects` to call methods `calcArea, setRadius and getRadius`. Line 30 calls the method `calcArea` (declared at lines 34-37 of Fig. 5.1) using variable `circle1` followed by a **dot separator** (.), the method name `calcArea` and an empty set of parentheses. This call causes the `calcArea` method to perform its task. In line 29, "`circle1`" indicates that `main` should use

the `Circle` object that was created on line 22. Line 34 of Fig. 5.1 indicates that method `calcArea` has an empty parameter list that is, `calcArea` does not require additional information to perform its task. For this reason, the method call (line 29 of Fig. 5.2) specifies an empty set of parentheses after the method name to indicate that no arguments are being passed to method `calcArea`. When method `calcArea` completes its task, method `main` continues executing at line 51. This is the end of method `main`, so the program terminates. See Figures 5.1a-5.1d for sample run output.

Methods and Constructors a Deeper Look

As we earlier mentioned, methods are used for specifying the tasks that objects of the class can perform. Constructors are special methods that are used for initializing objects when they are created. In order to see the differences between the constructors and regular (non-constructor) methods, we have presented below the structure of a method:

```
accessModifer returnType methodName( parameterList )
{
        statements
        return statement
}
```

Where:

accessModifer (access modifier) specifies the visibility of the method to other classes – the access modifier may be specified public or private. Public specifies that the method is visible to other classes "that is visible to the public" while private is the exact opposite. Private methods cannot be accessed outside the class from which it is defined and thus it cannot be inherited.

returnType specifies the "nature" of the value the method gives back to its calling method (if any) when it completes its task. The return type could a primitive type value or reference type value as the case may be.

methodName (method name) is an identifier used to make reference to the method.

parameterList is an optional list of identifiers representing the values (arguments) to be passed into the method. The parameters –often referred to as formal parameters – are used by methods

in carrying out of their tasks. They must have a type followed by an identifier. Several parameters must be separated using commas.

statements – instructions that enable the method to carry out its tasks.

return statement is used to return (send) data back to the calling method. When the return statement is implemented without a value following it, then the returnType must be specified as void.

Both regular methods and constructors are permitted to have parameters (formal parameters). Note: values passed into methods and constructors are referred to as actual parameters which are copies of the actual data except when the data are of type reference.

Differences between Methods and Constructors

a. Constructors must have the same name as the class - both in capitalization and in sequence - in which it is declared.

b. Constructors do not have a return type in its method definition header.

c. The return statement is not required.

d. Constructors are only executed when an object is created, they may not be invoked if the state of the object alters. In such a situation that the state of the object changes, public service methods will be required to assist in reflecting the change.

e. Constructors may not be declared as static as they are involved in the creation and initialization of the objects themselves.

General Learning Objectives for Week 6: Introduction to Applets

Specific Learning Objectives:

i. Know the structure of an Applet program.
ii. Write simple applet programs
iii. Accept input data using applets
iv. Implement the “this” keyword.

6.0 Introduction to Applets

There are two kinds of Java programs: Java applets and Java applications. A Java *applet* is a Java program that is intended to be embedded into an HTML document, transported across a network, and executed using a Web browser.

A Java *application* is a stand-alone program that can be executed using the Java interpreter. All programs presented thus far have been Java applications The Web enables users to send and receive various types of media, such as text, graphics, and sound, using a point-and-click interface that is extremely convenient and easy to use. A Java applet was the first kind of executable program that could be retrieved using Web software. Java applets are considered just another type of media that can be exchanged across the Web.

Though Java applets are generally intended to be transported across a network, they don't have to be. They can be viewed locally using a Web browser. For that matter, they don't even have to be executed through a Web browser at all. A tool in Sun's Java Software Development Kit called appletviewer can be used to interpret and execute an applet. We use appletviewer to display most of the applets in the book. However, usually the point of making a Java applet is to provide a link to it on a Web page and allow it to be retrieved and executed by Web users anywhere in the orld. Java bytecode (not Java source code) is linked to an HTML document and sent across the Web. A version of the Java interpreter embedded in a Web browser is used to execute the applet once it reaches its destination. A Java applet must be compiled into bytecode format before it can be used with the Web.

There are some important differences between the structure of a Java applet and the structure of a Java application. Because the Web browser that executes an applet is already running, applets can be thought of as a part of a larger program. As such they do not have a main method where execution starts. The paint method in an applet is automatically invoked by the applet.

Consider the program in Listing 6.1, though trivial, is used to show the processing stages of execution of applets. The program displays to the status bar of the applet window the name of the method in its currently in and then draw the users attention to it by displaying a dialog box. The three import statements at the beginning of the program explicitly indicate the packages that are used in the program. In this example, we need the JApplet class, which is part of the java.JApplet

package, the JOptionPane from the javax.swing package and various graphics capabilities defined in the java.awt package.

A class that defines an applet extends the JApplet class, as indicated in the header line of the class declaration. This process is making use of the object oriented concept of inheritance, which we explore in more detail later.

JApplet classes must also be declared as public.

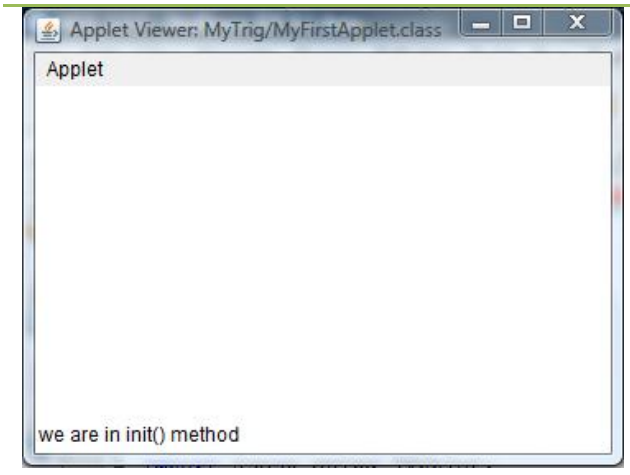
The paint method is one of several applet methods that have particular significance. It is invoked automatically whenever the graphic elements of the applet need to be painted to the screen, such as when the applet is first run or when another window that was covering it is moved.

We will create a JApplet class – MyFirstApplet.java – to demonstrate how a simple applet can be created and the life-cycle of an applet. The complete code listing is presented below.

```
/*
 * MyFirstApplet.java
 *
 */

import java.awt.Graphics;
import javax.swing.JOptionPane;
import javax.swing.JApplet;

public class MyFirstApplet extends JApplet
{
   public void init()
   {
      showStatus( "we are in init() method " );
      JOptionPane.showMessageDialog( null,
            "Check the status bar we are in init() " );
   }  // end method init

   public void start()
   {
      showStatus( "we are in start() method ");
      JOptionPane.showMessageDialog( null,
            "Check the status bar we are in start() " );
   }  // end method start

   public void paint(Graphics g)
```

```
    {
       super.paint( g );
       showStatus( "we are in paint() method ");
       JOptionPane.showMessageDialog( null,
             "Check the status bar we are in paint() " );
    }  // end method paint

    public void stop()
    {
       showStatus( "we are in stop() method ");
       JOptionPane.showMessageDialog( null,
             "Check the status bar we are in stop() " );
    }  // end method stop

    public void destroy()
    {
       showStatus( "we are in destroy() method ");
       JOptionPane.showMessageDialog( null,
             "Check the status bar we are in destroy() " );
    }  // end method destroy
}  // end JApplet class
```

Listing 6.1



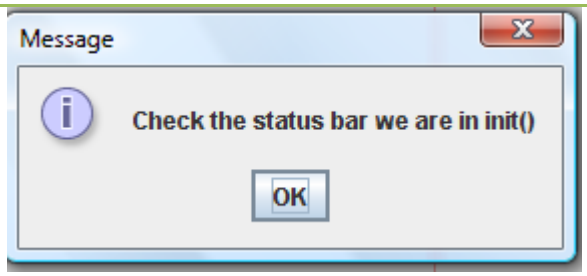
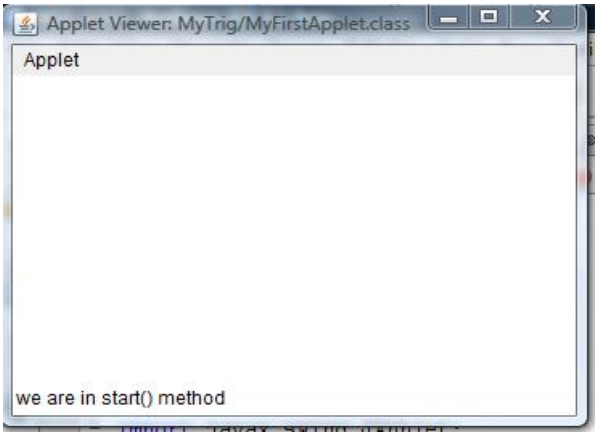


Fig 6.1a Fig 6.1b



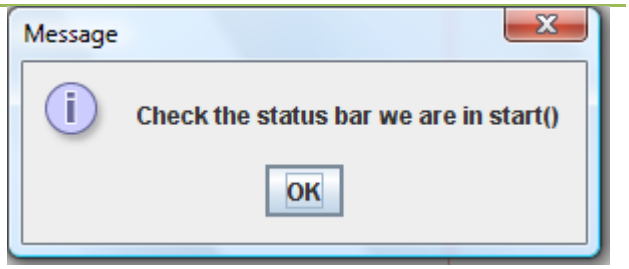
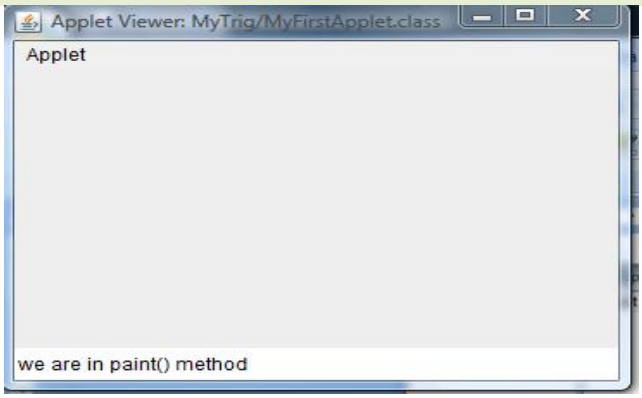


Fig 6.1c Fig 6.1d



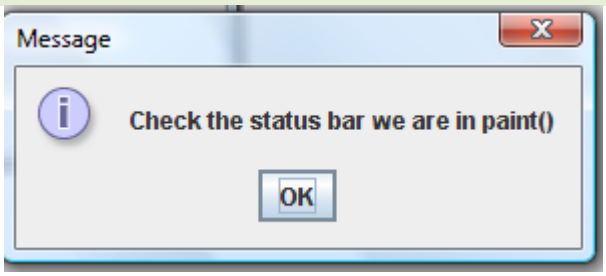
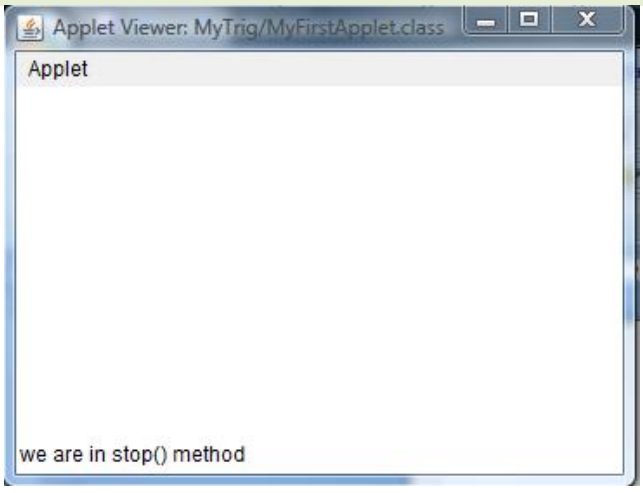


Fig 6.1e Fig 6.1f



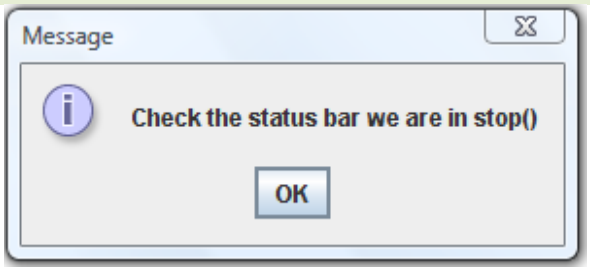
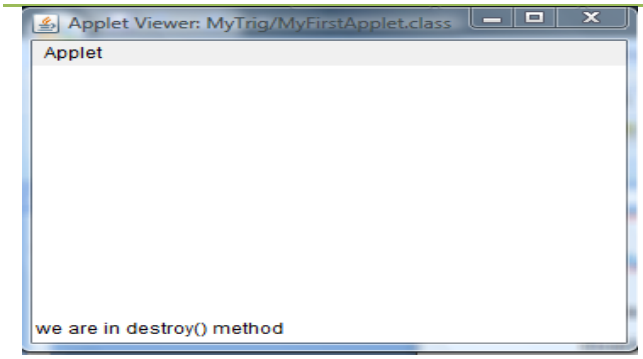


Fig 6.1g Fig 6.1h



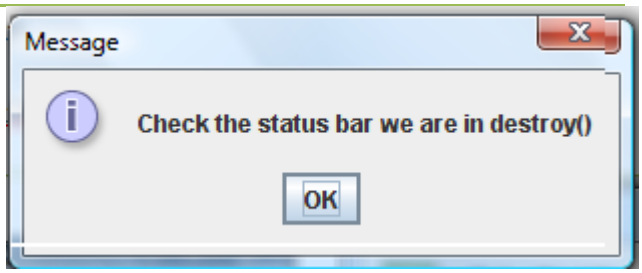


Fig 6.1i Fig 6.1j

Applet Life-Cycle Methods

Now that you have created an applet, let's consider the five applet methods that are called by the applet container from the time the applet is loaded into the browser to the time that the applet is terminated by the browser. These methods correspond to various aspects of an applet's life cycle. The table below lists these methods, which are inherited into your applet classes from class JApplet. The table specifies when each method gets called and explains its purpose. Other than method paint, these methods have empty bodies by default. If you would like to declare any of these methods in your applets and have the applet container call them, you must use the method headers shown in the table. If you modify the method headers (e.g., by changing the method names or by providing additional parameters), the applet container will not call your methods. Instead, it will call the superclass methods inherited from JApplet.

Method	**When the method is called and its purpose**
public void init()	
	Called once by the applet container when an applet is loaded for execution. This method initializes an applet. Typical actions performed here are initializing fields, creating GUI components, loading sounds to play, loading images to display etc
public void start()	
	Called by the applet container after method init completes execution. In addition, if the user browses to another Web site and later returns to the applet's HTML page, method start is called again. The method performs any tasks that must be completed when the applet is loaded for the first time and that must be performed every time the

	applet's HTML page is revisited. Actions performed here might include starting an animation
public void paint(Graphics g)	
	Called by the applet container after methods init and start. Method paint is also called when the applet needs to be repainted. For example, if the user covers the applet with another open window on the screen and later uncovers the applet, the paint method is called. Typical actions performed here involve drawing with the Graphics object g that is passed to the paint method by the applet container.
public void stop()	
	This method is called by the applet container when the user leaves the applet's Web page by browsing to another Web page. Since it is possible that the user might return to the Web page containing the applet, method stop performs tasks that might be required to suspend the applet's execution, so that the applet does not use computer processing time when it is not displayed on the screen. Typical actions performed here would stop the execution of animations and threads.
public void destroy()	
	This method is called by the applet container when the applet is being removed from memory. This occurs when the user exits the browsing session by closing all the browser windows and may also occur at the browser's discretion when the user has browsed to other Web pages. The method performs any tasks that are required to clean up resources allocated to the applet.

Executing Applets Using the Web

In order for the applet to be transmitted over the Web and executed by a browser, it must be referenced in a HyperText Markup Language (HTML) document. An HTML document contains *tags* that specify formatting instructions and identify the special types of media that are to be included in a document. A Java program is considered a specific media type, just as text, graphics, and sound are.

An HTML tag is enclosed in angle brackets. The following is an example of an applet

tag:
<applet code=”MyFirstApplet.class” width=350 height=175>
</applet>

This tag dictates that the bytecode stored in the file MyFirstApplet.class should be transported over the network and executed on the machine that wants to view this particular HTML document. The applet tag also indicates the width and height of the applet.

Note that the applet tag refers to the bytecode file of the MyFirstApplet applet, not to the source code file. Before an applet can be transported using the Web, it must be compiled into its bytecode format.

A JApplet Addition Program

We now present a JApplet program that accept from the user any two numbers (integer or floating point values) and calculate the sum. In this program we use the graphic object to draw the output of the sum inside a rectangle by using the graphics method drawRect() and the drawString() respectively.

The complete source code is presented below in listing 6.2 followed by the output generated when the program is executed.

