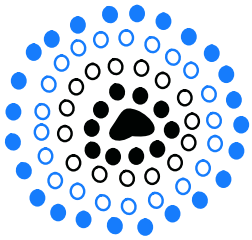
Chapter 1 Java Primer



Contents

1.1 Getting Started

Building data structures and algorithms requires that we communicate detailed instructions to a computer. An excellent way to perform such communication is using a high-level computer language, such as Java. In this chapter, we provide an overview of the Java programming language, and we continue this discussion in the next chapter, focusing on object-oriented design principles. We assume that readers are somewhat familiar with an existing high-level language, although not necessarily Java. This book does not provide a complete description of the Java language (there are numerous language references for that purpose), but it does introduce all aspects of the language that are used in code fragments later in this book.

We begin our Java primer with a program that prints "Hello Universe!" on the screen, which is shown in a dissected form in Figure 1.1.

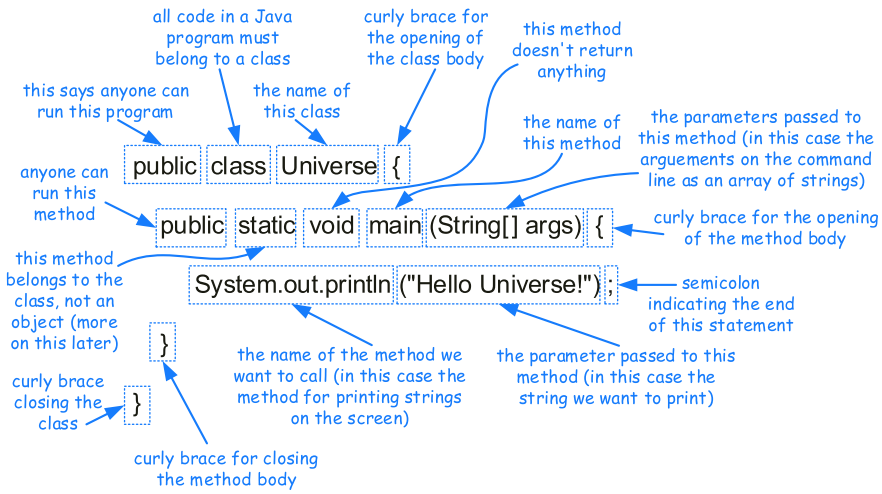


Figure 1.1: A "Hello Universe!" program.

In Java, executable statements are placed in functions, known as ***methods***, that belong to ***class*** definitions. The Universe class, in our first example, is extremely simple; its only method is a static one named main, which is the first method to be executed when running a Java program. Any set of statements between the braces "{" and "}" define a program ***block***. Notice that the entire Universe class definition is delimited by such braces, as is the body of the main method.

The name of a class, method, or variable in Java is called an ***identifier***, which can be any string of characters as long as it begins with a letter and consists of letters, numbers, and underscore characters (where "letter" and "number" can be from any written language defined in the Unicode character set). We list the exceptions to this general rule for Java identifiers in Table 1.1.

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

Table 1.1: A listing of the reserved words in Java. These names cannot be used as class, method, or variable names.

Comments

In addition to executable statements and declarations, Java allows a programmer to embed comments, which are annotations provided for human readers that are not processed by the Java compiler. Java allows two kinds of comments: inline comments and block comments. Java uses a "//" to begin an inline comment, ignoring everything subsequently on that line. For example:

```
// This is an inline comment.
```

We will intentionally color all comments in blue in this book, so that they are not confused with executable code.

While inline comments are limited to one line, Java allows multiline comments in the form of block comments. Java uses a "/*" to begin a block comment and a "*/" to close it. For example:

```
/*
 * This is a block comment.
 */
```

Block comments that begin with "/**" (note the second asterisk) have a special purpose, allowing a program, called Javadoc, to read these comments and automatically generate software documentation. We discuss the syntax and interpretation of Javadoc comments in Section 1.9.4.

1.1.1 Base Types

For the most commonly used data types, Java provides the following ***base types*** (also called ***primitive types***):

boolean	a boolean value: true or false
char	16-bit Unicode character
byte	8-bit signed two's complement integer
short	16-bit signed two's complement integer
int	32-bit signed two's complement integer
long	64-bit signed two's complement integer
float	32-bit floating-point number (IEEE 754-1985)
double	64-bit floating-point number (IEEE 754-1985)

A variable having one of these types simply stores a value of that type. Integer constants, like 14 or 195, are of type **int**, unless followed immediately by an 'L' or 'l', in which case they are of type **long**. Floating-point constants, like 3.1416 or 6.022e23, are of type **double**, unless followed immediately by an 'F' or 'f', in which case they are of type **float**. Code Fragment 1.1 demonstrates the declaration, and initialization in some cases, of various base-type variables.

```
1  boolean flag = true;
2  boolean verbose, debug;                  // two variables declared, but not yet initialized
3  char grade = 'A';
4  byte b = 12;
5  short s = 24;
6  int i, j, k = 257;                       // three variables declared; only k initialized
7  long l = 890L;                           // note the use of "L" here
8  float pi = 3.1416F;                      // note the use of "F" here
9  double e = 2.71828, a = 6.022e23;        // both variables are initialized
```

Code Fragment 1.1: Declarations and initializations of several base-type variables.