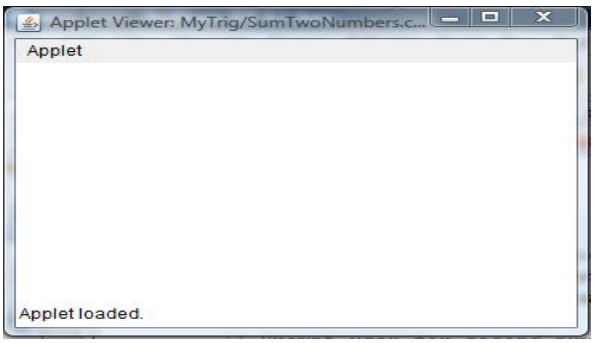


Figure 6.2a

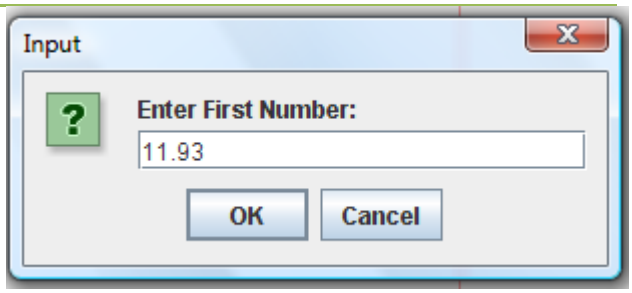


Figure 6.2b

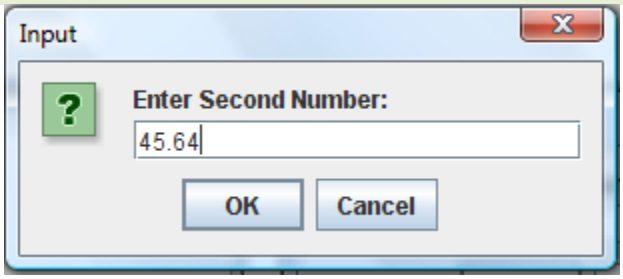


Figure 6.2c

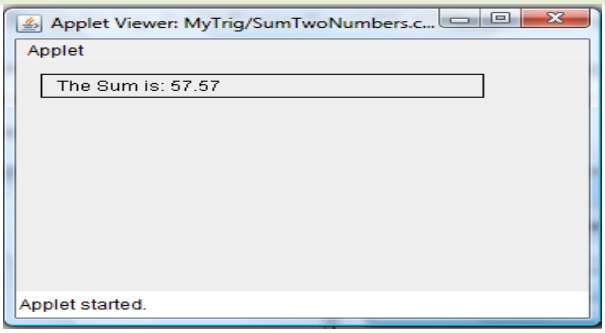


Figure 6.2d

WEEK 7

General Learning Objectives for Week7: Know the Use of Conditional Statements

Specific Objectives:

g. Understand algorithm
h. Understand pseudocodes
i. Know and identify relational and logical operators
j. Know how to write simple relational and logical expressions
k. Know the structure of the if-statement
l. Apply the if-statement
m. Know and apply the switch statement
n. Apply nested if-statements

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
3. is called an **algorithm**. The following example demonstrates that correctly specifying the order in which the actions execute is important.

Consider the "rise-and-shine algorithm" followed by one executive for getting out of bed and going to work: (1) Get out of bed; (2) take off pajamas; (3) take a shower; (4) get dressed; (5) eat breakfast; (6) carpool to work. This routine gets the executive to work well prepared to make critical decisions. Suppose that the same steps are performed in a slightly different order: (1) Get out of bed; (2) take off pajamas; (3) get dressed; (4) take a shower; (5) eat breakfast; (6) carpool to work. In this case, our executive shows up for work soaking wet.

Pseudocode

Pseudocode is an informal language that helps programmers develop algorithms without having to worry about the strict details of Java language syntax. The pseudocode we present is particularly useful for developing algorithms that will be converted to structured portions of Java programs. Pseudocode is similar to everyday Englishit is convenient and user friendly, but it is not an actual computer programming language.

Pseudocode does not execute on computers. Rather, it helps the programmer "think out" a program before attempting to write it in a programming language, such as Java. This chapter provides several examples of how to use pseudocode to develop Java programs.

The style of pseudocode we present consists purely of characters, so programmers can type pseudocode conveniently, using any text-editor program. A carefully prepared pseudocode program can easily be converted to a corresponding Java program. In many cases, this simply requires replacing pseudocode statements with Java equivalents.

Pseudocode normally describes only statements representing the actions that occur after a programmer converts a program from pseudocode to Java and the program is run on a computer. Such actions might include input, output or a calculation. We typically do not include variable declarations in our pseudocode. However, some programmers choose to list variables and mention their purposes at the beginning of their pseudocode

Specifying the order in which statements (actions) execute in a program is called **program control**. Normally, statements in a program are executed one after the other in the order in which they are written. This process is called **sequential execution**. Various Java statements, which we will soon discuss, enable the programmer to specify that the next statement to execute is not necessarily the next one in sequence. This is called **transfer of control**.

During the 1960s, it became clear that the indiscriminate use of transfers of control was the root of much difficulty experienced by software development groups. The blame was pointed at the **`goto statement`** (used in most programming languages of the time), which allows the programmer to specify a transfer of control to one of a very wide range of possible destinations in a program. The notion of so-called **structured programming** became almost synonymous with "`goto` elimination." [Note: Java does not have a `goto` statement; however, the word `goto` is reserved by Java and should not be used as an identifier in programs.]

The research of Bohm and Jacopinimk:@MSITStore:C:\java\Java%20-%20How%20To%20Program,%206th%20Edition%20(2004).chm::/0131483986/ch04lev1sec4.html - ch04fn1 had demonstrated that programs could be written without any `goto` statements. The challenge of the era for programmers was to shift their styles to "`goto`-less programming." Not until the 1970s did programmers start taking structured programming seriously. The results were impressive. Software development groups reported shorter development times, more frequent on-time delivery of systems and more frequent within-budget completion of software projects. The key to these successes was that structured programs were clearer, easier to debug and modify, and more likely to be bug free in the first place.

Bohm and Jacopini's work demonstrated that all programs could be written in terms of only three control structuresthe **sequence structure**, the **selection structure** and the **repetition structure**.

The term "control structures" comes from the field of computer science. When we introduce Java's implementations of control structures, we will refer to them in the terminology of the Java Language Specification as "control statements."

Sequence Structure in Java

The sequence structure is built into Java. Unless directed otherwise, the computer executes Java statements one after the other in the order in which they are written, that is, in sequence. In Fig. 7.1 illustrates a typical sequence structure in which two calculations are performed in order. Java lets us have as many actions as we want in a sequence structure. As we will soon see, anywhere a single action may be placed, we may place several actions in sequence.

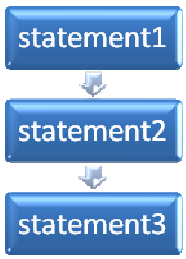


Fig. 7.1 Sequence of instruction executed one after the other.

Selection Statements in Java

Java has three types of selection statements. The `if` statement either performs (selects) an action if a condition is true or skips the action, if the condition is false. The `if...else` statement performs an action if a condition is true and performs a different action if the condition is false. The `switch` statement performs one of many different actions, depending on the value of an expression.

The `if` statement is a **single-selection statement** because it selects or ignores a single action (or, as we will soon see, a single group of actions). The `if...else` statement is called a **double-selection statement** because it selects between two different actions (or groups of actions). The

`switch` statement is called a **multiple-selection statement** because it selects among many different actions (or groups of actions).

`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"
```

determines whether the condition "student's grade is greater than or equal to 60" is true or false. If the condition is true, "Passed" is printed, and the next pseudocode statement in order is "performed." (Remember that pseudocode is not a real programming language.) If the condition is false, the Print statement is ignored, and the next pseudocode statement in order is performed. The indentation of the second line of this selection statement is optional, but recommended, because it emphasizes the inherent structure of structured programs.

The preceding pseudocode If statement may be written in Java as

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

Note that the Java code corresponds closely to the pseudocode. This is one of the properties of pseudocode that makes it such a useful program development tool.

`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 the programmer 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"
```

prints "Passed" if the student's grade is greater than or equal to 60, but prints "Failed" if it is less than 60. In either case, after printing occurs, the next pseudocode statement in sequence is "performed."

The preceding If...Else pseudocode statement can be written in Java as

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

Note that the body of the `else` is also indented. Whatever indentation convention you choose should be applied consistently throughout your programs. It is difficult to read programs that do not obey uniform spacing conventions.

Conditional Operator (`?:`)

Java provides the conditional operator (`?:`) that can be used in place of an `if`...`else` statement. This is Java's only **ternary operator** this means that it takes three operands. Together, the operands and the `?:` symbol form a **conditional expression.** The first operand (to the left of the `?`) is a **`boolean`** expression (i.e., a condition that evaluates to a `boolean` value`true` or **`false`**), the second operand (between the `?` and `:`) is the value of the conditional expression if the `boolean` expression is `TRue` and the third operand (to the right of the `:`) is the value of the conditional expression if the `boolean` expression evaluates to `false`. For example, the statement

```
System.out.println( studentGrade >= 60 ? "Passed" : "Failed" );
```

prints the value of `println`'s conditional-expression argument. The conditional expression in this statement evaluates to the string `"Passed"` if the `boolean` expression `studentGrade >= 60`

is true and evaluates to the string `"Failed"` if the `boolean` expression is false. Thus, this statement with the conditional operator performs essentially the same function as the `if...else` statement shown earlier in this section. The precedence of the conditional operator is low, so the entire conditional expression is normally placed in parentheses. We will see that conditional expressions can be used in some situations where `if...else` statements cannot.

Nested `if...else` Statements

A program can test multiple cases by placing `if...else` statements inside other `if...else` statements to create **`nested if...else statements`**. For example, the following pseudocode represents a nested `if...else` that prints `A` for exam grades greater than or equal to 90, `B` for grades in the range 80 to 89, `C` for grades in the range 70 to 79, `D` for grades in the range 60 to 69 and `F` for all other grades:

```
If student's grade is greater than or equal to 90
    Print "A"
else if student's grade is greater than or equal to 80
        Print "B"
    else
        If student's grade is greater than or equal to 70
            Print "C"
        else
            If student's grade is greater than or equal to 60
                Print "D"
            else
                Print "F"
```

This pseudocode may be written in Java as

```
if ( studentGrade >= 90 )
   System.out.println( "A" );
else
   if ( studentGrade >= 80 )
      System.out.println( "B" );
   else
      if ( studentGrade >= 70 )
         System.out.println( "C" );
      else
         if ( studentGrade >= 60 )
            System.out.println( "D" );
         else
            System.out.println( "F" );
```

If `studentGrade` is greater than or equal to 90, the first four conditions will be true, but only the statement in the `if`-part of the first `if`...`else` statement will execute. After that statement executes, the `else`-part of the "outermost" `if`...`else` statement is skipped. Most Java programmers prefer to write the preceding `if`...`else` statement as

```
if ( studentGrade >= 90 )
   System.out.println( "A" );
else if ( studentGrade >= 80 )
   System.out.println( "B" );
else if ( studentGrade >= 70 )
   System.out.println( "C" );
else if ( studentGrade >= 60 )
   System.out.println( "D" );
else
   System.out.println( "F" );
```

The two forms are identical except for the spacing and indentation, which the compiler ignores. The latter form is popular because it avoids deep indentation of the code to the right. Such indentation often leaves little room on a line of code, forcing lines to be split and decreasing program readability.

Dangling-`else` Problem

The Java compiler always associates an `else` with the immediately preceding `if` unless told to do otherwise by the placement of braces (`{` and `}`). This behavior can lead to what is referred to as the **`dangling-else problem`**. For example,

```
if ( x > 5 )
   if ( y > 5 )
      System.out.println( "x and y are > 5" );
else
   System.out.println( "x is <= 5" );
```

appears to indicate that if `x` is greater than `5`, the nested `if` statement determines whether `y` is also greater than `5`. If so, the string `"x and y are > 5"` is output. Otherwise, it appears that if `x` is not greater than `5`, the `else` part of the `if`...`else` outputs the string `"x is <= 5"`.

Beware! This nested `if...else` statement does not execute as it appears. The compiler actually interprets the statement as

```
if ( x > 5 )
   if ( y > 5 )
      System.out.println( "x and y are > 5" );
   else
      System.out.println( "x is <= 5" );
```

in which the body of the first `if` is a nested `if...else`. The outer `if` statement tests whether `x` is greater than `5`. If so, execution continues by testing whether `y` is also greater than `5`. If the second condition is true, the proper string`"x and y are > 5"`is displayed. However, if the second condition is false, the string `"x is <= 5"` is displayed, even though we know that `x` is greater than `5`.

To force the nested `if...else` statement to execute as it was originally intended, we must write it as follows:

```
if ( x > 5 )
{
   if ( y > 5 )
      System.out.println( "x and y are > 5" );
}
else
   System.out.println( "x is <= 5" );
```

The braces (`{}`) indicate to the compiler that the second `if` statement is in the body of the first `if` and that the `else` is associated with the first `if`.

Blocks

The `if` statement normally expects only one statement in its body. To include several statements in the body of an `if` (or the body of an `else` for an `if...else` statement), enclose the statements in braces (`{` and `}`). A set of statements contained within a pair of braces is called a **block**. A block can be placed anywhere in a program that a single statement can be placed.

The following example includes a block in the `else`-part of an `if...else` statement:

```
if ( grade >= 60 )
   System.out.println( "Passed" );
else
{
   System.out.println( "Failed" );
   System.out.println( "You must take this course again." );
}
```

In this case, if `grade` is less than 60, the program executes both statements in the body of the `else` and prints

```
Failed.
You must take this course again.
```

Note the braces surrounding the two statements in the `else` clause. These braces are important. Without the braces, the statement

```
System.out.println( "You must take this course again." );
```

would be outside the body of the `else`-part of the `if`...`else` statement and would execute regardless of whether the grade was less than 60.

Syntax errors (e.g., when one brace in a block is left out of the program) are caught by the compiler. A **logic error** (e.g., when both braces in a block are left out of the program) has its effect at execution time. A **fatal logic error** causes a program to fail and terminate prematurely. A **nonfatal logic error** allows a program to continue executing, but causes the program to produce incorrect results.

WEEK 8

General Learning Objectives for Week8: Know the Use of Selection Statements

Specific Objectives:

a. Apply the while statement
b. Apply the do statement
c. Write simple programs to implement the while and do statements
d. Develop algorithms for solving simple repetitive problems – counter controlled and sentinel-controlled algorithms.
e. Applies a JTextArea and a JScrollPane class to display the numbers.

The `while` Repetition Statement

A **repetition statement** (also called a looping statement or a **loop**) allows the programmer 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
```

describes the repetition that occurs during a shopping trip. The condition "there are more items on my shopping list" may be true or false. If it is true, then the action "Purchase next item and cross it off my list" is performed. This action will be performed repeatedly while the condition remains true. The statement(s) contained in the While repetition statement constitute the body of the While repetition statement, which may be a single statement or a block. Eventually, the condition will become false (when the last item on the shopping list has been purchased and crossed off the list). At this point, the repetition terminates, and the first statement after the repetition statement executes.

As an example of Java's `while` repetition statement, consider a program segment designed to find the first power of 3 larger than 100. Suppose that the `int` variable `product` is initialized to `3`. When the following `while` statement finishes executing, `product` contains the result:

```
int product = 3;

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

When this `while` statement begins execution, the value of variable `product` is 3. Each iteration of the `while` statement multiplies `product` by 3, so `product` takes on the values 9, 27, 81 and 243 successively. When variable `product` becomes 243, the `while` statement condition`product <= 100`becomes false. This terminates the repetition, so the final value of `product` is 243. At this point, program execution continues with the next statement after the `while` statement.

Formulating Algorithms: Counter-Controlled Repetition

To illustrate how algorithms are developed, we will create a class `TenNos` (declared in TenNos.java) to generate and sum the first ten integer numbers from 1 to 10 by default. Below is the algorithm (pseudocode for generating and calculating the sum of ten numbers from 1 to 10.

Pseudocode: Generate an calculate the sum of the first ten numbers from 1 to 10

First Pseudocode:

Step1: Initialize variables

Step2: Generate numbers from 1 to 10

Step3: Calculate sum of the numbers

Step4: Display numbers and sum

Step5: Stop.

The algorithm may be refined further, for example step1 may be broken down such that the variables to be initialize will be specified thus; step1 becomes:

Step1: counter = 1, sum = 0, n = 10

counter is set to 1 because we will start the counting from 1, sum will start from zero so the summation of the numbers will be accurate. The variable n is set to 10 because the lst number in the sequence is ten.

Step2: Generate numbers from 1 to 10

Step3: Calculate sum of the numbers

Steps 2 and 3 will be simplified and expanded further. As each number is generate the sum will be calculated and updated. The number will continuously be generated as long as the number generated does not exceed ten. When this happens we will terminate the loop. Hence steps 2 and 3 becomes:

Step2: while counter is less than or equal to n
Step3: add counter to sum
Step4: increment counter by 1
Step5 return to Step 2

The original step 4 and 5 from the first pseudocode becomes Step 6 and 7 repectively. The final refinement is presented below:

Final Refined Pseudocode:

Step1: counter = 1, sum = 0, n = 10
Step2: while counter is less than or equal to n
Step3: add counter to sum
Step4: increment counter by 1
Step5 return to Step 2
Step6: Display numbers and sum
Step7:Stop

Based on this algorithm the Java program in declared as TenNos.java was developed. The code listing is presented below:

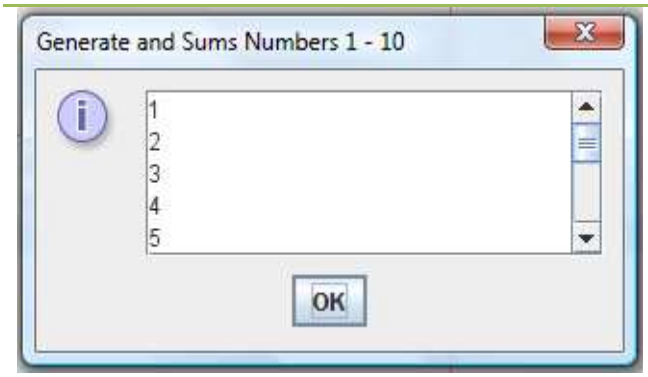
```
 1 /*
 2  * TenNos.Java
 3  * Generates the First Ten Numbers and Calculate their Sum
 4  *
 5  */
 6
 7
 8 import javax.swing.JOptionPane;
 9 import javax.swing.JTextArea;
10 import javax.swing.JScrollPane;
11
12 public class TenNos
13 {
14
15     public static void main(String[] args)
16     {
17         int sum = 0;
18         int counter = 1;
19         int n = 10;
20         String nos = "";
21
22         // Create JTextArea for displaying numbers
23         JTextArea output = new JTextArea( 5, 20 );
24
25         // Generate numbersn
26         while(counter <= n ){
27             output.append(counter + "\n");  // add numbers to the JTextarea
28             sum += counter;                 // calculate sum
29             counter++;                      // increment counter by 1
30         }  // end while i <= n
31
32         nos = "\nSum = " + sum;
33         output.append(nos);
34
35         // Append a JScrollpane to the JTextArea object
36         JScrollPane outputArea = new JScrollPane( output );
37
38         // Display numbers and their sum
39         JOptionPane.showMessageDialog(null, outputArea,
```

```
40                  "Generate and Sums Numbers 1 - 10",
41                  JOptionPane.INFORMATION_MESSAGE);
42
43      }  // end method main
44
45 }  // end of class TenNos
```

Figure 8.1 TenNos.java

Lines 8 – 10 contains the import declaration of the classes we need to build our class TenNos.java. The local variables representing the counter, the nth value and the sum are initialized in lines 17 – 20. An object reference of class JTextArea (package javax.swing) was created in line 23. The while loop was declared from lines 26 – 30. The while statement will continue to iterate as long as the variable `counter` is less than or equal to the variable `n`. With each iteration of the while statement, the value of counter is added to the text area object (output) –line 2, sums the number in stores the result in the variable sum and increments the counter by one and then control is returned to the while statement for re-evaluation if the value of counter is greater the ten, the loop terminates and control is transferred to the first executable statement after the closing curly brace enclosing the body of the while loop (at line 32)..

The sum of the numbers is then added to the string variable `nos` (line32) and appended to the JTextArea object `output` (lines 33). The entire numbers generated and their sum is displayed using the static object `showMessageDialog` of the JOptionPane class (lines 39-41). See figures 8.1a and 8.1b for a sample output run.



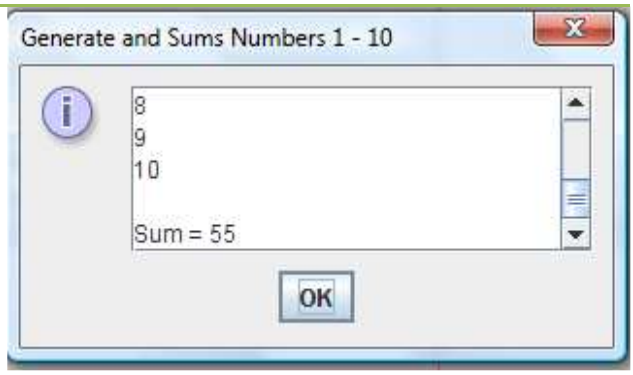


Figure 8.1a Figure 8.1b

General Learning Objectives for Week9: Recursion

Specific Objectives:

a. Understand the concepts of recursion
b. Write simple recursive methods

Recursive Concepts

The programs we have discussed thus far are generally structured as methods that call one another in a disciplined, hierarchical manner. For some problems, however, it is useful to have a method call itself. Such a method is known as a **recursive method**. A recursive method can be called either directly or indirectly through another method.

Recursive problem-solving approaches have a number of elements in common. When a recursive method is called to solve a problem, the method actually is capable of solving only the simplest case(s), or **base case(s)**. If the method is called with a base case, the method returns a result. If the method is called with a more complex problem, the method typically divides the problem into two conceptual piecesa piece that the method knows how to do and a piece that the method does not know how to do. To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or smaller version of it. Because this new problem looks like the original problem, so the method calls a fresh copy of itself to work on the smaller problem this is referred to as a **recursive call** and is also called the **recursion step**. The recursion step normally includes a `return` statement, because its result will be combined with the portion of the problem the method knew how to solve to form a result that will be passed back to the original caller.

The recursion step executes while the original call to the method is still active (i.e., while it has not finished executing). The recursion step can result in many more recursive calls as the method divides each new sub-problem into two conceptual pieces. For the recursion to eventually terminate, each time the method calls itself with a simpler version of the original problem, the sequence of smaller and smaller problems must converge on a base case. At that point, the method recognizes the base case and returns a result to the previous copy of the method. A sequence of returns ensues until the original method call returns the final result to the caller.

A recursive method may call another method, which may in turn make a call back to the recursive method. Such a process is known as an indirect recursive call or **indirect recursion**. For example, method `A` calls method `B`, which makes a call back to method `A`. This is still

considered recursion, because the second call to method A is made while the first call to method A is active that is, the first call to method A has not yet finished executing (because it is waiting on method B to return a result to i) and has not returned to method A's original caller.

Example Using Recursion: Factorials

Let us write a recursive program to perform a popular mathematical calculation. Consider the factorial of a positive integer n, written n! (and pronounced "n factorial"), which is the product

$$n.(n-1).(n-2). \ \ldots \ \ .1$$

with 1! equal to 1 and 0! defined to be 1. For example, 5! is the product 5 · 4 · 3 · 2 · 1, which is equal to 120.

The factorial of integer `number` (where `number` ≥ 0) can be calculated iteratively (non-recursively) using a `for` statement as follows:

```
factorial = 1;
for ( int counter = number; counter >= 1; counter-- )
   factorial *= counter;
```

A recursive declaration of the factorial method is arrived at by observing the following relationship:

$$n! = n.(n-1)!$$

For example, 5! is clearly equal to 5 · 4!, as is shown by the following equations:

5!=5.4.3.2.1
5!=5.(4.3.2.1)
5!=5.4!

The evaluation of 5! would proceed as shown in Fig. 9.1a and Fig 9.1b shows how the succession of recursive calls proceeds until 1! (the base case) is evaluated to be 1, which terminates the recursion. Fig 9.1c shows the values returned from each recursive call to its caller until the final value is calculated and returned.

Figure 9.1a Sequence of Recursive calls

Figure9.1b values returned from each recursive calls

Final value 120

5!=5*24=120 is returned

4!=4*6=24 is returned

3!=3*2=6 is returned

2!=2*1=2 is returned

1 is returned

Figure 9.1c

Figure 9.2 uses recursion to calculate and print the factorials of the integers from 010. The recursive method `factorial` (lines 8-14) first tests to determine whether a terminating condition (line 10) is `True`. If `number` is less than or equal to `1` (the base case), `factorial` returns `1`, no further recursion is necessary and the method returns. If `number` is greater than `1`, line 13 expresses the problem as the product of `number` and a recursive call to `factorial` evaluating the factorial of `number - 1`, which is a slightly simpler problem than the original calculation, `factorial( number )`.

```
 1 /*
 2  * FactorialCalculator.java
 3  *
 4  */
 5
 6 public class FactorialCalculator {
 7     // recursive method factorial
 8     public long factorial( long number ) {
 9         if( number <= 1 )
10             return 1;
11         else  // recursive step
12             return number * factorial( number - 1 );
13
14     }  // end method factorial
15
16     // output factorials from values from 0 thruogh 10
17     public void displayFactorials() {
18         // calculate the factorials from 0 through 10
19         for( int counter=0; counter <= 10; counter++ )
20             System.out.printf( "%d! = %d\n", counter, factorial(counter));
21     }  // end method displayFactorials
22 } // end class FactorialCalculator
```

Figure 9.2

```
 1 /*
 2  * FactorialTest.java
 3  *
 4  */
 5
 6 package hello;
 7
 8 public class FactorialTest {
 9
10     public static void main(String[] args) {
11         FactorialCalculator factorialCalculator = new
FactorialCalculator();
12         factorialCalculator.displayFactorials();
13     }  // end method main
14
15 } // end class FactorialTest
```

Figure 9.3

Method `displayFactorials` (lines 17-22) displays the factorials of 0-10. The call to method `factorial` occurs in line 21. Method `factorial` receives a parameter of type `long` and returns a result of type `long`. Figure 9.2 tests our `factorial` and `displayFactorials` methods by calling `displayFactorials` (line 10). As can be seen from the output of Fig. 9.2, factorial values become large quickly. We use type `long` (which can represent relatively large integers) so the program can calculate factorials greater than 12!. Unfortunately, the `factorial` method produces large values so quickly that factorial values soon exceed the maximum value that can be stored even in a `long` variable.

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Figure 9.4 sample output

General Learning Objectives for Week10: Characters and Strings

Specific Objectives:

a. Describe and manipulate character type data
b. Differentiate between string and string buffer classes
c. Differentiate between equivalence and equality for string objects
d. Show how objects are passed to and returned from methods

Fundamentals of Characters and Strings

Characters are the fundamental building blocks of Java source programs. Every program is composed of a sequence of characters that when grouped together meaningfully are interpreted by the computer as a series of instructions used to accomplish a task. A program may contain **character literals**. A character literal is an integer value represented as a character in single quotes. For example, `'z'` represents the integer value of `z`, and `'\n'` represents the integer value of newline. The value of a character literal is the integer value of the character in the **Unicode character set**.

What are Strings?

A string is a sequence of characters treated as a single unit. A string may include letters, digits and various **special characters**, such as `+`, `-`, `*`, `/` and `$`. A string is an object of class `String`. **String literals** (stored in memory as `String` objects) are written as a sequence of characters in double quotation marks, as in:

`"John Q. Doe"`	(a name)
`"9999 Main Street"`	(a street address)
`"Waltham, Massachusetts"`	(a city and state)
`"(201) 555-1212"`	(a telephone number)

A string may be assigned to a `String` reference. The declaration

```
String color = "blue";
```

initializes `String` reference color to refer to a String object that contains the string `"blue"`.

Class String

Class `String` is used to represent strings in Java. The next several subsections cover many of class `String`'s capabilities.

`String` Constructors

Class `String` provides constructors for initializing `String` objects in a variety of ways. Four of the constructors are demonstrated in the `main` method of Fig. 10.1.

```
 1  // Fig. 10.1 StringConstructors.java
 2  // String class constructors.
 3
 4  public class StringConstructors
 5  {
 6     public static void main( String args[] )
 7     {
 8       char charArray[] = { 'b', 'i', 'r', 't', 'h', ' ', 'd', 'a' , 'y' };
 9        String s = new String( "hello" );
10
11        // use String constructors
12        String s1 = new String();
13        String s2 = new String( s );
14        String s3 = new String( charArray );
15        String s4 = new String( charArray, 6, 3);
16
17        System.out.printf(
18           "s1 = %s\ns2 = %s\ns3 = %s\ns4 = %s\n",
19           s1, s2, s3, s4 ); // display strings
20     } // end main
21  } // end class StringConstructors
```

```
s1 =
s2 = hello
s3 = birth day
s4 = day
```

Line 12 instantiates a new `String` object using class `String`'s no-argument constructor and assigns its reference to `s1`. The new `String` object contains no characters (the **empty string**) and has a length of 0.

Line 13 instantiates a new `String` object using class `String`'s constructor that takes a `String` object as an argument and assigns its reference to `s2`. The new `String` object contains the same sequence of characters as the `String` object `s` that is passed as an argument to the constructor.

Line 14 instantiates a new `String` object and assigns its reference to `s3` using class `String`'s constructor that takes a `char` array as an argument. The new `String` object contains a copy of the characters in the array.

Line 15 instantiates a new `String` object and assigns its reference to `s4` using class `String`'s constructor that takes a `char` array and two integers as arguments. The second argument specifies the starting position (the offset) from which characters in the array are accessed. Remember that the first character is at position `0`. The third argument specifies the number of characters (the count) to access in the array. The new `String` object contains a string formed from the accessed characters. If the offset or the count specified as an argument results in accessing an element outside the bounds of the character array, a `StringIndexOutOfBoundsException` is thrown.

`String` Methods `length`, `charAt` and `getChars`

`String` methods **length**, `charAt` and **getChars** return the length of a string, obtain the character at a specific location in a string and retrieve a set of characters from a string as a char array, respectively. The application in Fig. 10.2 demonstrates each of these methods.

```
 1  // Fig. 10.2: StringMiscellaneous.java
 2  // This application demonstrates the length, charAt and getChars
 3  // methods of the String class.
 4
 5  public class StringMiscellaneous
 6  {
 7     public static void main( String args[] )
 8     {
 9        String s1 = "hello there";
10        char charArray[] = new char[ 5 ];
11
12        System.out.printf( "s1: %s", s1 );
13
14        // test length method
15        System.out.printf( "\nLength of s1: %d", s1.length() );
16
17        // loop through characters in s1 with charAt and display reversed
18        System.out.print( "\nThe string reversed is: " );
19
20        for ( int count = s1.length() - 1; count >= 0; count-- )
21           System.out.printf( "%s ", s1.charAt( count ) );
22
23        // copy characters from string into charArray
24        s1.getChars( 0, 5, charArray, 0 );
25        System.out.print( "\nThe character array is: " );
26
27        for ( char character : charArray )
28           System.out.print( character );
29
30        System.out.println();
31     } // end main
```

```
32   } // end class StringMiscellaneous
```

```
s1: hello there
Length of s1: 11
The string reversed is: e r e h t   o l l e h

The character array is: hello
```

Line 15 uses `String` method `length` to determine the number of characters in string `s1`. Like arrays, strings always know their own length. However, unlike arrays, you cannot access a `String`'s length via a `length` fieldinstead you must call the `String`'s `length` method.

The `for` statement at lines 20-21 print the characters of the string `s1` in reverse order (and separated by spaces). `String` method `charAt` (line 21) returns the character at a specific position in the string. Method `charAt` receives an integer argument that is used as the index and returns the character at that position. Like arrays, the first element of a string is at position 0.

Line 24 uses `String` method `getChars` to copy the characters of a string into a character array. The first argument is the starting index in the string from which characters are to be copied. The second argument is the index that is one past the last character to be copied from the string. The third argument is the character array into which the characters are to be copied. The last argument is the starting index where the copied characters are placed in the target character array. Next, line 28 prints the `char` array contents one character at a time.

Comparing Strings

Class `String` provides several methods for comparing strings these are demonstrated in the next two examples.

To understand what it means for one string to be greater than or less than another string, consider the process of alphabetizing a series of last names. You would, no doubt, place "Jones" before "Smith" because the first letter of "Jones" comes before the first letter of "Smith" in the alphabet. But the alphabet is more than just a list of 26 letters it is an ordered set of characters. Each letter occurs in a specific position within the set. Z is more than just a letter of the alphabet it is specifically the twenty-sixth letter of the alphabet.

How does the computer know that one letter comes before another? All characters are represented in the computer as numeric codes.When the computer compares two strings, it actually compares the numeric codes of the characters in the strings.

Figure 10.3 demonstrates `String` methods `equals, equalsIgnoreCase, compareTo` and `regionMatches` and using the equality operator == to compare `String` objects.