Note that it is possible to declare (and initialize) multiple variables of the same type in a single statement, as done on lines 2, 6, and 9 of this example. In this code fragment, variables verbose, debug, i, and j remain uninitialized. Variables declared locally within a block of code must be initialized before they are first used.

A nice feature of Java is that when base-type variables are declared as instance variables of a class (see next section), Java ensures initial default values if not explicitly initialized. In particular, all numeric types are initialized to zero, a boolean is initialized to false, and a character is initialized to the null character by default.

1.2 Classes and Objects

In more complex Java programs, the primary "actors" are ***objects***. Every object is an ***instance*** of a class, which serves as the ***type*** of the object and as a blueprint, defining the data which the object stores and the methods for accessing and modifying that data. The critical ***members*** of a class in Java are the following:

- ***Instance variables***, which are also called ***fields***, represent the data associated with an object of a class. Instance variables must have a ***type***, which can either be a base type (such as **int**, **float**, or **double**) or any class type (also known as a ***reference type*** for reasons we soon explain).
- ***Methods*** in Java are blocks of code that can be called to perform actions (similar to functions and procedures in other high-level languages). Methods can accept parameters as arguments, and their behavior may depend on the object upon which they are invoked and the values of any parameters that are passed. A method that returns information to the caller without changing any instance variables is known as an ***accessor method***, while an ***update method*** is one that may change one or more instance variables when called.

For the purpose of illustration, Code Fragment 1.2 provides a complete definition of a very simple class named Counter, to which we will refer during the remainder of this section.

```
1  public class Counter {
2    private int count;                                  // a simple integer instance variable
3    public Counter( ) { }                               // default constructor (count is 0)
4    public Counter(int initial) { count = initial; }    // an alternate constructor
5    public int getCount( ) { return count; }            // an accessor method
6    public void increment( ) { count++; }               // an update method
7    public void increment(int delta) { count += delta; }  // an update method
8    public void reset( ) { count = 0; }                 // an update method
9  }
```

Code Fragment 1.2: A Counter class for a simple counter, which can be queried, incremented, and reset.

This class includes one instance variable, named count, which is declared at line 2. As noted on the previous page, the count will have a default value of zero, unless we otherwise initialize it.

The class includes two special methods known as constructors (lines 3 and 4), one accessor method (line 5), and three update methods (lines 6–8). Unlike the original Universe class from page 2, our Counter class does not have a main method, and so it cannot be run as a complete program. Instead, the purpose of the Counter class is to create instances that might be used as part of a larger program.

1.2.1 Creating and Using Objects

Before we explore the intricacies of the syntax for our Counter class definition, we prefer to describe how Counter instances can be created and used. To this end, Code Fragment 1.3 presents a new class named CounterDemo.

```
1  public class CounterDemo {
2    public static void main(String[ ] args) {
3      Counter c;                    // declares a variable; no counter yet constructed
4      c = new Counter( );           // constructs a counter; assigns its reference to c
5      c.increment( );               // increases its value by one
6      c.increment(3);               // increases its value by three more
7      int temp = c.getCount( );     // will be 4
8      c.reset( );                   // value becomes 0
9      Counter d = new Counter(5);// declares and constructs a counter having value 5
10     d.increment( );               // value becomes 6
11     Counter e = d;                // assigns e to reference the same object as d
12     temp = e.getCount( );         // will be 6 (as e and d reference the same counter)
13     e.increment(2);               // value of e (also known as d) becomes 8
14   }
15 }
```

Code Fragment 1.3: A demonstration of the use of Counter instances.

There is an important distinction in Java between the treatment of base-type variables and class-type variables. At line 3 of our demonstration, a new variable c is declared with the syntax:

```
Counter c;
```

This establishes the identifier, c, as a variable of type Counter, but it does not create a Counter instance. Classes are known as ***reference types*** in Java, and a variable of that type (such as c in our example) is known as a ***reference variable***. A reference variable is capable of storing the location (i.e., ***memory address***) of an object from the declared class. So we might assign it to reference an existing instance or a newly constructed instance. A reference variable can also store a special value, **null**, that represents the lack of an object.

In Java, a new object is created by using the **new** operator followed by a call to a constructor for the desired class; a constructor is a method that always shares the same name as its class. The **new** operator returns a ***reference*** to the newly created instance; the returned reference is typically assigned to a variable for further use.

In Code Fragment 1.3, a new Counter is constructed at line 4, with its reference assigned to the variable c. That relies on a form of the constructor, Counter( ), that takes no arguments between the parentheses. (Such a zero-parameter constructor is known as a ***default constructor***.) At line 9 we construct another counter using a one-parameter form that allows us to specify a nonzero initial value for the counter.