Figure 10.3. `String` comparisons.

```
// Fig. 10.3: StringCompare.java
// String methods equals, equalsIgnoreCase, compareTo and regionMatches.

public class StringCompare
{
   public static void main( String args[] )
   {
      String s1 = new String( "hello" ); // s1 is a copy of "hello"
      String s2 = "goodbye";
      String s3 = "Happy Birthday";
      String s4 = "happy birthday";

      System.out.printf(
         "s1 = %s\ns2 = %s\ns3 = %s\ns4 = %s\n\n", s1, s2, s3, s4 );

      // test for equality
      if ( s1 equals( "hello" ) ) // true
         System.out.println( "s1 equals \"hello\"" );
      else
         System.out.println( "s1 does not equal \"hello\"" );

      // test for equality with ==
      if ( s1 == "hello" ) // false; they are not the same object
         System.out.println( "s1 is the same object as \"hello\"" );
      else
         System.out.println( "s1 is not the same object as \"hello\"" );

      // test for equality (ignore case)
      if ( s3.equalsIgnoreCase( s4 ) ) // true
         System.out.printf( "%s equals %s with case ignored\n", s3, s4 );
      else
         System.out.println( "s3 does not equal s4" );

      // test compareTo
      System.out.printf(
         "\ns1.compareTo( s2 ) is %d", s1.compareTo( s2 ) );
      System.out.printf(
         "\ns2.compareTo( s1 ) is %d", s2.compareTo( s1 ) );
      System.out.printf(
         "\ns1.compareTo( s1 ) is %d", s1.compareTo( s1 ) );
      System.out.printf(
         "\ns3.compareTo( s4 ) is %d", s3.compareTo( s4 ) );
      System.out.printf(
         "\ns4.compareTo( s3 ) is %d\n\n", s4.compareTo( s3 ) );
```

```
45
46          // test regionMatches (case sensitive)
47          if ( s3.regionMatches( 0, s4, 0, 5 ) )
48             System.out.println( "First 5 characters of s3 and s4 match" );
49          else
50             System.out.println(
51                "First 5 characters of s3 and s4 do not match" );
52
53          // test regionMatches (ignore case)
54          if ( s3. regionMatches( true, 0, s4, 0, 5 ) )
55             System.out.println( "First 5 characters of s3 and s4 match" );
56          else
57             System.out.println(
58                "First 5 characters of s3 and s4 do not match" );
59       } // end main
60  } // end class StringCompare
```

```
s1 = hello
s2 = goodbye
s3 = Happy Birthday
s4 = happy birthday

s1 equals "hello"
s1 is not the same object as "hello"
Happy Birthday equals happy birthday with case ignored

s1.compareTo( s2 ) is 1
s2.compareTo( s1 ) is -1
s1.compareTo( s1 ) is 0
s3.compareTo( s4 ) is -32
s4.compareTo( s3 ) is 32

First 5 characters of s3 and s4 do not match
First 5 characters of s3 and s4 match
```

The condition at line 17 uses method `equals` to compare string `s1` and the string literal `"hello"` for equality. Method `equals` (a method of class `Object` overridden in `String`) tests any two objects for equality the strings contained in the two objects are identical. The method returns `true` if the contents of the objects are equal, and `false` otherwise. The preceding condition is `True` because string `s1` was initialized with the string literal `"hello"`. Method `equals` uses a **lexicographical comparison**it compares the integer Unicode values that represent each character in each string. Thus, if the string `"hello"` is compared with the string `"HELLO"`, the result is `false`, because the integer representation of a lowercase letter is different from that of the corresponding uppercase letter.

The condition at line 23 uses the equality operator == to compare string `s1` for equality with the string literal `"hello"`. Operator == has different functionality when it is used to compare references than when it is used to compare values of primitive types. When primitive-type values are compared with ==, the result is `true` if both values are identical. When references are compared with ==, the result is `true` if both references refer to the same object in memory. To compare the actual contents (or state information) of objects for equality, a method must be invoked. In the case of `Strings`, that method is `equals`. The preceding condition evaluates to `false` at line 23 because the reference `s1` was initialized with the statement

```
s1 = new String( "hello" );
```

which creates a new `String` object with a copy of string literal `"hello"` and assigns the new object to variable `s1`. If `s1` had been initialized with the statement

```
s1 = "hello";
```

which directly assigns the string literal `"hello"` to variable `s1`, the condition would be `TRue`. Remember that Java treats all string literal objects with the same contents as one `String` object to which there can be many references. Thus, lines 8, 17 and 23 all refer to the same `String` object `"hello"` in memory.

If you are sorting `Strings`, you may compare them for equality with method `equals-IgnoreCase`, which ignores whether the letters in each string are uppercase or lowercase when performing the comparison. Thus, the string `"hello"` and the string `"HELLO"` compare as equal. Line 29 uses `String` method `equalsIgnoreCase` to compare string s3`Happy Birthday`for equality with string `s4happy birthday`. The result of this comparison is `true` because the comparison ignores case sensitivity.

Lines 35-44 use method `compareTo` to compare strings. Method `compareTo` is declared in the `Comparable` interface and implemented in the `String` class. Line 36 compares string `s1` to string `s2`. Method `compareTo` returns 0 if the strings are equal, a negative number if the string that invokes `compareTo` is less than the string that is passed as an argument and a positive number if the string that invokes `compareTo` is greater than the string that is passed as an argument.

Method `compareTo` uses a lexicographical comparisonit compares the numeric values of corresponding characters in each string.

The condition at line 47 uses `String` method `regionMatches` to compare portions of two strings for equality. The first argument is the starting index in the string that invokes the method. The second argument is a comparison string. The third argument is the starting index in the comparison string. The last argument is the number of characters to compare between the two strings. The method returns `TRue` only if the specified number of characters are lexicographically equal.

Finally, the condition at line 54 uses a five-argument version of `String` method `regionMatches` to compare portions of two strings for equality. When the first argument is true, the method ignores the case of the characters being compared. The remaining arguments are identical to those described for the four-argument `regionMatches` method.

The second example in this section (Fig. 10.4) demonstrates `String` methods **startsWith** and **endsWith**. Method `main` creates array `strings` containing the strings `"started"`, `"starting"`, `"ended"` and `"ending"`. The remainder of method `main` consists of three for statements that test the elements of the array to determine whether they start with or end with a particular set of characters.

```
 1  // Fig. 10.4: StringStartEnd.java
 2  // String methods startsWith and endsWith.
 3
 4  public class StringStartEnd
 5  {
 6     public static void main( String args[] )
 7     {
 8        String strings[] = { "started", "starting", "ended", "ending" };
 9
10        // test method startsWith
11        for ( String string : strings )
12        {
13           if ( string.startsWith( "st" ) )
14              System.out.printf( "\"%s\" starts with \"st\"\n", string );
15        } // end for
16
17        System.out.println();
18
19        // test method startsWith starting from position 2 of string
20        for ( String string : strings )
```

```
21         {
22            if ( string.startsWith( "art", 2 ) )
23               System.out.printf(
24                  "\"%s\" starts with \"art\" at position 2\n", string );
25         } // end for
26
27         System.out.println();
28
29         // test method endsWith
30         for ( String string : strings )
31         {
32            if ( string.endsWith( "ed" ) )
33               System.out.printf( "\"%s\" ends with \"ed\"\n", string );
34         } // end for
35      } // end main
36   } // end class StringStartEnd
```

```
"started" starts with "st"
"starting" starts with "st"

"started" starts with "art" at position 2
"starting" starts with "art" at position 2

"started" ends with "ed"
"ended" ends with "ed"
```

WEEK 11

General Learning Objectives for Week11: Arrays

Specific Objectives:

a. Manipulate a set of data values using arrays
b. Declare and use arrays of primitive types
c. Declare and use arrays of objects

This week we will introduce an important topic of **data structures** collections of related data items. **Arrays** are data structures consisting of related data items of the same type. Arrays are fixed-length entitiesthey remain the same length once they are created, although an array variable may be reassigned such that it refers to a new array of a different length.

Arrays

An array is a group of variables (called **elements** or **components**) containing values that all have the same type. Recall that types are divided into two categories primitive types and reference types. Arrays are objects, so they are considered reference types. As you will soon see, what we typically think of as an array is actually a reference to an array object in memory. The elements of an array can be either primitive types or reference types. To refer to a particular element in an array, we specify the name of the reference to the array and the position number of the element in the array. The position number of the element is called the element's **index** or **subscript**

Figure 11.1 shows a logical representation of an integer array called `c`. This array contains 12 elements. A program refers to any one of these elements with an **array-access expression** that includes the name of the array followed by the index of the particular element in **square brackets** (`[]`). The first element in every array has **index zero** and is sometimes called the **zeroth element**. Thus, the elements of array `c` are `c[ 0 ]`, `c[ 1 ]`, `c[ 2 ]` and so on. The highest index in array `c` is 11, which is 1 less than 12 the number of elements in the array. Array names follow the same conventions as other variable names.

c[0]	11
c[1]	-34
c[2]	100
c[3]	32
c[4]	98
c[5]	-987
c[6]	112
c[7]	2309
c[8]	7
c[9]	8
c[10]	23
c[11]	1

Figure 11.1

An index must be a nonnegative integer. A program can use an expression as an index. For example, if we assume that variable a is `5` and variable `b` is `6`, then the statement

```
c[ a + b ] += 2;
```

adds `2` to array element `c[ 11 ]`. Note that an indexed array name is an array-access expression. Such expressions can be used on the left side of an assignment to place a new value into an array element.

Let us examine array `c` in Fig. 7.1 more closely. The **name** of the array is `c`. Every array object knows its own length and maintains this information in a **`length`** field. The expression `c.length` accesses array `c`'s `length` field to determine the length of the array. Note that, even though the `length` member of an array is `public`, it cannot be changed because it is a `final` variable. This array's 12 elements are referred to as `c[ 0 ]`, `c[ 1 ]`, `c[ 2 ]`, ..., `c[ 11 ]`. The value of `c[ 0 ]` is `-45`, the value of `c[ 1 ]` is `6`, the value of `c[ 2 ]` is `0`, the value of `c[ 7 ]` is `62` and the value of `c[ 11 ]` is `78`. To calculate the sum of the values contained in the first three elements of array `c` and store the result in variable `sum`, we would write

```
sum = c[ 0 ] + c[ 1 ] + c[ 2 ];
```

To divide the value of `c[ 6 ]` by `2` and assign the result to the variable `x`, we would write

```
x = c[ 6 ] / 2;
```

Declaring and Creating Arrays

Array objects occupy space in memory. Like other objects, arrays are created with keyword `new`. To create an array object, the programmer specifies the type of the array elements and the number of elements as part of an **array-creation expression** that uses keyword `new`. Such an expression returns a reference that can be stored in an array variable. The following declaration and array-creation expression create an array object containing 12 `int` elements and store the array's reference in variable `c`:

```
int c[] = new int[ 12 ];
```

This expression can be used to create the array shown in Fig. 7.1. This task also can be performed in two steps as follows:

```
int c[];             // declare the array variable
c = new int[ 12 ];   // create the array; assign to array variable
```

In the declaration, the square brackets following the variable name `c` indicate that `c` is a variable that will refer to an array (i.e., the variable will store an array reference). In the assignment statement, the array variable `c` receives the reference to a new array of 12 `int` elements. When an array is created, each element of the array receives a default valuezero for the numeric primitive-type elements, `false` for `boolean` elements and `null` for references (any nonprimitive type). As we will soon see, we can provide specific, nondefault initial element values when we create an array.

A program can create several arrays in a single declaration. The following `String` array declaration reserves 100 elements for `b` and 27 elements for `x`:

```
String b[] = new String[ 100 ], x[] = new String[ 27 ];
```

In this case, the class name `String` applies to each variable in the declaration. For readability, we prefer to declare only one variable per declaration, as in:

```
String b[] = new String[ 100 ]; // create array b
String x[] = new String[ 27 ]; // create array x
```

When an array is declared, the type of the array and the square brackets can be combined at the beginning of the declaration to indicate that all the identifiers in the declaration are array variables. For example, the declaration

```
double[] array1, array2;
```

indicates that `array1` and `array2` are "array of `double`" variables. The preceding declaration is equivalent to:

```
double array1[];
double array2[];
```

or

```
double[] array1;
double[] array2;
```

The preceding pairs of declarations are equivalentwhen only one variable is declared in each declaration, the square brackets can be placed either after the type or after the array variable name.

A program can declare arrays of any type. Every element of a primitive-type array contains a value of the array's declared type. Similarly, in an array of a reference type, every element is a reference to an object of the array's declared type. For example, every element of an `int` array is an int value, and every element of a `String` array is a reference to a `String` object.

Examples Using Arrays

This section presents several examples that demonstrate declaring arrays, creating arrays, initializing arrays and manipulating array elements.

Creating and Initializing an Array

The application of Fig. 11.2 uses keyword `new` to create an array of 10 `int` elements, which are initially zero (the default for `int` variables).

Figure 11.2. Initializing the elements of an array to default values of zero.

```
 1  // Fig. 11.2: InitArray.java
 2  // Creating an array.
 3
 4  public class InitArray
 5  {
 6     public static void main( String args[] )
 7     {
 8        int array[]; // declare array named array
 9
10        array = new int[ 10 ]; // create the space for array
11
12        System.out.printf( "%s%8s\n", "Index", "Value" ); //column headings
13
14        // output each array element's value
```

```
15         for ( int counter = 0; counter < array.length; counter++ )
16            System.out.printf( "%5d%8d\n", counter, array[ counter ] );
17      } // end main
18   } // end class InitArray
```

```
Index   Value
    0       0
    1       0
    2       0
    3       0
    4       0
    5       0
    6       0
    7       0
    8       0
    9       0
```

Line 8 declares `array`a reference capable of referring to an array of `int` elements. Line 10 creates the array object and assigns its reference to variable `array`. Line 12 outputs the column headings. The first column contains the index (09) of each array element, and the second column contains the default value (0) of each array element.

The `for` statement in lines 1516 outputs the index number (represented by `counter`) and value of each array element (represented by `array[ counter ]`). Note that the loop control variable `counter` is initially `0`index values start at 0, so using zero-based counting allows the loop to access every element of the array. The `for`'s loop-continuation condition uses the expression `array`. `length` (line 15) to determine the length of the array. In this example, the length of the array is 10, so the loop continues executing as long as the value of control variable `counter` is less than 10. The highest index value of a 10-element array is 9, so using the less-than operator in the loop-continuation condition guarantees that the loop does not attempt to access an element beyond the end of the array (i.e., during the final iteration of the loop, `counter` is `9`). We will soon see what Java does when it encounters such an out-of-range index at execution time.

Using an Array Initializer

A program can create an array and initialize its elements with an **array initializer**, which is a comma-separated list of expressions (called an **initializer list**) enclosed in braces (`{` and `}`). In

this case, the array length is determined by the number of elements in the initializer list. For example, the declaration

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

creates a five-element array with index values `0`, `1`, `2`, `3` and `4`. Element `n[ 0 ]` is initialized to `10`, `n[ 1 ]` is initialized to `20`, and so on. This declaration does not require `new` to create the array object. When the compiler encounters an array declaration that includes an initializer list, the compiler counts the number of initializers in the list to determine the size of the array, then sets up the appropriate `new` operation "behind the scenes."

The application in Fig. 7.3 initializes an integer array with 10 values (line 9) and displays the array in tabular format. The code for displaying the array elements (lines 1415) is identical to that in Fig. 7.2 (lines 1516).

```
 1  // Fig. 7.3: InitArray.java
 2  // Initializing the elements of an array with an array initializer.
 3
 4  public class InitArray
 5  {
 6     public static void main( String args[] )
 7     {
 8        // initializer list specifies the value for each element
 9        int array[] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10
11        System.out.printf( "%s%8s\n", "Index", "Value" ); //column headings
12
13        // output each array element's value
14        for ( int counter = 0; counter < array.length; counter++ )
15           System.out.printf( "%5d%8d\n", counter, array[ counter ] );
16     } // end main
17  } // end class InitArray
```

```
Index   Value
    0      32
    1      27
    2      64
    3      18
    4      95
    5      14
    6      90
    7      70
    8      60
    9      37
```

Calculating a Value to Store in Each Array Element

Some programs calculate the value stored in each array element. The application in Fig. 11.4 creates a 10-element array and assigns to each element one of the even integers from 2 to 20 (2, 4, 6, ..., 20). Then the application displays the array in tabular format. The for statement at lines 1213 calculates an array element's value by multiplying the current value of the for loop's control variable counter by 2, then adding 2.

```
 1  // Fig. 11.4: InitArray.java
 2  // Calculating values to be placed into elements of an array.
 3
 4  public class InitArray
 5  {
 6     public static void main( String args[] )
 7     {
 8        final int ARRAY_LENGTH = 10; // declare constant
 9        int array[] = new int[ ARRAY_LENGTH ]; // create array
10
11        // calculate value for each array element
12        for ( int counter = 0; counter < array.length; counter++ )
13           array[ counter ] = 2 + 2 * counter;
14
15        System.out.printf( "%s%8s\n", "Index", "Value" ); //column headings
16
17        // output each array element's value
18        for ( int counter = 0; counter < array.length; counter++ )
19           System.out.printf( "%5d%8d\n", counter, array[ counter ] );
20     } // end main
21  } // end class InitArray
```

```
Index   Value
    0       2
    1       4
    2       6
    3       8
    4      10
    5      12
    6      14
    7      16
    8      18
    9      20
```

Line 8 uses the modifier **`final`** to declare the **constant variable** `ARRAY_LENGTH`, whose value is `10`. Constant variables (also known as `final` variables) must be initialized before they are used and cannot be modified thereafter. If an attempt is made to modify a `final` variable after it is initialized in its declaration (as in line 8), the compiler issues the error message

`cannot assign a value to final variable` variableName

If an attempt is made to access the value of a `final` variable before it is initialized, the compiler issues the error message

`variable` variableName `might not have been initialized`

Summing the Elements of an Array

Often, the elements of an array represent a series of values to be used in a calculation. For example, if the elements of an array represent exam grades, a professor may wish to total the elements of the array and use that sum to calculate the class average for the exam. The examples using class `GradeBook` later in the chapter, namely Fig. 7.14 and Fig. 7.18, use this technique.

The application in Fig. 11.5 sums the values contained in a 10-element integer array. The program declares, creates and initializes the array at line 8. The `for` statement performs the calculations. [Note: The values supplied as array initializers are often read into a program rather than specified in an initializer list. For example, an application could input the values from a user or from a file on disk. Reading the data into a program makes the program more reusable, because it can be used with different sets of data.]

```
 1  // Fig. 11.5: SumArray.java
 2  // Computing the sum of the elements of an array.
 3
 4  public class SumArray
 5  {
 6     public static void main( String args[] )
 7     {
 8        int array[] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
 9        int total = 0;
10
11        // add each element's value to total
12        for ( int counter = 0; counter < array.length; counter++ )
13           total += array[ counter ];
14
15        System.out.printf( "Total of array elements: %d\n", total );
16     } // end main
17  } // end class SumArray
```

```
Total of array elements: 849
```

WEEK 12

General Learning Objectives for Week12: Event Driven Programs

Specific Objectives:

a. Understand the concepts of event driven programs
b. Understand how to place objects on a frame
c. Write simple event drive programs

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." Providing different applications with consistent, intuitive user interface components allows users to be somewhat familiar with an application, so that they can learn it more quickly and use it more productively.

GUIs are built from **GUI components**. These are sometimes called controls or widgets short for window gadgets in other languages. 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.

Overview of Swing Components

Though it is possible to perform input and output using the `JOptionPane` dialogs presented in earlier weeks, most GUI applications require more elaborate, customized user interfaces. The remainder of this text discusses many GUI components that enable application developers to create robust GUIs. Figure 13.4 lists several **Swing GUI components** from package **`javax.swing`** that are used to build Java GUIs. Most Swing components are pure Java componentsthey are written, manipulated and displayed completely in Java. They are part of the Java Foundation Classes (JFC)Java's libraries for cross-platform GUI development. Visit java.sun.com/products/jfc for more information on JFC.

Figure 13.1 Some basic GUI components.

Component	Description
`JLabel`	Displays uneditable text or icons.
`JTextField`	Enables user to enter data from the keyboard. Can also be used to display editable or uneditable text.
`JButton`	Triggers an event when clicked with the mouse.
`JCheckBox`	Specifies an option that can be selected or not selected.
`JComboBox`	Provides a drop-down list of items from which the user can make a selection by

Figure 13.1 Some basic GUI components.

Component	Description
	clicking an item or possibly by typing into the box.
`JList`	Provides a list of items from which the user can make a selection by clicking on any item in the list. Multiple elements can be selected.
`JPanel`	Provides an area in which components can be placed and organized. Can also be used as a drawing area for graphics.

Displaying Text and Images in a Window

Our next example introduces a framework for building GUI applications. This framework uses several concepts that you will see in many of our GUI applications. This is our first example in which the application appears in its own window. Most windows you will create are an instance of class `JFrame` or a subclass of `JFrame`. `JFrame` provides the basic attributes and behaviors of a windowa title bar at the top of the window, and buttons to minimize, maximize and close the window. Since an application's GUI is typically specific to the application, most of our examples will consist of two classesa subclass of `JFrame` that helps us demonstrate new GUI concepts and an application class in which `main` creates and displays the application's primary window.

Labeling GUI Components

A typical GUI consists of many components. In a large GUI, it can be difficult to identify the purpose of every component unless the GUI designer provides text instructions or information stating the purpose of each component. Such text is known as a label and is created with class **`JLabel`**a subclass of `JComponent`. A `JLabel` displays a single line of read-only text, an image, or both text and an image. Applications rarely change a label's contents after creating it.

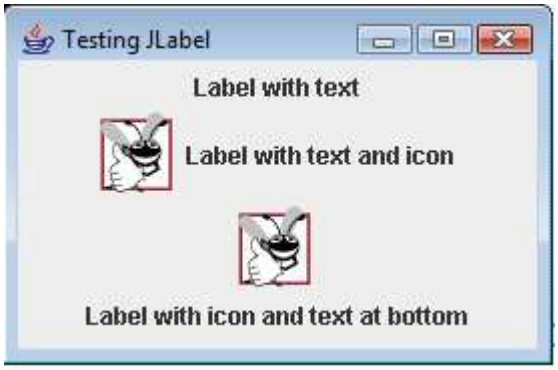
The application of Fig. 13.2 and Fig.13.3 demonstrates several `JLabel` features and presents the framework we use in most of our GUI examples. We did not highlight the code in this example since most of it is new. [Note: There are many more features for each GUI component than we

can cover in our examples. To learn the complete details of each GUI component, visit its page in the online documentation.

```
1  // Fig. 13.2: LabelFrame.java
2  // Demonstrating the JLabel class.
3  import java.awt.FlowLayout; // specifies how components are arranged
4  import javax.swing.JFrame; // provides basic window features
5  import javax.swing.JLabel; // displays text and images
6  import javax.swing.SwingConstants; // common constants used with Swing
7  import javax.swing.Icon; // interface used to manipulate images
8  import javax.swing.ImageIcon; // loads images
9
10 public class LabelFrame extends JFrame
11 {
12    private JLabel label1; // JLabel with just text
13    private JLabel label2; // JLabel constructed with text and icon
14    private JLabel label3; // JLabel with added text and icon
15
16    // LabelFrame constructor adds JLabels to JFrame
17    public LabelFrame()
18    {
19       super( "Testing JLabel" );
20       setLayout( new FlowLayout() ); // set frame layout
21
22       // JLabel constructor with a string argument
23       label1 = new JLabel( "Label with text" );
24       label1.setToolTipText( "This is label1" );
25       add( label1 ); // add label1 to JFrame
26
27       // JLabel constructor with string, Icon and alignment arguments
28       Icon bug = new ImageIcon( getClass().getResource( "bug1.gif" ) );
29       label2 = new JLabel( "Label with text and icon", bug,
30          SwingConstants.LEFT );
31       label2.setToolTipText( "This is label2" );
32       add( label2 ); // add label2 to JFrame
33
34       label3 = new JLabel(); // JLabel constructor no arguments
35       label3.setText( "Label with icon and text at bottom" );
36       label3.setIcon( bug ); // add icon to JLabel
37       label3.setHorizontalTextPosition( SwingConstants.CENTER );
38       label3.setVerticalTextPosition( SwingConstants.BOTTOM );
39       label3.setToolTipText( "This is label3" );
40       add( label3 ); // add label3 to JFrame
41    } // end LabelFrame constructor
42 } // end class LabelFrame
```

```
1  // Fig. 13.3: LabelTest.java
2  // Testing LabelFrame.
3  import javax.swing.JFrame;
4
5  public class LabelTest
6  {
```

```
 7     public static void main( String args[] )
 8     {
 9        LabelFrame labelFrame = new LabelFrame(); // create LabelFrame
10        labelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        labelFrame.setSize( 275, 180 ); // set frame size
12        labelFrame.setVisible( true ); // display frame
13     } // end main
14  } // end class LabelTest
```



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 interactionknown as an **event**drives the program to perform a task. Some common events (user interactions) that might cause an application to perform a task include clicking a button, typing in a text field, selecting an item from a menu, closing a window and moving the mouse. The code that performs a task in response to an event is called an **event handler** and the overall process of responding to events is known as **event handling**.

In this section, we introduce two new GUI components that can generate events**`JTextFields`** and **`JPasswordFields`** (package `javax.swing`). Class `JTextField` extends class **`JTextComponent`** (package `javax.swing.text`), which provides many features common to Swing's text-based components. Class `JPasswordField` extends `JTextField` and adds several

methods that are specific to processing passwords. Each of these components is a single-line area in which the user can enter text via the keyboard. Applications can also display text in a `JTextField` (see the output of Fig. 13.3). A `JPasswordField` shows that characters are being typed as the user enters them, but hides the actual characters with an echo character, assuming that they represent a password that should remain known only to the user.

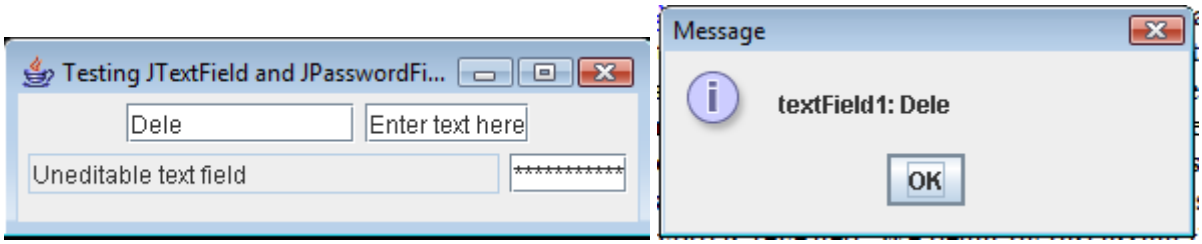
When the user types data into a `JTextField` or a `JPasswordField`, then presses Enter, an event occurs. Our next example demonstrates how a program can perform a task when that event occurs. The techniques shown here are applicable to all GUI components that generate events.

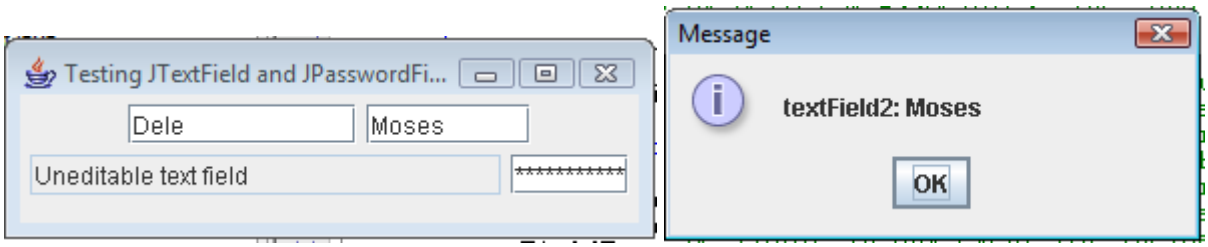
The application of Fig. 13.2 and Fig. 13.3 uses classes `JTextField` and `JPasswordField` to create and manipulate four text fields. When the user types in one of the text fields, then presses Enter, the application displays a message dialog box containing the text the user typed. You can only type in the text field that is "in **focus**." A component receives the focus when the user clicks the component. This is important because the text field with the focus is the one that generates an event when the user presses Enter. In this example, when the user presses Enter in the `JPasswordField`, the password is revealed. We begin by discussing the setup of the GUI, then discuss the event-handling code.

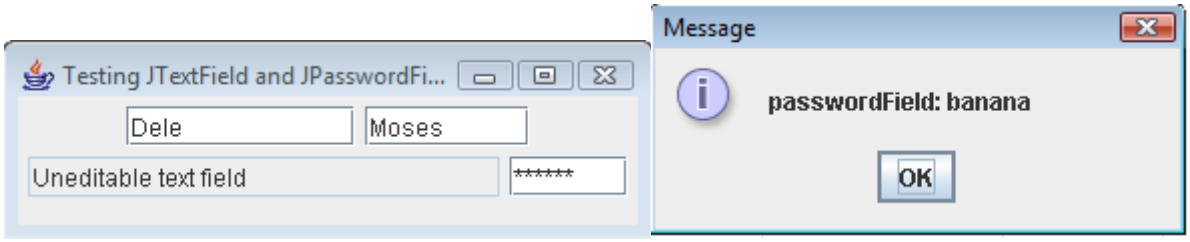
```
 1  // Fig. 13.2: TextFieldFrame.java
 2  // Demonstrating the JTextField class.
 3  import java.awt.FlowLayout;
 4  import java.awt.event.ActionListener;
 5  import java.awt.event.ActionEvent;
 6  import javax.swing.JFrame;
 7  import javax.swing.JTextField;
 8  import javax.swing.JPasswordField;
 9  import javax.swing.JOptionPane;
10
11  public class TextFieldFrame extends JFrame
12  {
13     private JTextField textField1; // text field with set size
14     private JTextField textField2; // text field constructed with text
15     private JTextField textField3; // text field with text and size
16     private JPasswordField passwordField; // password field with text
17
18     // TextFieldFrame constructor adds JTextFields to JFrame
19     public TextFieldFrame()
20     {
21        super( "Testing JTextField and JPasswordField" );
22        setLayout( new FlowLayout() ); // set frame layout
23
24        // construct textfield with 10 columns
```

```
25          textField1 = new JTextField( 10 );
26          add( textField1 ); // add textField1 to JFrame
27
28          // construct textfield with default text
29          textField2 = new JTextField( "Enter text here" );
30          add( textField2 ); // add textField2 to JFrame
31
32          // construct textfield with default text and 21 columns
33          textField3 = new JTextField( "Uneditable text field", 21 );
34          textField3.setEditable( false ); // disable editing
35          add( textField3 ); // add textField3 to JFrame
36
37          // construct passwordfield with default text
38          passwordField = new JPasswordField( "Hidden text" );
39          add( passwordField ); // add passwordField to JFrame
40
41          // register event handlers
42          TextFieldHandler handler = new TextFieldHandler();
43          textField1.addActionListener( handler );
44          textField2.addActionListener( handler );
45          textField3.addActionListener( handler );
46          passwordField.addActionListener( handler );
47       } // end TextFieldFrame constructor
48
49       // private inner class for event handling
50       private class TextFieldHandler implements ActionListener
51       {
52          // process text field events
53          public void actionPerformed( ActionEvent event )
54          {
55             String string = ""; // declare string to display
56
57             // user pressed Enter in JTextField textField1
58             if ( event.getSource() == textField1 )
59                string = String.format( "textField1: %s",
60                   event.getActionCommand() );
61
62             // user pressed Enter in JTextField textField2
63             else if ( event.getSource() == textField2 )
64                string = String.format( "textField2: %s",
65                   event.getActionCommand() );
66
67             // user pressed Enter in JTextField textField3
68             else if ( event.getSource() == textField3 )
69                string = String.format( "textField3: %s",
70                   event.getActionCommand() );
71
72             // user pressed Enter in JTextField passwordField
73             else if ( event.getSource() == passwordField )
74                string = String.format( "passwordField: %s",
75                   new String( passwordField.getPassword() ) );
76
77             // display JTextField content
78             JOptionPane.showMessageDialog( null, string );
79          } // end method actionPerformed
80       } // end private inner class TextFieldHandler
81    } // end class TextFieldFrame
```

```
 1  // Fig. 13.3: TextFieldTest.java
 2  // Testing TextFieldFrame.
 3  import javax.swing.JFrame;
 4
 5  public class TextFieldTest
 6  {
 7     public static void main( String args[] )
 8     {
 9        TextFieldFrame textFieldFrame = new TextFieldFrame();
10        textFieldFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        textFieldFrame.setSize( 325, 100 ); // set frame size
12        textFieldFrame.setVisible( true ); // display frame
13     } // end main
14  } // end class TextFieldTest
```







Lines 39 import the classes and interfaces we use in this example. Class `TextFieldFrame` extends `JFrame` and declares three `JTextField` variables and a `JPasswordField` variable (lines 13-16). Each of the corresponding text fields is instantiated and attached to the `TextFieldFrame` in the constructor (lines 1947).

Creating the GUI

Line 22 sets the layout of the `TextFieldFrame` to `FlowLayout`. Line 25 creates `textField1` with `10` columns of text. The width in pixels of a text column is determined by the average width of a character in the text field's current font. When text is displayed in a text field and the text is wider than the text field itself, a portion of the text at the right side is not visible. If you are typing in a text field and the cursor reaches the right edge of the text field, the text at the left edge is pushed off the left side of the text field and will no longer be visible. Users can use the left and right arrow keys to move through the complete text even though the entire text will not be visible at one time. Line 26 adds `textField1` to the `JFrame`.

Line 29 creates `textField2` with the initial text `"Enter text here"` to display in the text field. The width of the text field is determined by the width of the default text specified in the constructor. Line 30 adds `textField2` to the `JFrame`.