Three events occur as part of the creation of a new instance of a class:

- A new object is dynamically allocated in memory, and all instance variables are initialized to standard default values. The default values are **null** for reference variables and 0 for all base types except **boolean** variables (which are **false** by default).
- The constructor for the new object is called with the parameters specified. The constructor may assign more meaningful values to any of the instance variables, and perform any additional computations that must be done due to the creation of this object.
- After the constructor returns, the **new** operator returns a reference (that is, a memory address) to the newly created object. If the expression is in the form of an assignment statement, then this address is stored in the object variable, so the object variable ***refers*** to this newly created object.

The Dot Operator

One of the primary uses of an object reference variable is to access the members of the class for this object, an instance of its class. That is, an object reference variable is useful for accessing the methods and instance variables associated with an object. This access is performed with the dot (".") operator. We call a method associated with an object by using the reference variable name, following that by the dot operator and then the method name and its parameters. For example, in Code Fragment 1.3, we call c.increment( ) at line 5, c.increment(3) at line 6, c.getCount( ) at line 7, and c.reset( ) at line 8. If the dot operator is used on a reference that is currently **null**, the Java runtime environment will throw a NullPointerException.

If there are several methods with this same name defined for a class, then the Java runtime system uses the one that matches the actual number of parameters sent as arguments, as well as their respective types. For example, our Counter class supports two methods named increment: a zero-parameter form and a one-parameter form. Java determines which version to call when evaluating commands such as c.increment( ) versus c.increment(3). A method's name combined with the number and types of its parameters is called a method's ***signature***, for it takes all of these parts to determine the actual method to perform for a certain method call. Note, however, that the signature of a method in Java does not include the type that the method returns, so Java does not allow two methods with the same signature to return different types.

A reference variable v can be viewed as a "pointer" to some object o. It is as if the variable is a holder for a remote control that can be used to control the newly created object (the device). That is, the variable has a way of pointing at the object and asking it to do things or give us access to its data. We illustrate this concept in Figure 1.2. Using the remote control analogy, a **null** reference is a remote control holder that is empty.

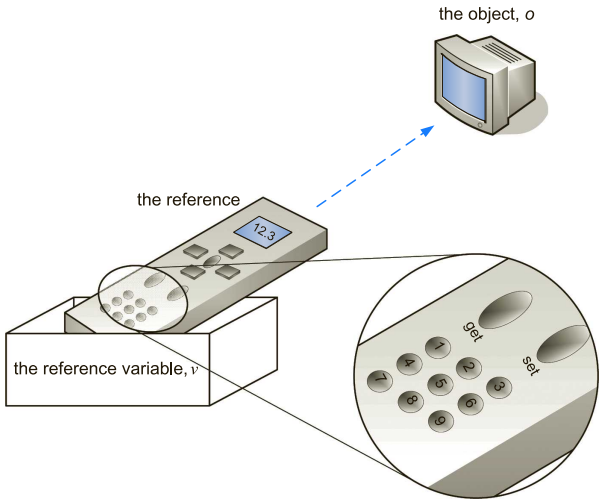


Figure 1.2: Illustrating the relationship between objects and object reference variables. When we assign an object reference (that is, memory address) to a reference variable, it is as if we are storing that object's remote control at that variable.

There can, in fact, be many references to the same object, and each reference to a specific object can be used to call methods on that object. Such a situation would correspond to our having many remote controls that all work on the same device. Any of the remotes can be used to make a change to the device (like changing a channel on a television). Note that if one remote control is used to change the device, then the (single) object pointed to by all the remotes changes. Likewise, if one object reference variable is used to change the state of the object, then its state changes for all the references to it. This behavior comes from the fact that there are many references, but they all point to the same object.

Returning to our CounterDemo example, the instance constructed at line 9 as

```
Counter d = new Counter(5);
```

is a distinct instance from the one identified as c. However, the command at line 11,

```
Counter e = d;
```

does not result in the construction of a new Counter instance. This declares a new *reference variable* named e, and assigns that variable a reference to the existing counter instance currently identified as d. At that point, both variables d and e are aliases for the same object, and so the call to d.getCount( ) behaves just as would e.getCount( ). Similarly, the call to update method e.increment(2) is affecting the same object identified by d.

It is worth noting, however, that the aliasing of two reference variables to the same object is not permanent. At any point in time, we may reassign a reference variable to a new instance, to a different existing instance, or to **null**.

1.2.2 Defining a Class

Thus far, we have provided definitions for two simple classes: the Universe class on page 2 and the Counter class on page 5. At its core, a class definition is a block of code, delimited by braces "{" and "}" , within which is included declarations of instance variables and methods that are the members of the class. In this section, we will undertake a deeper examination of class definitions in Java.

Modifiers

Immediately before the definition of a class, instance variable, or method in Java, keywords known as ***modifiers*** can be placed to convey additional stipulations about that definition.