Line 33 creates `textField3` and calls the `JTextField` constructor with two argumentsthe default text `"Uneditable text field"` to display and the number of columns (`21`). The width of the text field is determined by the number of columns specified. Line 34 uses method **`setEditable`** (inherited by `JTextField` from class `JTextComponent`) to make the text field uneditablei.e., the user cannot modify the text in the text field. Line 35 adds `textField3` to the `JFrame`.

Line 38 creates `passwordField` with the text `"Hidden text"` to display in the text field. The width of the text field is determined by the width of the default text. When you execute the application, notice that the text is displayed as a string of asterisks. Line 39 adds `passwordField` to the `JFrame`.

Steps Required to Set Up Event Handling for a GUI Component

This example should display a message dialog containing the text from a text field when the user presses Enter in that text field. Before an application can respond to an event for a particular GUI component, you must perform several coding steps:

1. Create a class that represents the event handler.

2. Implement an appropriate interface, known as an **event-listener interface**, in the class from Step 1.

3. Indicate that an object of the class from Steps 1 and 2 should be notified when the event occurs. This is known as **registering the event handler**.

Using a Nested Class to Implement an Event Handler

All the classes discussed so far were so-called **top-level classes**that is, the classes were not declared inside another class. Java allows you to declare classes inside other classesthese are called **nested classes**. Nested classes can be `static` or `non-static`. Non-`static` nested classes are called **inner classes** and are frequently used for event handling.

Before an object of an inner class can be created, there must first be an object of the top-level class that contains the inner class. This is required because an inner-class object implicitly has a reference to an object of its top-level class. There is also a special relationship between these objectsthe inner-class object is allowed to directly access all the instance variables and methods of the outer class. A nested class that is `static` does not require an object of its top-level class and does not implicitly have a reference to an object of the top-level class.

The event handling in this example is performed by an object of the `private` inner class `TextFieldHandler` (lines 5080). This class is `private` because it will be used only to create event handlers for the text fields in top-level class `TextFieldFrame`. As with other members of a class, inner classes can be declared `public`, `protected` or `private`.

GUI components can generate a variety of events in response to user interactions. Each event is represented by a class and can be processed only by the appropriate type of event handler. In most cases, the events a GUI component supports are described in the Java API documentation for that component's class and its superclasses. When the user presses Enter in a `JTextField` or `JPasswordField`, the GUI component generates an **`ActionEvent`** (package `java.awt.event`). Such an event is processed by an object that implements the interface **`ActionListener`** (package

java.awt.event). The information discussed here is available in the Java API documentation for classes JTextField and ActionEvent. Since JPasswordField is a subclass of JTextField, JPasswordField supports the same events.

To prepare to handle the events in this example, inner class TextFieldHandler implements interface ActionListener and declares the only method in that interfaceaction-Performed (lines 53-79). This method specifies the tasks to perform when an ActionEvent occurs. So inner class TextFieldHandler satisfies Steps 1 and 2 listed earlier in this section. We'll discuss the details of method actionPerformed shortly.

Registering the Event Handler for Each Text Field

In the TextFieldFrame constructor, line 42 creates a TextFieldHandler object and assigns it to variable handler. This object's actionPerformed method will be called automatically when the user presses Enter in any of the GUI's text fields. However, before this can occur, the program must register this object as the event handler for each text field. Lines 4346 are the event-registration statements that specify handler as the event handler for the three JTextFields and the JPasswordField. The application calls JTextField method **addActionListener** to register the event handler for each component. This method receives as its argument an ActionListener object, which can be an object of any class that implements ActionListener. The object handler is an ActionListener, because class TextFieldHandler implements ActionListener. After lines 43-46 execute, the object handler listens for events. Now, when the user presses Enter in any of these four text fields, method actionPerformed (line 5379) in class TextFieldHandler is called to handle the event. If an event handler is not registered for a particular text field, the event that occurs when the user presses Enter in that text field is consumedi.e., it is simply ignored by the application.

Details of Class TextFieldHandler's actionPerformed Method

In this example, we are using one event-handling object's actionPerformed method (lines 5379) to handle the events generated by four text fields. Since we'd like to output the name of each text field's instance variable for demonstration purposes, we must determine which text field generated the event each time actionPerformed is called. The GUI component with which the

user interacts is the **event source**. In this example, the event source is one of the text fields or the password field. When the user presses Enter while one of these GUI components has the focus, the system creates a unique `ActionEvent` object that contains information about the event that just occurred, such as the event source and the text in the text field. The system then passes this `ActionEvent` object in a method call to the event listener's `actionPerformed` method. In this example, we display some of that information in a message dialog. Line 55 declares the `String` that will be displayed. The variable is initialized with the empty stringa string containing no characters. The compiler requires this in case none of the branches of the nested `if` in lines 5875 executes.

`ActionEvent` method `getSource` (called in lines 58, 63, 68 and 73) returns a reference to the event source. The condition in line 58 asks, "Is the event source `textField1`?" This condition compares the references on either side of the `==` operator to determine whether they refer to the same object. If they both refer to `textField1`, then the program knows that the user pressed Enter in `textField1`. In this case, lines 59-60 create a `String` containing the message that line 78 will display in a message dialog. Line 60 uses `ActionEvent` method **`getActionCommand`** to obtain the text the user typed in the text field that generated the event.

If the user interacted with the `JPasswordField`, lines 7475 use `JPasswordField` method **`getPassword`** to obtain the password and create the `String` to display. This method returns the password as an array of type `char` that is used as an argument to a `String` constructor to create a string containing the characters in the array.

WEEK 14

General Learning Objectives for Week14: Inheritance

Specific Objectives:

a. Understand the concepts of inheritance
b. Understand of the is-a and has-a relationship in inheritance hierarchy
c. Write simple Java programs implementing inheritance

Introduction to Inheritance

This week we will discuss one of the primary features of object-oriented programming (OOP) that is **inheritance**, which is a form of software reuse in which a new class is created by absorbing an existing class's members and embellishing them with new or modified capabilities. With inheritance, programmers save time during program development by reusing proven and debugged high-quality software. This also increases the likelihood that a system will be implemented effectively.

When creating a class, rather than declaring completely new members, the programmer can designate that the new class should inherit the members of an existing class. The existing class is called the **superclass**, and the new class is the **subclass**. (The C++ programming language refers to the superclass as the **base class** and the subclass as the **derived class**.) Each subclass can become the superclass for future subclasses.

A subclass normally adds its own fields and methods. Therefore, a subclass is more specific than its superclass and represents a more specialized group of objects. Typically, the subclass exhibits the behaviors of its superclass and additional behaviors that are specific to the subclass.

The **direct superclass** is the superclass from which the subclass explicitly inherits. An **indirect superclass** is any class above the direct superclass in the **class hierarchy**, which defines the inheritance relationships between classes. In Java, the class hierarchy begins with class `Object` (in package `java.lang`), which every class in Java directly or indirectly **extends** (or "inherits from"). In the case of **single inheritance**, a class is derived from one direct superclass. Java, unlike C++, does not support multiple inheritance (which occurs when a class is derived from more than one direct superclass).

Experience in building software systems indicates that significant amounts of code deal with closely-related special cases. When programmers are preoccupied with special cases, the details can obscure the big picture. With object-oriented programming, programmers focus on the commonalities among objects in the system rather than on the special cases.

We distinguish between the **"is-a" relationship** and the **"has-a" relationship**. "Is-a" represents inheritance. In an "is-a" relationship, an object of a subclass can also be treated as an object of its superclass. For example, a car is a vehicle. By contrast, "has-a" represents composition. In a "has-a" relationship, an object contains one or more object references as members. For example, a car has a steering wheel (and a car object has a reference to a steering wheel object).

New classes can inherit from classes in **class libraries.** Organizations develop their own class libraries and can take advantage of others available worldwide. Some day, most new software likely will be constructed from **standardized reusable components**, just as automobiles and most computer hardware are constructed today. This will facilitate the development of more powerful, abundant and economical software.

Superclasses and Subclasses

Often, an object of one class "is an" object of another class as well. For example, in geometry, a rectangle is a quadrilateral (as are squares, parallelograms and trapezoids). Thus, in Java, class `Rectangle` can be said to inherit from class `Quadrilateral`. In this context, class `Quadrilateral` is a superclass and class `Rectangle` is a subclass. A rectangle is a specific type of quadrilateral, but it is incorrect to claim that every quadrilateral is a rectanglethe quadrilateral could be a parallelogram or some other shape. Figure 14.1 lists several simple examples of superclasses and subclassesnote that superclasses tend to be "more general" and subclasses tend to be "more specific."

Figure 14.1. Inheritance examples.

Superclass	Subclasses
`Student`	`GraduateStudent`, `UndergraduateStudent`
`Shape`	`Circle`, `TRiangle`, `Rectangle`
`Loan`	`CarLoan`, `HomeImprovementLoan`, `MortgageLoan`
`Employee`	`Faculty`, `Staff`
`BankAccount`	`CheckingAccount`, `SavingsAccount`

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 typically 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.

Inheritance relationships form tree-like hierarchical structures. A superclass exists in a hierarchical relationship with its subclasses. When classes participate in inheritance relationships, they become "affiliated" with other classes. A class becomes either a superclass, supplying members to other classes, or a subclass, inheriting its members from other classes. In some cases, a class is both a superclass and a subclass.

Let us develop a sample class hierarchy also called an **inheritance hierarchy**. A university community has thousands of members, including employees, students and alumni. Employees are either faculty members or staff members. Faculty members are either administrators (such as deans and department chairpersons) or teachers. Note that 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.

Now consider the `Shape` inheritance hierarchy. This hierarchy begins with superclass `Shape`, which is extended by subclasses `TwoDimensionalShape` and `ThreeDimensionalShapeShapes` are either `TwoDimensionalShapes` or `ThreeDimensionalShapes`. The third level of this hierarchy contains some more specific types of `TwoDimensionalShapes` and `THReeDimensionalShapes`. From the bottom of the hierarchy to the topmost superclass in this class hierarchy to identify several "is-a" relationships. For instance, a `triangle` is a `TwoDimensionalShape` and is a `Shape`, while a `Sphere` is a `ThreeDimensionalShape` and is a `Shape`. Note that this hierarchy could contain many other classes. For example, ellipses and trapezoids are `TwoDimensionalShapes`.

Not every class relationship is an inheritance relationship. The "has-a" relationship, in which classes have members that are references to objects of other classes. Such relationships create classes by composition of existing classes. For example, given the classes `Employee`, `BirthDate`

and `TelephoneNumber`, it is improper to say that an `Employee` is a `BirthDate` or that an `Employee` is a `TelephoneNumber`. However, an `Employee` has a `BirthDate`, and an `Employee` has a `TelephoneNumber`.

It is possible to treat superclass objects and subclass objects similarly their commonalities are expressed in the members of the superclass. Objects of all classes that extend a common superclass can be treated as objects of that superclass (i.e., such objects have an "is-a" relationship with the superclass). However, superclass objects cannot be treated as objects of their subclasses. For example, all cars are vehicles, but not all vehicles are cars (the other vehicles could be trucks, planes or bicycles, for example).

One problem with inheritance is that a subclass can inherit methods that it does not need or should not have. Even when a superclass method is appropriate for a subclass, that subclass often needs a customized version of the method. In such cases, the subclass can **override** (redefine) the superclass method with an appropriate implementation, as we will see often in the chapter's code examples.

Relationship between Superclasses and Subclasses

In this section, we 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 commission employees and base-salaried commission employees into five examples. The first example 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 the latter class by writing every line of code the class requireswe will soon see that it is much more efficient to create this class by inheriting from class `CommissionEmployee`.

The third example declares a separate `BasePlusCommissionEmployee2` class that extends class `CommissionEmployee` (i.e., a `BasePlusCommissionEmployee2` is a `CommissionEmployee` who also has a base salary) and attempts to access class `CommissionEmployee`'s `private` membersthis 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`, a `BasePlusCommissionEmployee3` class that extends class `CommissionEmployee2` can access that data directly. For this purpose, we declare class `CommissionEmployee2` with `protected` instance variables. Both of the `BasePlusCommissionEmployee` classes contain identical functionality, but we show how the class `BasePlusCommissionEmployee3` 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` in class `CommissionEmployee3` to enforce good software engineering. Then we show how a separate `BasePlusCommissionEmployee4` class, which extends class `CommissionEmployee3`, can use `CommissionEmployee3`'s `public` methods to manipulate `CommissionEmployee3`'s `private` instance variables

Creating and Using a `CommissionEmployee` Class

We begin by declaring class `CommissionEmployee` (Fig. 14.1). Line 4 begins the class declaration and indicates that class `CommissionEmployee` extends (i.e., inherits from) class `Object` (from package `java.lang`). Java programmers use inheritance to create classes from existing classes. In fact, every class in Java (except `Object`) extends an existing class. Because class `CommissionEmployee` extends class `Object`, class `CommissionEmployee` inherits the methods of class `Object`class `Object` does not have any fields. In fact, every Java class directly

or indirectly inherits `Object`'s methods. If a class does not specify that it extends another class, the new class implicitly extends `Object`. For this reason, programmers typically do not include "`extends Object`" in their codewe do so in this example for demonstration purposes.

```
 1  // Fig. 14.1 CommissionEmployee.java
 2  // CommissionEmployee class represents a commission employee.
 3
 4  public class CommissionEmployee extends Object
 5  {
 6     private String firstName;
 7     private String lastName;
 8     private String socialSecurityNumber;
 9     private double grossSales; // gross weekly sales
10     private double commissionRate; // commission percentage
11
12    // five-argument constructor
13    public CommissionEmployee( String first, String last, String ssn,
14       double sales, double rate )
15    {
16       // implicit call to Object constructor occurs here
17       firstName = first;
18       lastName = last;
19       socialSecurityNumber = ssn;
20       setGrossSales( sales ); // validate and store gross sales
21       setCommissionRate( rate ); // validate and store commission rate
22    } // end five-argument CommissionEmployee constructor
23
24    // set first name
25    public void setFirstName( String first )
26    {
27       firstName = first;
28    } // end method setFirstName
29
30    // return first name
31    public String getFirstName()
32    {
33       return firstName;
34    } // end method getFirstName
35
36    // set last name
37    public void setLastName( String last )
38    {
39       lastName = last;
40    } // end method setLastName
41
42    // return last name
43    public String getLastName()
44    {
45       return lastName;
46    } // end method getLastName
47
48    // set social security number
49    public void setSocialSecurityNumber( String ssn )
50    {
```

```
51          socialSecurityNumber = ssn; // should validate
52       } // end method setSocialSecurityNumber
53
54       // return social security number
55       public String getSocialSecurityNumber()
56       {
57          return socialSecurityNumber;
58       } // end method getSocialSecurityNumber
59
60       // set gross sales amount
61       public void setGrossSales( double sales )
62       {
63          grossSales = ( sales < 0.0 ) ? 0.0 : sales;
64       } // end method setGrossSales
65
66       // return gross sales amount
67       public double getGrossSales()
68       {
69          return grossSales;
70       } // end method getGrossSales
71
72       // set commission rate
73       public void setCommissionRate( double rate )
74       {
75          commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
76       } // end method setCommissionRate
77
78       // return commission rate
79       public double getCommissionRate()
80       {
81          return commissionRate;
82       } // end method getCommissionRate
83
84       // calculate earnings
85       public double earnings()
86       {
87          return commissionRate * grossSales;
88       } // end method earnings
89
90       // return String representation of CommissionEmployee object
91       public String toString()
92       {
93          return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
94             "commission employee", firstName, lastName,
95             "social security number", socialSecurityNumber,
96             "gross sales", grossSales,
97             "commission rate", commissionRate );
98       } // end method toString
99    } // end class CommissionEmployee
```

The `public` services of class `CommissionEmployee` include a constructor (lines 13-22) and methods `earnings` (lines 85-88) and `toString` (lines 9198). Lines 25-82 declare `public` get and set methods for manipulating the class's instance variables (declared in lines 6-10) `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`. Class

`CommissionEmployee` declares each of its instance variables as `private`, so objects of other classes cannot directly access these variables. Declaring instance variables as `private` and providing get and set methods to manipulate and validate the instance variables helps enforce good software engineering. Methods `setGrossSales` and `setCommissionRate`, for example, validate their arguments before assigning the values to instance variables `grossSales` and `commissionRate`, respectively.

Constructors are not inherited, so class `CommissionEmployee` does not inherit class `Object`'s constructor. However, class `CommissionEmployee`'s constructor calls class `Object`'s constructor implicitly. In fact, the first task of any subclass constructor is to call its direct superclass's constructor, either explicitly or implicitly (if no constructor call is specified), to ensure that the instance variables inherited from the superclass are initialized properly. The syntax for calling a superclass constructor explicitly is discussed later on in the test. If the code does not include an explicit call to the superclass constructor, Java implicitly calls the superclass's default or no-argument constructor. The comment in line 16 of Fig. 14.1 indicates where the implicit call to the superclass `Object`'s default constructor is made (the programmer does not write the code for this call). `Object`'s default (empty) constructor does nothing. Note that even if a class does not have constructors, the default constructor that the compiler implicitly declares for the class will call the superclass's default or no-argument constructor.

After the implicit call to `Object`'s constructor occurs, lines 17-21 of `CommissionEmployee`'s constructor assign values to the class's instance variables. Note that we do not validate the values of arguments `first`, `last` and `ssn` before assigning them to the corresponding instance variables. While validating data is good software engineering, including extensive validation in this class could add a potentially large amount of code that would obscure the focus of this example. We certainly could validate the first and last namesperhaps by ensuring that they are of a reasonable length. Similarly, a social security number could be validated to ensure that it contains nine digits, with or without dashes (e.g., `123-45-6789` or `123456789`).

Method `earnings` (lines 8588) calculates a `CommissionEmployee`'s earnings. Line 87 multiplies the `commissionRate` by the `grossSales` and returns the result.

Method `toString` (lines 91-98) is special it is one of the methods that every class inherits directly or indirectly from class `Object`, which is the root of the Java class hierarchy. Method `toString` returns a `String` representing an object. This method is called implicitly by a program whenever an object must be converted to a string representation, such as when an object is output by `printf` or `String` method `format` using the `%s` format specifier. Class `Object`'s `toString` method returns a `String` that includes the name of the object's class. It is primarily a placeholder that can be overridden by a subclass to specify an appropriate string representation of the data in a subclass object. Method `toString` of class `CommissionEmployee` overrides (redefines) class `Object`'s `toString` method. When invoked, `CommissionEmployee`'s `toString` method uses `String` method `format` to return a `String` containing information about the `CommissionEmployee`. We use format specifier `%.2f` to format both the `grossSales` and the `commissionRate` with two digits of precision to the right of the decimal point. To override a superclass method, a subclass must declare a method with the same signature (method name, number of parameters and parameter types) as the superclass method`Object`'s `toString` method takes no parameters, so `CommissionEmployee` declares `toString` with no parameters.

It is a syntax error to override a method with a more restricted access modifiera `public` method of the superclass cannot become a `protected` or `private` method in the subclass; a `protected` method of the superclass cannot become a `private` method in the subclass. Doing so would break the "is-a" relationship in which it is required that all subclass objects be able to respond to method calls that are made to `public` methods declared in the superclass. If a `public` method could be overridden as a `protected` or `private` method, the subclass objects would not be able to respond to the same method calls as superclass objects. Once a method is declared `public` in a superclass, the method remains `public` for all that class's direct and indirect subclasses.

Figure 14.2 tests class `CommissionEmployee`. Lines 910 instantiate a `CommissionEmployee` object and invoke `CommissionEmployee`'s constructor (lines 13-22 of Fig. 14.1) to initialize it with `"Sue"` as the first name, `"Jones"` as the last name, `"222-22-2222"` as the social security number, 10000 as the gross sales amount and `.06` as the commission rate. Lines 1524 use `CommissionEmployee`'s get methods to retrieve the object's instance variable values for output. Lines 26-27 invoke the object's methods `setGrossSales` and `setCommissionRate` to change the

values of instance variables `grossSales` and `commissionRate`. Lines 29-30 output the string representation of the updated `CommissionEmployee`. Note that when an object is output using the `%s` format specifier, the object's `toString` method is invoked implicitly to obtain the object's string representation.

```
 1  // Fig. 14.2: CommissionEmployeeTest.java
 2  // Testing class CommissionEmployee.
 3
 4  public class CommissionEmployeeTest
 5  {
 6     public static void main( String args[] )
 7     {
 8        // instantiate CommissionEmployee object
 9        CommissionEmployee employee = new CommissionEmployee(
10           "Sue", "Jones", "222-22-2222", 10000, .06 );
11
12        // get commission employee data
13        System.out.println(
14           "Employee information obtained by get methods: \n" );
15        System.out.printf( "%s %s\n", "First name is",
16           employee.getFirstName()  );
17        System.out.printf( "%s %s\n", "Last name is",
18           employee.getLastName()  );
19        System.out.printf( "%s %s\n", "Social security number is",
20           employee.getSocialSecurityNumber()  );
21        System.out.printf( "%s %.2f\n", "Gross sales is",
22           employee.getGrossSales() );
23        System.out.printf( "%s %.2f\n", "Commission rate is",
24           employee.getCommissionRate() );
25
26        employee.setGrossSales( 500 ); // set gross sales
27        employee.setCommissionRate( .1 ); // set commission rate
28
29        System.out.printf( "\n%s:\n\n%s\n",
30           "Updated employee information obtained by toString", employee );
31     } // end main
32  } // end class CommissionEmployeeTest
```

```
Employee information obtained by get methods:

First name is Sue
Last name is Jones
Social security number is 222-22-2222
Gross sales is 10000.00
Commission rate is 0.06

Updated employee information obtained by toString:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 500.00
commission rate: 0.10
```

CommissionEmployeeBasePlusCommissionEmployee Inheritance Hierarchy Using private Instance Variables

We now reexamine our hierarchy once more, this time using the best software engineering practices. Class `CommissionEmployee3` (Fig. 14.3) declares instance variables `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate` as `private` (lines 6-10) and provides `public` methods `setFirstName`, `getFirstName`, `setLastName`, `getLastName`, `setSocialSecurityNumber`, `getSocialSecurityNumber`, `setGrossSales`, `getGrossSales`, `setCommissionRate`, `getCommissionRate`, `earnings` and `toString` for manipulating these values. Note that methods `earnings` (lines 8588) and `toString` (lines 91-98) use the class's get methods to obtain the values of its instance variables. If we decide to change the instance variable names, the earnings and toString declarations will not require modificationonly the bodies of the get and set methods that directly manipulate the instance variables will need to change. Note that these changes occur solely within the superclassno changes to the subclass are needed. Localizing the effects of changes like this is a good software engineering practice. Subclass `BasePlusCommissionEmployee4` (Fig. 14.4) inherits `CommissionEmployee3`'s non-`private` methods and can access the `private` superclass members via those methods.

```
 1  // Fig. 14.3: CommissionEmployee3.java
 2  // CommissionEmployee3 class represents a commission employee.
 3
 4  public class CommissionEmployee3
 5  {
 6     private String firstName;
 7     private String lastName;
 8     private String socialSecurityNumber;
 9     private double grossSales; // gross weekly sales
10     private double commissionRate; // commission percentage
11
12     // five-argument constructor
13     public CommissionEmployee3( String first, String last, String ssn,
14        double sales, double rate )
15     {
16        // implicit call to Object constructor occurs here
17        firstName = first;
18        lastName = last;
19        socialSecurityNumber = ssn;
20        setGrossSales( sales ); // validate and store gross sales
21        setCommissionRate( rate ); // validate and store commission rate
22     } // end five-argument CommissionEmployee3 constructor
23
```

```
24     // set first name
25     public void setFirstName( String first )
26     {
27        firstName = first;
28     } // end method setFirstName
29
30    // return first name
31    public String getFirstName()
32    {
33       return firstName;
34    } // end method getFirstName
35
36    // set last name
37    public void setLastName( String last )
38    {
39       lastName = last;
40    } // end method setLastName
41
42    // return last name
43    public String getLastName()
44    {
45       return lastName;
46    } // end method getLastName
47
48    // set social security number
49    public void setSocialSecurityNumber( String ssn )
50    {
51       socialSecurityNumber = ssn; // should validate
52    } // end method setSocialSecurityNumber
53
54    // return social security number
55    public String getSocialSecurityNumber()
56    {
57       return socialSecurityNumber;
58    } // end method getSocialSecurityNumber
59
60    // set gross sales amount
61    public void setGrossSales( double sales )
62    {
63       grossSales = ( sales < 0.0 ) ? 0.0 : sales;
64    } // end method setGrossSales
65
66    // return gross sales amount
67    public double getGrossSales()
68    {
69       return grossSales;
70    } // end method getGrossSales
71
72    // set commission rate
73    public void setCommissionRate( double rate )
74    {
75       commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
76    } // end method setCommissionRate
77
78    // return commission rate
79    public double getCommissionRate()
80    {
```

```
81          return commissionRate;
82       } // end method getCommissionRate
83
84       // calculate earnings
85       public double earnings()
86       {
87          return getCommissionRate() * getGrossSales();
88       } // end method earnings
89
90       // return String representation of CommissionEmployee3 object
91       public String toString()
92       {
93          return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
94             "commission employee", getFirstName(), getLastName(),
95             "social security number", getSocialSecurityNumber(),
96             "gross sales", getGrossSales(),
97             "commission rate", getCommissionRate() );
98       } // end method toString
99    } // end class CommissionEmployee3
```

```
1  // Fig.  14.4: BasePlusCommissionEmployee4.java
2  //BasePlusCommissionEmployee4 class inherits from CommissionEmployee3 and
3  // accesses CommissionEmployee3's private data via CommissionEmployee3's
4  // public methods.
5
6  public class BasePlusCommissionEmployee4 extends CommissionEmployee3
7  {
8     private double baseSalary; // base salary per week
9
10    // six-argument constructor
11    public BasePlusCommissionEmployee4( String first, String last,
12       String ssn, double sales, double rate, double salary )
13    {
14       super( first, last, ssn, sales, rate );
15       setBaseSalary( salary ); // validate and store base salary
16    } // end six-argument BasePlusCommissionEmployee4 constructor
17
18    // set base salary
19    public void setBaseSalary( double salary )
20    {
21       baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
22    } // end method setBaseSalary
23
24    // return base salary
25    public double getBaseSalary()
26    {
27       return baseSalary;
28    } // end method getBaseSalary
29
30    // calculate earnings
31    public double earnings()
32    {
33       return getBaseSalary() + super.earnings();
34    } // end method earnings
35
36    // return String representation of BasePlusCommissionEmployee4
37    public String toString()
```

```
38     {
39        return String.format( "%s %s\n%s: %.2f", "base-salaried",
40           super.toString(), "base salary", getBaseSalary() );
41     } // end method toString
42  } // end class BasePlusCommissionEmployee4
```

Class `BasePlusCommissionEmployee4` (Fig. 14.3) has several changes to its method implementations that distinguish it from class `BasePlusCommissionEmployee3` (Fig. 14.3). Methods `earnings` (Fig. 14.2, lines 31-34) and `toString` (lines 37-41) each invoke method `getBaseSalary` to obtain the base salary value, rather than accessing `baseSalary` directly. If we decide to rename instance variable `baseSalary`, only the bodies of method `setBaseSalary` and `getBaseSalary` will need to change.

Class `BasePlusCommissionEmployee4`'s `earnings` method (Fig. 14.2, lines 31-34) overrides class `CommissionEmployee3`'s `earnings` method (Fig.14.4 lines 85-88) to calculate the earnings of a base-salaried commission employee. The new version obtains the portion of the employee's earnings based on commission alone by calling `CommissionEmployee3`'s `earnings` method with the expression `super.earnings()` (Fig. 14.3, line 33). `BasePlusCommissionEmployee4`'s `earnings` method then adds the base salary to this value to calculate the total earnings of the employee. Note the syntax used to invoke an overridden superclass method from a subclassplace the keyword `super` and a dot (`.`) separator before the superclass method name. This method invocation is a good software-engineering practice: Recall from Software Engineering Observation 8.5 that if a method performs all or some of the actions needed by another method, call that method rather than duplicate its code. By having `BasePlusCommissionEmployee4`'s `earnings` method invoke `CommissionEmployee3`'s `earnings` method to calculate part of a `BasePlusCommissionEmployee4` object's earnings, we avoid duplicating the code and reduce code-maintenance problems.

WEEK 15

General Learning Objectives for Week15: Polymorphism

Specific Objectives:

a. Understand the concepts of polymorphism using class hierarchy
b. Know how to create abstract classes
c. Write abstract methods
d. Write simple programs implementing polymorphism

Polymorphism

We now continue our study of object-oriented programming by explaining and demonstrating **polymorphism** with inheritance hierarchies. Polymorphism enables us to "program in the general" rather than "program in the specific." In particular, polymorphism enables us to write programs that process objects that share the same superclass in a class hierarchy as if they are all objects of the superclass.

Consider the following example of polymorphism. Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes `Fish`, `Frog` and `Bird` represent the three types of animals under investigation. Imagine that each of these classes extends superclass `Animal`, which contains a method `move` and maintains an animal's current location as x-y coordinates. Each subclass implements method `move`. Our program maintains an array of references to objects of the various `Animal` subclasses. To simulate the animals' movements, the program sends each object the same message once per secondnamely, `move`. However, each specific type of `Animal` responds to a `move` message in a unique waya `Fish` might swim three feet, a `Frog` might jump five feet and a `Bird` might fly ten feet. The program issues the same message (i.e., `move`) to each animal object generically, but each object knows how to modify its x-y coordinates appropriately for its specific type of movement. Relying on each object to know how to "do the right thing" (i.e., do what is appropriate for that type of object) in response to the same method call is the key concept of polymorphism. The same message (in this case, `move`) sent to a variety of objects has "many forms" of resultshence the term polymorphism.

With polymorphism, we can design and implement systems that are easily extensiblenew classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically. The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that the programmer adds to the hierarchy. For example, if we extend class `Animal` to create class `Tortoise` (which might respond to a `move` message by crawling one inch), we need to write only the `Tortoise` class and the part of the simulation that

instantiates a `Tortoise` object. The portions of the simulation that process each `Animal` generically can remain the same.

This chapter has several key parts. First, we discuss common examples of polymorphism. We then provide a live-code example demonstrating polymorphic behavior. As you will soon see, you will use superclass references to manipulate both superclass objects and subclass objects polymorphically.

We then present a case study that revisits the employee hierarchy of Section 9.4.5. We develop a simple payroll application that polymorphically calculates the weekly pay of several different types of employees using each employee's `earnings` method. Though the earnings of each type of employee are calculated in a specific way, polymorphism allows us to process the employees "in the general." In the case study, we enlarge the hierarchy to include two new classes`SalariedEmployee` (for people paid a fixed weekly salary) and `HourlyEmployee` (for people paid an hourly salary and so-called time-and-a-half for overtime). We declare a common set of functionality for all the classes in the updated hierarchy in a so-called abstract class, `Employee`, from which classes `SalariedEmployee`, `HourlyEmployee` and `CommissionEmployee` inherit directly and class `BasePlusCommissionEmployee4` inherits indirectly. As you will soon see, when we invoke each employee's `earnings` method off a superclass `Employee` reference, the correct earnings calculation is performed due to Java's polymorphic capabilities.

Occasionally, when performing polymorphic processing, we need to program "in the specific." Our `Employee` case study demonstrates that a program can determine the type of an object at execution time and act on that object accordingly. In the case study, we use these capabilities to determine whether a particular employee object is a `BasePlusCommissionEmployee`. If so, we increase that employee's base salary by 10%.

The chapter continues with an introduction to Java interfaces. An interface describes a set of methods that can be called on an object, but does not provide concrete implementations for the methods. Programmers can declare classes that implement (i.e., provide concrete implementations for the methods of) one or more interfaces. Each interface method must be declared in all the classes that implement the interface. Once a class implements an interface, all

objects of that class have an is-a relationship with the interface type, and all objects of the class are guaranteed to provide the functionality described by the interface. This is true of all subclasses of that class as well.

Interfaces are particularly useful for assigning common functionality to possibly unrelated classes. This allows objects of unrelated classes to be processed polymorphicallyobjects of classes that implement the same interface can respond to the same method calls. To demonstrate creating and using interfaces, we modify our payroll application to create a general accounts payable application that can calculate payments due for company employees and invoice amounts to be billed for purchased goods. As you will see, interfaces enable polymorphic capabilities similar to those possible with inheritance.

Polymorphism Examples

We now consider several additional examples of polymorphism. If class `Rectangle` is derived from class `Quadrilateral`, then a `Rectangle` object is a more specific version of a `Quadrilateral` object. Any operation (e.g., calculating the perimeter or the area) that can be performed on a `Quadrilateral` object can also be performed on a `Rectangle` object. These operations can also be performed on other `Quadrilaterals`, such as `Squares`, `Parallelograms` and `trapezoids`. The polymorphism occurs when a program invokes a method through a superclass variableat execution time, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable.

As another example, suppose we design a video game that manipulates objects of many different types, including objects of classes `Martian`, `Venusian`, `Plutonian`, `SpaceShip` and `LaserBeam`. Imagine that each class inherits from the common superclass called `SpaceObject`, which contains method `draw`. Each subclass implements this method. A screen-manager program maintains a collection (e.g., a `SpaceObject` array) of references to objects of the various classes. To refresh the screen, the screen manager periodically sends each object the same messagenamely, `draw`. However, each object responds in a unique way. For example, a `Martian` object might draw itself in red with the appropriate number of antennae. A `SpaceShip` object might draw itself as a bright silver flying saucer. A `LaserBeam` object might draw itself as a

bright red beam across the screen. Again, the same message (in this case, `draw`) sent to a variety of objects has "many forms" of results.