Access Control Modifiers

The first set of modifiers we discuss are known as ***access control modifiers***, as they control the level of access (also known as ***visibility***) that the defining class grants to other classes in the context of a larger Java program. The ability to limit access among classes supports a key principle of object-orientation known as encapsulation (see Section 2.1). In general, the different access control modifiers and their meaning are as follows:

- The **public** class modifier designates that all classes may access the defined aspect. For example, line 1 of of Code Fragment 1.2 designates

  ```
  public class Counter {
  ```

  and therefore all other classes (such as CounterDemo) are allowed to construct new instances of the Counter class, as well as to declare variables and parameters of type Counter. In Java, each public class must be defined in a separate file named *classname*`.java`, where "*classname*" is the name of the class (for example, file `Counter.java` for the Counter class definition).

  The designation of **public** access for a particular *method* of a class allows any other class to make a call to that method. For example, line 5 of Code Fragment 1.2 designates

  ```
  public int getCount( ) { return count; }
  ```

  which is why the CounterDemo class may call c.getCount( ).

  If an instance variable is declared as public, dot notation can be used to directly access the variable by code in any other class that possesses a reference to an instance of this class. For example, were the count variable of Counter to be declared as public (which it is not), then the CounterDemo would be allowed to read or modify that variable using a syntax such as c.count.

- The **protected** class modifier designates that access to the defined aspect is only granted to the following groups of other classes:
  - Classes that are designated as ***subclasses*** of the given class through inheritance. (We will discuss inheritance as the focus of Section 2.2.)
  - Classes that belong to the same ***package*** as the given class. (We will discuss packages within Section 1.8.)
- The **private** class modifier designates that access to a defined member of a class be granted only to code within that class. Neither subclasses nor any other classes have access to such members.

  For example, we defined the count instance variable of the Counter class to have private access level. We were allowed to read or edit its value from within methods of that class (such as getCount, increment, and reset), but other classes such as CounterDemo cannot directly access that field. Of course, we did provide other public methods to grant outside classes with behaviors that depended on the current count value.
- Finally, we note that if no explicit access control modifier is given, the defined aspect has what is known as ***package-private*** access level. This allows other classes in the same package (see Section 1.8) to have access, but not any classes or subclasses from other packages.

The **static** Modifier

The **static** modifier in Java can be declared for any variable or method of a class (or for a nested class, as we will introduce in Section 2.6).

When a variable of a class is declared as **static**, its value is associated with the class as a whole, rather than with each individual instance of that class. Static variables are used to store "global" information about a class. (For example, a static variable could be used to maintain the total number of instances of that class that have been created.) Static variables exist even if no instance of their class exists.

When a method of a class is declared as **static**, it too is associated with the class itself, and not with a particular instance of the class. That means that the method is not invoked on a particular instance of the class using the traditional dot notation. Instead, it is typically invoked using the name of the class as a qualifier.

As an example, in the java.lang package, which is part of the standard Java distribution, there is a Math class that provides many static methods, including one named sqrt that computes square roots of numbers. To compute a square root, you do not need to create an instance of the Math class; that method is called using a syntax such as Math.sqrt(2), with the class name Math as the qualifier before the dot operator.

Static methods can be useful for providing utility behaviors related to a class that need not rely on the state of any particular instance of that class.

The **abstract** Modifier

A method of a class may be declared as **abstract**, in which case its signature is provided but without an implementation of the method body. Abstract methods are an advanced feature of object-oriented programming to be combined with inheritance, and the focus of Section 2.3.3. In short, any subclass of a class with abstract methods is expected to provide a concrete implementation for each abstract method.

A class with one or more abstract methods must also be formally declared as **abstract**, because it is essentially incomplete. (It is also permissible to declare a class as abstract even if it does not contain any abstract methods.) As a result, Java will not allow any instances of an abstract class to be constructed, although reference variables may be declared with an abstract type.

The **final** Modifier

A variable that is declared with the **final** modifier can be initialized as part of that declaration, but can never again be assigned a new value. If it is a base type, then it is a constant. If a reference variable is **final**, then it will always refer to the same object (even if that object changes its internal state). If a member variable of a class is declared as **final**, it will typically be declared as **static** as well, because it would be unnecessarily wasteful to have every instance store the identical value when that value can be shared by the entire class.

Designating a method or an entire class as **final** has a completely different consequence, only relevant in the context of inheritance. A final method cannot be overridden by a subclass, and a final class cannot even be subclassed.

Declaring Instance Variables

When defining a class, we can declare any number of instance variables. An important principle of object-orientation is that each instance of a class maintains its own individual set of instance variables (that is, in fact, why they are called *instance* variables). So in the case of the Counter class, each instance will store its own (independent) value of count.

The general syntax for declaring one or more instance variables of a class is as follows (with optional portions bracketed):

[*modifiers*] *type identifier*$_1$[=*initialValue*$_1$], *identifier*$_2$[=*initialValue*$_2$];

In the case of the Counter class, we declared

```
private int count;
```

where **private** is the modifier, **int** is the type, and count is the identifier. Because we did not declare an initial value, it automatically receives the default of zero as a base-type integer.