A polymorphic screen manager might use polymorphism to facilitate adding new classes to a system with minimal modifications to the system's code. Suppose that we want to add `Mercurian` objects to our video game. To do so, we must build a class `Mercurian` that extends `SpaceObject` and provides its own `draw` method implementation. When objects of class `Mercurian` appear in the `SpaceObject` collection, the screen manager code invokes method `draw`, exactly as it does for every other object in the collection, regardless of its type. So the new `Mercurian` objects simply "plug right in" without any modification of the screen manager code by the programmer. Thus, without modifying the system (other than to build new classes and modify the code that creates new objects), programmers can use polymorphism to include additional types that were not envisioned when the system was created.

With polymorphism, the same method name and signature can be used to cause different actions to occur, depending on the type of object on which the method is invoked. This gives the programmer tremendous expressive capability.

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

Demonstrating Polymorphic Behavior

In week fourteen created a commission employee class hierarchy, in which class `BasePlusCommissionEmployee` inherited from class `CommissionEmployee`. The examples in that section manipulated `CommissionEmployee` and `BasePlusCommissionEmployee` objects by using references to them to invoke their methods. We aimed superclass references at superclass objects and subclass references at subclass objects. These assignments are natural and straightforwardsuperclass references are intended to refer to superclass objects, and subclass references are intended to refer to subclass objects. However, as you will soon see, other assignments are possible.

In the next example, we aim a superclass reference at a subclass object. We then show how invoking a method on a subclass object via a superclass reference invokes the subclass functionalitythe type of the actual referenced object, not the type of the reference, determines which method is called. This example demonstrates the key concept that an object of a subclass can be treated as an object of its superclass. This enables various interesting manipulations. A program can create an array of superclass references that refer to objects of many subclass types. This is allowed because each subclass object is an object of its superclass. For instance, we can assign the reference of a `BasePlusCommissionEmployee` object to a superclass `CommissionEmployee` variable because a `BasePlusCommissionEmployee` is a `CommissionEmployeewe` can treat a `BasePlusCommissionEmployee` as a `CommissionEmployee`.

As you will learn later in the chapter, we cannot treat a superclass object as a subclass object because a superclass object is not an object of any of its subclasses. For example, we cannot assign the reference of a `CommissionEmployee` object to a subclass `BasePlusCommissionEmployee` variable because a `CommissionEmployee` is not a `BasePlusCommissionEmployeea CommissionEmployee` does not have a `baseSalary` instance variable and does not have methods `setBaseSalary` and `getBaseSalary`. The is-a relationship applies only from a subclass to its direct (and indirect) superclasses, and not vice versa.

It turns out that the Java compiler does allow the assignment of a superclass reference to a subclass variable if we explicitly cast the superclass reference to the subclass type. Why would we ever want to perform such an assignment? A superclass reference can be used to invoke only the methods declared in the superclass attempting to invoke subclass-only methods through a superclass reference results in compilation errors. If a program needs to perform a subclass-specific operation on a subclass object referenced by a superclass variable, the program must first cast the superclass reference to a subclass reference through a technique known as **downcasting**. This enables the program to invoke subclass methods that are not in the superclass. We will show you a concrete example of downcasting later in the text.

```
1  // Fig. 15.1: PolymorphismTest.java
2  // Assigning superclass and subclass references to superclass and
3  // subclass variables.
```

```
 4
 5  public class PolymorphismTest
 6  {
 7     public static void main( String args[] )
 8     {
 9        // assign superclass reference to superclass variable
10        CommissionEmployee3 commissionEmployee = new CommissionEmployee3(
11           "Sue", "Jones", "222-22-2222", 10000, .06 );
12
13        // assign subclass reference to subclass variable
14        BasePlusCommissionEmployee4 basePlusCommissionEmployee =
15           new BasePlusCommissionEmployee4(
16           "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
17
18        // invoke toString on superclass object using superclass variable
19        System.out.printf( "%s %s:\n\n%s\n\n",
20           "Call CommissionEmployee3's toString with superclass reference
",
21           "to superclass object", commissionEmployee.toString() );
22
23        // invoke toString on subclass object using subclass variable
24        System.out.printf( "%s %s:\n\n%s\n\n",
25           "Call BasePlusCommissionEmployee4's toString with subclass",
26           "reference to subclass object",
27           basePlusCommissionEmployee.toString() );
28
29        // invoke toString on subclass object using superclass variable
30        CommissionEmployee3 commissionEmployee2 =
31           basePlusCommissionEmployee;
32        System.out.printf( "%s %s:\n\n%s\n",
33           "Call BasePlusCommissionEmployee4's toString with superclass",
34           "reference to subclass object", commissionEmployee2.toString()
);
35     } // end main
36  } // end class PolymorphismTest
```

```
Call CommissionEmployee3's toString with superclass reference to superclass
object:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Call BasePlusCommissionEmployee4's toString with subclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Call BasePlusCommissionEmployee4's toString with superclass reference to
subclass object:
```

```
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

In Fig. 15.1, lines 10-11 create a `CommissionEmployee3` object and assign its reference to a `CommissionEmployee3` variable. Lines 14-16 create a `BasePlusCommissionEmployee4` object and assign its reference to a `BasePlusCommissionEmployee4` variable. These assignments are naturalfor example, a `CommissionEmployee3` variable's primary purpose is to hold a reference to a `CommissionEmployee3` object. Lines 19-21 use reference `commissionEmployee` to invoke `toString` explicitly. Because `commissionEmployee` refers to a `CommissionEmployee3` object, superclass `CommissionEmployee3`'s version of `toString` is called. Similarly, lines 24-27 use `basePlusCommissionEmployee` to invoke `toString` explicitly on the `BasePlusCommissionEmployee4` object. This invokes subclass `BasePlusCommissionEmployee4`'s version of `toString`.

Lines 30-31 then assign the reference to subclass object `basePlusCommissionEmployee` to a superclass `CommissionEmployee3` variable, which lines 32-34 use to invoke method `toString`. A superclass variable that contains a reference to a subclass object and is used to call a method actually calls the subclass version of the method. Hence, `commissionEmployee2.toString()` in line 34 actually calls class `BasePlusCommissionEmployee4`'s `toString` method. The Java compiler allows this "crossover" because an object of a subclass is an object of its superclass (but not vice versa). When the compiler encounters a method call made through a variable, the compiler determines if the method can be called by checking the variable's class type. If that class contains the proper method declaration (or inherits one), the compiler allows the call to be compiled. At execution time, the type of the object to which the variable refers determines the actual method to use.

Abstract Classes and Methods

When we think of a class type, we assume that programs will create objects of that type. In some cases, however, it is useful to declare classes for which the programmer never intends to

instantiate objects. Such classes are called abstract classes. Because they are used only as superclasses in inheritance hierarchies, we refer to them as **abstract superclasses**. These classes cannot be used to instantiate objects, because, as we will soon see, abstract classes are incomplete. Subclasses must declare the "missing pieces." We demonstrate abstract classes in later.

The purpose of an abstract class is primarily to provide an appropriate superclass from which other classes can inherit and thus share a common design. In the `Shape` hierarchy for example, subclasses inherit the notion of what it means to be a `Shape`common attributes such as `location`, `color` and `borderThickness`, and behaviors such as `draw`, `move`, `resize` and `changeColor`. Classes that can be used to instantiate objects are called **concrete classes**. Such classes provide implementations of every method they declare (some of the implementations can be inherited). For example, we could derive concrete classes `Circle`, `Square` and `triangle` from abstract superclass `TwoDimensionalShape`. Similarly, we could derive concrete classes `Sphere`, `Cube` and `Tetrahedron` from abstract superclass `THReeDimensionalShape`. Abstract superclasses are too general to create real objectsthey specify only what is common among subclasses. We need to be more specific before we can create objects. For example, if you send the `draw` message to abstract class `TwoDimensionalShape`, it knows that two-dimensional shapes should be drawable, but it does not know what specific shape to draw, so it cannot implement a real `draw` method. Concrete classes provide the specifics that make it reasonable to instantiate objects.

Not all inheritance hierarchies contain abstract classes. However, programmers often write client code that uses only abstract superclass types to reduce client code's dependencies on a range of specific subclass types. For example, a programmer can write a method with a parameter of an abstract superclass type. When called, such a method can be passed an object of any concrete class that directly or indirectly extends the superclass specified as the parameter's type.

Abstract classes sometimes constitute several levels of the hierarchy. For example, the `Shape` hierarchy of begins with abstract class `Shape`. On the next level of the hierarchy are two more abstract classes, `TwoDimensionalShape` and `ThreeDimensionalShape`. The next level of the hierarchy declares concrete classes for `TwoDimensionalShapes` (`Circle`, `Square` and `TRiangle`) and for `ThreeDimensionalShapes` (`Sphere`, `Cube` and `TeTRahedron`).

You 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 do not provide implementations. A class that contains any abstract methods must be declared as an abstract class even if that class contains concrete (non-abstract) methods. Each concrete subclass of an abstract superclass also must provide concrete implementations 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. Similarly, subclasses cannot override `static` methods, so an `abstract static` method could never be implemented.

Although we cannot instantiate objects of abstract superclasses, you will soon see that we can use abstract superclasses to declare variables that can hold references to objects of any concrete class derived from those abstract classes. Programs typically use such variables to manipulate subclass objects polymorphically. We also can use abstract superclass names to invoke `static` methods declared in those abstract superclasses.

Consider another application of polymorphism. A drawing program needs to display many shapes, including new shape types that the programmer will add to the system after writing the drawing program. The drawing program might need to display shapes, such as `Circles`, `TRiangles`, `Rectangles` or others, that derive from abstract superclass `Shape`. The drawing program uses `Shape` variables to manage the objects that are displayed. To draw any object in this inheritance hierarchy, the drawing program uses a superclass `Shape` variable containing a reference to the subclass object to invoke the object's `draw` method. This method is declared `abstract` in superclass `Shape`, so each concrete subclass must implement method `draw` in a manner specific to that shape. Each object in the `Shape` inheritance hierarchy knows how to draw itself. The drawing program does not have to worry about the type of each object or whether the drawing program has ever encountered objects of that type.

Polymorphism is particularly effective for implementing so-called layered software systems. In operating systems, for example, each type of physical device could operate quite differently from the others. Even so, commands to read or write data from and to devices may have a certain uniformity. For each device, the operating system uses a piece of software called a device driver to control all communication between the system and the device. The write message sent to a device-driver object needs to be interpreted specifically in the context of that driver and how it manipulates devices of a specific type. However, the write call itself really is no different from the write to any other device in the system: Place some number of bytes from memory onto that device. An object-oriented operating system might use an abstract superclass to provide an "interface" appropriate for all device drivers. Then, through inheritance from that abstract superclass, subclasses are formed that all behave similarly. The device driver methods are declared as abstract methods in the abstract superclass. The implementations of these abstract methods are provided in the subclasses that correspond to the specific types of device drivers. New devices are always being developed, and often long after the operating system has been released. When you buy a new device, it comes with a device driver provided by the device vendor. The device is immediately operational after you connect it to your computer and install the driver. This is another elegant example of how polymorphism makes systems extensible.

Creating Abstract Superclass `Employee`

Class `Employee` (Fig. 15.4) provides methods `earnings` and `toString`, in addition to the get and set methods that manipulate `Employee`'s instance variables. An `earnings` method certainly applies generically to all employees. But each earnings calculation depends on the employee's class. So we declare `earnings` as `abstract` in superclass `Employee` because a default implementation does not make sense for that methodthere is not enough information to determine what amount `earnings` should return. Each subclass overrides `earnings` with an appropriate implementation. To calculate an employee's earnings, the program assigns a reference to the employee's object to a superclass `Employee` variable, then invokes the `earnings` method on that variable. We maintain an array of `Employee` variables, each of which holds a reference to an `Employee` object (of course, there cannot be `Employee` objects because `Employee` is an abstract classbecause of inheritance, however, all objects of all subclasses of `Employee` may nevertheless be thought of as `Employee` objects). The program iterates through the array and

calls method `earnings` for each `Employee` object. Java processes these method calls polymorphically. Including `earnings` as an abstract method in `Employee` forces every direct subclass of `Employee` to override `earnings` in order to become a concrete class. This enables the designer of the class hierarchy to demand that each subclass provide an appropriate pay calculation.

Method `toString` in class `Employee` returns a `String` containing the first name, last name and social security number of the employee. As we will see, each subclass of `Employee` overrides method `toString` to create a string representation of an object of that class that contains the employee's type (e.g., `"salaried employee:"`) followed by the rest of the employee's information.

The diagram in Fig. 15.3 shows each of the five classes in the hierarchy down the left side and methods `earnings` and `toString` across the top. For each class, the diagram shows the desired results of each method. [Note: We do not list superclass `Employee`'s get and set methods because they are not overridden in any of the subclasseseach of these methods is inherited and used "as is" by each of the subclasses.]

```
 1  // Fig. 15.4: Employee.java
 2  // Employee abstract superclass.
 3
 4  public abstract class Employee
 5  {
 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;
22     } // end method setFirstName
23
24     // return first name
25     public String getFirstName()
```

```
26     {
27        return firstName;
28     } // end method getFirstName
29
30     // set last name
31     public void setLastName( String last )
32     {
33        lastName = last;
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     public String toString()
56     {
57        return String.format( "%s %s\nsocial security number: %s",
58           getFirstName(), getLastName(), getSocialSecurityNumber() );
59     } // end method toString
60
61     // abstract method overridden by subclasses
62     public abstract double earnings(); // no implementation here
63  } // end abstract class Employee
```

Why did we decide to declare `earnings` as an `abstract` method? It simply does not make sense to provide an implementation of this method in class `Employee`. We cannot calculate the earnings for a general `Employee`we first must know the specific `Employee` type to determine the appropriate earnings calculation. By declaring this method `abstract`, we indicate that each concrete subclass must provide an appropriate `earnings` implementation and that a program will be able to use superclass `Employee` variables to invoke method `earnings` polymorphically for any type of `Employee`.

Creating Concrete Subclass `SalariedEmployee`

Class `SalariedEmployee` (Fig. 15.5) extends class `Employee` (line 4) and overrides `earnings` (lines 29-32), which makes `SalariedEmployee` a concrete class. The class includes a constructor (lines 9-14) that takes a first name, a last name, a social security number and a weekly salary as arguments; a set method to assign a new non-negative value to instance variable `weeklySalary` (lines 17-20); a get method to return `weeklySalary`'s value (lines 23-26); a method `earnings` (lines 29-32) to calculate a `SalariedEmployee`'s earnings; and a method `toString` (lines 35-39), which returns a `String` including the employee's type, namely, `"salaried employee:"` followed by employee-specific information produced by superclass `Employee`'s `toString` method and `SalariedEmployee`'s `getWeeklySalary` method. Class `SalariedEmployee`'s constructor passes the first name, last name and social security number to the `Employee` constructor (line 12) to initialize the `private` instance variables not inherited from the superclass. Method `earnings` overrides abstract method `earnings` in `Employee` to provide a concrete implementation that returns the `SalariedEmployee`'s weekly salary. If we do not implement `earnings`, class `SalariedEmployee` must be declared `abstract`otherwise, a compilation error occurs (and, of course, we want `SalariedEmployee` here to be a concrete class).

```
 1  // Fig. 10.5: SalariedEmployee.java
 2  // SalariedEmployee class extends Employee.
 3
 4  public class SalariedEmployee extends Employee
 5  {
 6     private double weeklySalary;
 7
 8     // 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        weeklySalary = salary < 0.0 ? 0.0 : salary;
20     } // end method setWeeklySalary
21
22     // return salary
23     public double getWeeklySalary()
```

```
24     {
25        return weeklySalary;
26     } // end method getWeeklySalary
27
28     // calculate earnings; override abstract method earnings in Employee
29     public double earnings()
30     {
31        return getWeeklySalary();
32     } // end method earnings
33
34     // return String representation of SalariedEmployee object
35     public String toString()
36     {
37        return String.format( "salaried employee: %s\n%s: $%,.2f",
38           super.toString(), "weekly salary", getWeeklySalary() );
39     } // end method toString
40  } // end class SalariedEmployee
```

Method `toString` (lines 35-39) of class `SalariedEmployee` overrides `Employee` method `toString`. If class `SalariedEmployee` did not override `toString`, `SalariedEmployee` would have inherited the `Employee` version of `toString`. In that case, `SalariedEmployee`'s `toString` method would simply return the employee's full name and social security number, which does not adequately represent a `SalariedEmployee`. To produce a complete string representation of a `SalariedEmployee`, the subclass's `toString` method returns `"salaried employee:"` followed by the superclass `Employee`-specific information (i.e., first name, last name and social security number) obtained by invoking the superclass's `toString` (line 38)this is a nice example of code reuse. The string representation of a `SalariedEmployee` also contains the employee's weekly salary obtained by invoking the class's `getWeeklySalary` method.