Declaring Methods

A method definition has two parts: the ***signature***, which defines the name and parameters for a method, and the ***body***, which defines what the method does. The method signature specifies how the method is called, and the method body specifies what the object will do when it is called. The syntax for defining a method is as follows:

[*modifiers*] *returnType methodName*(*type*$_1$ *param*$_1$, ..., *type*$_n$ *param*$_n$) {
    // method body . . .
}

Each of the pieces of this declaration has an important purpose. We have already discussed the significance of *modifiers* such as **public**, **private**, and **static**. The *returnType* designation defines the type of value returned by the method. The *methodName* can be any valid Java identifier. The list of parameters and their types declares the local variables that correspond to the values that are to be passed as arguments to this method. Each type declaration *type*$_i$ can be any Java type name and each *param*$_i$ can be any distinct Java identifier. This list of parameters and their types can be empty, which signifies that there are no values to be passed to this method when it is invoked. These parameter variables, as well as the instance variables of the class, can be used inside the body of the method. Likewise, other methods of this class can be called from inside the body of a method.

When a (nonstatic) method of a class is called, it is invoked on a specific instance of that class and can change the state of that object. For example, the following method of the Counter class increases the counter's value by the given amount.

```
public void increment(int delta) {
  count += delta;
}
```

Notice that the body of this method uses count, which is an instance variable, and delta, which is a parameter.

Return Types

A method definition must specify the type of value the method will return. If the method does not return a value (as with the increment method of the Counter class), then the keyword **void** must be used. To return a value in Java, the body of the method must use the **return** keyword, followed by a value of the appropriate return type. Here is an example of a method (from the Counter class) with a nonvoid return type:

```
public int getCount( ) {
  return count;
}
```

Java methods can return only one value. To return multiple values in Java, we should instead combine all the values we want to return in a ***compound object***, whose instance variables include all the values we want to return, and then return a reference to that compound object. In addition, we can change the internal state of an object that is passed to a method as another way of "returning" multiple results.

Parameters

A method's parameters are defined in a comma-separated list enclosed in parentheses after the name of the method. A parameter consists of two parts, the parameter type and the parameter name. If a method has no parameters, then only an empty pair of parentheses is used.

All parameters in Java are passed ***by value***, that is, any time we pass a parameter to a method, a copy of that parameter is made for use within the method body. So if we pass an **int** variable to a method, then that variable's integer value is copied. The method can change the copy but not the original. If we pass an object reference as a parameter to a method, then the reference is copied as well. Remember that we can have many different variables that all refer to the same object. Reassigning the internal reference variable inside a method will not change the reference that was passed in.

For the sake of demonstration, we will assume that the following two methods were added to an arbitrary class (such as CounterDemo).

```
public static void badReset(Counter c) {
  c = new Counter( );              // reassigns local name c to a new counter
}

public static void goodReset(Counter c) {
  c.reset( );                      // resets the counter sent by the caller
}
```

Now we will assume that variable strikes refers to an existing Counter instance in some context, and that it currently has a value of 3.

If we were to call badReset(strikes), this has *no* effect on the Counter known as strikes. The body of the badReset method reassigns the (local) parameter variable c to reference a newly created Counter instance; but this does not change the state of the existing counter that was sent by the caller (i.e., strikes).

In contrast, if we were to call goodReset(strikes), this does indeed reset the caller's counter back to a value of zero. That is because the variables c and strikes are both reference variables that refer to the same Counter instance. So when c.reset( ) is called, that is effectively the same as if strikes.reset( ) were called.

Defining Constructors

A ***constructor*** is a special kind of method that is used to initialize a newly created instance of the class so that it will be in a consistent and stable initial state. This is typically achieved by initializing each instance variable of the object (unless the default value will suffice), although a constructor can perform more complex computation. The general syntax for declaring a constructor in Java is as follows:

modifiers *name*(*type*$_0$ *parameter*$_0$, ..., *type*$_{n-1}$ *parameter*$_{n-1}$) {
    // constructor body . . .
}

Constructors are defined in a very similar way as other methods of a class, but there are a few important distinctions:

1. Constructors cannot be **static**, **abstract**, or **final**, so the only modifiers that are allowed are those that affect visibility (i.e., **public**, **protected**, **private**, or the default package-level visibility).
2. The name of the constructor must be identical to the name of the class it constructs. For example, when defining the Counter class, a constructor must be named Counter as well.
3. We don't specify a return type for a constructor (not even **void**). Nor does the body of a constructor explicitly return anything. When a user of a class creates an instance using a syntax such as

   ```
   Counter d = new Counter(5);
   ```

   the **new** operator is responsible for returning a reference to the new instance to the caller; the responsibility of the constructor method is only to initialize the state of the new instance.

A class can have many constructors, but each must have a different ***signature***, that is, each must be distinguished by the type and number of the parameters it takes. If no constructors are explicitly defined, Java provides an implicit ***default constructor*** for the class, having zero arguments and leaving all instance variables initialized to their default values. However, if a class defines one or more nondefault constructors, no default constructor will be provided.

As an example, our Counter class defines the following pair of constructors:

```
public Counter( ) { }
public Counter(int initial) { count = initial; }
```

The first of these has a trivial body, { }, as the goal for this default constructor is to create a counter with value zero, and that is already the default value of the integer instance variable, count. However, it is still important that we declared such an explicit constructor, because otherwise none would have been provided, given the existence of the nondefault constructor. In that scenario, a user would have been unable to use the syntax, **new** Counter( ).

The Keyword **this**

Within the body of a (nonstatic) method in Java, the keyword **this** is automatically defined as a reference to the instance upon which the method was invoked. That is, if a caller uses a syntax such as thing.foo(a, b, c), then within the body of method foo for that call, the keyword **this** refers to the object known as thing in the caller's context. There are three common reasons why this reference is needed from within a method body:

1. To store the reference in a variable, or send it as a parameter to another method that expects an instance of that type as an argument.

2. To differentiate between an instance variable and a local variable with the same name. If a local variable is declared in a method having the same name as an instance variable for the class, that name will refer to the local variable within that method body. (We say that the local variable *masks* the instance variable.) In this case, the instance variable can still be accessed by explicitly using the dot notation with **this** as the qualifier. For example, some programmers prefer to use the following style for a constructor, with a parameter having the same name as the underlying variable.

```
public Counter(int count) {
  this.count = count;   // set the instance variable equal to parameter
}
```

3. To allow one constructor body to invoke another constructor body. When one method of a class invokes another method of that same class on the current instance, that is typically done by using the (unqualified) name of the other method. But the syntax for calling a constructor is special. Java allows use of the keyword **this** to be used as a method within the body of one constructor, so as to invoke another constructor with a different signature.

   This is often useful because all of the initialization steps of one constructor can be reused with appropriate parameterization. As a trivial demonstration of the syntax, we could reimplement the zero-argument version of our Counter constructor to have it invoke the one-argument version sending 0 as an explicit parameter. This would be written as follows:

```
public Counter( ) {
  this(0);          // invoke one-parameter constructor with value zero
}
```

   We will provide a more meaningful demonstration of this technique in a later example of a CreditCard class in Section 1.7.

The main Method

Some Java classes, such as our Counter class, are meant to be used by other classes, but are not intended to serve as a self-standing program. The primary control for an application in Java must begin in some class with the execution of a special method named main. This method must be declared as follows:

```
public static void main(String[ ] args) {
  // main method body...
}
```

The args parameter is an array of String objects, that is, a collection of indexed strings, with the first string being args[0], the second being args[1], and so on. (We say more about strings and arrays in Section 1.3.) Those represent what are known as ***command-line arguments*** that are given by a user when the program is executed.

Java programs can be called from the command line using the `java` command (in a Windows, Linux, or Unix shell), followed by the name of the Java class whose main method we want to run, plus any optional arguments. For example, to execute the main method of a class named Aquarium, we could issue the following command:

```
java Aquarium
```

In this case, the Java runtime system looks for a compiled version of the Aquarium class, and then invokes the special main method in that class.

If we had defined the Aquarium program to take an optional argument that specifies the number of fish in the aquarium, then we might invoke the program by typing the following in a shell window:

```
java Aquarium 45
```

to specify that we want an aquarium with 45 fish in it. In this case, args[0] would refer to the string `"45"`. It would be up to the body of the main method to interpret that string as the desired number of fish.

Programmers who use an integrated development environment (IDE), such as Eclipse, can optionally specify command-line arguments when executing the program through the IDE.

Unit Testing

When defining a class, such as Counter, that is meant to be used by other classes rather than as a self-standing program, there is no need to define a main method. However, a nice feature of Java's design is that we could provide such a method as a way to test the functionality of that class in isolation, knowing that it would not be run unless we specifically invoke the `java` command on that isolated class. However, for more robust testing, frameworks such as JUnit are preferred.

1.3 Strings, Wrappers, Arrays, and Enum Types

The String Class

Java's **char** base type stores a value that represents a single text ***character***. In Java, the set of all possible characters, known as an ***alphabet***, is the Unicode international character set, a 16-bit character encoding that covers most used written languages. (Some programming languages use the smaller ASCII character set, which is a proper subset of the Unicode alphabet based on a 7-bit encoding.) The form for expressing a character literal in Java is using single quotes, such as `'G'`.

Because it is common to work with sequences of text characters in programs (e.g., for user interactions or data processing), Java provides support in the form of a **String** ***class***. A string instance represents a sequence of zero or more characters. The class provides extensive support for various text-processing tasks, and in Chapter 13 we will examine several of the underlying algorithms for text processing. For now, we will only highlight the most central aspects of the String class. Java uses double quotes to designate string literals. Therefore, we might declare and initialize a String instance as follows:

```
String title = "Data Structures & Algorithms in Java"
```

Character Indexing

Each character c within a string s can be referenced by using an ***index***, which is equal to the number of characters that come before c in s. By this convention, the first character is at index 0, and the last is at index $n-1$, where n is the length of the string. For example, the string title, defined above, has length 36. The character at index 2 is `'t'` (the third character), and the character at index 4 is `' '` (the space character). Java's String class supports a method length( ), which returns the length of a string instance, and a method charAt(k), which returns the character at index k.

Concatenation

The primary operation for combining strings is called ***concatenation***, which takes a string P and a string Q combines them into a new string, denoted $P+Q$, which consists of all the characters of P followed by all the characters of Q. In Java, the "+" operation performs concatenation when acting on two strings, as follows:

```
String term = "over" + "load";
```

This statement defines a variable named term that references a string with value `"overload"`. (We will discuss assignment statements and expressions such as that above in more detail later in this chapter.)

The StringBuilder Class

An important trait of Java's String class is that its instances are ***immutable***; once an instance is created and initialized, the value of that instance cannot be changed. This is an intentional design, as it allows for great efficiencies and optimizations within the Java Virtual Machine.

However, because String is a class in Java, it is a reference type. Therefore, variables of type String can be reassigned to another string instance (even if the current string instance cannot be changed), as in the following:

```
String greeting = "Hello";
greeting = "Ciao";                    // we changed our mind
```

It is also quite common in Java to use string concatenation to build a new string that is subsequently used to replace one of the operands of concatenation, as in:

```
greeting = greeting + '!';          // now it is "Ciao!"
```

However, it is important to remember that this operation does create a new string instance, copying all the characters of the existing string in the process. For long string (such as DNA sequences), this can be very time consuming. (In fact, we will experiment with the efficiency of string concatenation to begin Chapter 4.)

In order to support more efficient editing of character strings, Java provides a **StringBuilder *class***, which is effectively a ***mutable*** version of a string. This class combines some of the accessor methods of the String class, while supporting additional methods including the following (and more):

setCharAt(k,c): Change the character at index k to character c.

insert(k,s): Insert a copy of string s starting at index k of the sequence, shifting existing characters further back to make room.

append(s): Append string s to the end of the sequence.

reverse(): Reverse the current sequence.

toString(): Return a traditional String instance based on the current character sequence.

An error condition occurs, for both String and StringBuilder classes, if an index k is out of the bounds of the indices of the character sequence.

The StringBuilder class can be very useful, and it serves as an interesting case study for data structures and algorithms. We will further explore the empirical efficiency of the StringBuilder class in Section 4.1 and the theoretical underpinnings of its implementation in Section 7.2.4.

Wrapper Types

There are many data structures and algorithms in Java's libraries that are specifically designed so that they only work with object types (not primitives). To get around this obstacle, Java defines a ***wrapper class*** for each base type. An instance of each wrapper type stores a single value of the corresponding base type. In Table 1.2, we show the base types and their corresponding wrapper class, along with examples of how objects are created and accessed.

Base Type	*Class Name*	*Creation Example*	*Access Example*
boolean	Boolean	obj = new Boolean(true);	obj.booleanValue()
char	Character	obj = new Character('Z');	obj.charValue()
byte	Byte	obj = new Byte((byte) 34);	obj.byteValue()
short	Short	obj = new Short((short) 100);	obj.shortValue()
int	Integer	obj = new Integer(1045);	obj.intValue()
long	Long	obj = new Long(10849L);	obj.longValue()
float	Float	obj = new Float(3.934F);	obj.floatValue()
double	Double	obj = new Double(3.934);	obj.doubleValue()

Table 1.2: Java's wrapper classes. Each class is given with its corresponding base type and example expressions for creating and accessing such objects. For each row, we assume the variable obj is declared with the corresponding class name.

Automatic Boxing and Unboxing

Java provides additional support for implicitly converting between base types and their wrapper types through a process known as automatic boxing and unboxing.

In any context for which an Integer is expected (for example, as a parameter), an **int** value k can be expressed, in which case Java automatically ***boxes*** the **int**, with an implicit call to **new** Integer(k). In reverse, in any context for which an **int** is expected, an Integer value v can be given in which case Java automatically ***unboxes*** it with an implicit call to v.intValue( ). Similar conversions are made with the other base-type wrappers. Finally, all of the wrapper types provide support for converting back and forth between string literals. Code Fragment 1.4 demonstrates many such features.

```
1  int j = 8;
2  Integer a = new Integer(12);
3  int k = a;                          // implicit call to a.intValue()
4  int m = j + a;                      // a is automatically unboxed before the addition
5  a = 3 * m;                          // result is automatically boxed before assignment
6  Integer b = new Integer("-135");    // constructor accepts a String
7  int n = Integer.parseInt("2013");   // using static method of Integer class
```

Code Fragment 1.4: A demonstration of the use of the Integer wrapper class.

Arrays

A common programming task is to keep track of an ordered sequence of related values or objects. For example, we may want a video game to keep track of the top ten scores for that game. Rather than using ten different variables for this task, we would prefer to use a single name for the group and use index numbers to refer to the high scores in that group. Similarly, we may want a medical information system to keep track of the patients currently assigned to beds in a certain hospital. Again, we would rather not have to introduce 200 variables in our program just because the hospital has 200 beds.

In such cases, we can save programming effort by using an ***array***, which is a sequenced collection of variables all of the same type. Each variable, or ***cell***, in an array has an ***index***, which uniquely refers to the value stored in that cell. The cells of an array a are numbered 0, 1, 2, and so on. We illustrate an array of high scores for a video game in Figure 1.3.

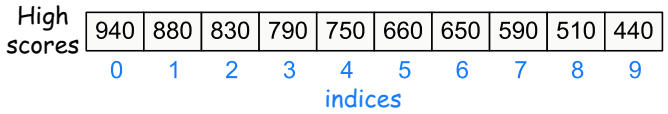


Figure 1.3: An illustration of an array of ten (**int**) high scores for a video game.

Array Elements and Capacities

Each value stored in an array is called an ***element*** of that array. Since the length of an array determines the maximum number of things that can be stored in the array, we will sometimes refer to the length of an array as its ***capacity***. In Java, the length of an array named a can be accessed using the syntax a.length. Thus, the cells of an array a are numbered 0, 1, 2, and so on, up through a.length-1, and the cell with index k can be accessed with syntax $a[k]$.

Out of Bounds Errors

It is a dangerous mistake to attempt to index into an array a using a number outside the range from 0 to a.length-1. Such a reference is said to be ***out of bounds***. Out of bounds references have been exploited numerous times by hackers using a method called the ***buffer overflow attack*** to compromise the security of computer systems written in languages other than Java. As a safety feature, array indices are always checked in Java to see if they are ever out of bounds. If an array index is out of bounds, the runtime Java environment signals an error condition. The name of this condition is the ArrayIndexOutOfBoundsException. This check helps Java avoid a number of security problems, such as buffer overflow attacks.

Declaring and Constructing Arrays

Arrays in Java are somewhat unusual, in that they are not technically a base type nor are they instances of a particular class. With that said, an instance of an array is treated as an object by Java, and variables of an array type are *reference variables*.

To declare a variable (or parameter) to have an array type, we use an empty pair of square brackets just after the type of element that the array will store. For example, we might declare:

```
int[ ] primes;
```

Because arrays are a reference type, this declares the variable primes to be a reference to an array of integer values, but it does not immediately construct any such array. There are two ways for creating an array.

The first way to create an array is to use an assignment to a literal form when initially declaring the array, using a syntax as:

$$elementType[\,]\ arrayName = \{initialValue_0, initialValue_1, \ldots, initialValue_{N-1}\};$$

The *elementType* can be any Java base type or class name, and *arrayName* can be any valid Java identifier. The initial values must be of the same type as the array. For example, we could initialize the array of primes to contain the first ten prime numbers as:

```
int[ ] primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
```

When using an initializer, an array is created having precisely the capacity needed to store the indicated values.

The second way to create an array is to use the **new** operator. However, because an array is not an instance of a class, we do not use a typical constructor syntax. Instead we use the syntax:

new *elementType*[*length*]

where *length* is a positive integer denoting the length of the new array. The **new** operator returns a reference to the new array, and typically this would be assigned to an array variable. For example, the following statement declares an array variable named measurements, and immediately assigns it a new array of 1000 cells.

```
double[ ] measurements = new double[1000];
```

When arrays are created using the **new** operator, all of their elements are automatically assigned the default value for the element type. That is, if the element type is numeric, all cells of the array are initialized to zero, if the element type is boolean, all cells are **false**, and if the element type is a reference type (such as with an array of String instances), all cells are initialized to **null**.

Enum Types

In olden times, programmers would often define a series of constant integer values to be used for representing a finite set of choices. For example, in representing a day of the week, they might declare variable today as an **int** and then set it with value 0 for Monday, 1 for Tuesday, and so on.

A slightly better programming style is to define static constants (with the **final** keyword), to make the associations, such as:

```
static final int MON = 0;
static final int TUE = 1;
static final int WED = 2;
...
```

because then it becomes possible to make assignments such as today = TUE, rather than the more obscure today = 1. Unfortunately, the variable today is still declared as an **int** using such a programming style, and it may not be clear that you intend for it to represent a day of the week when storing it as an instance variable or sending it as a parameter.

Java supports a more elegant approach to representing choices from a finite set by defining what is known as an enumerated type, or enum for short. These are types that are only allowed to take on values that come from a specified set of names. They are declared as follows:

modifier **enum** *name* { $valueName_0$, $valueName_1$, ..., $valueName_{n-1}$ };

where the *modifier* can be blank, **public**, **protected**, or **private**. The name of this enum, *name*, can be any legal Java identifier. Each of the value identifiers, $valueName_i$, is the name of a possible value that variables of this enum type can take on. Each of these name values can also be any legal Java identifier, but the Java convention is that these should usually be capitalized words. For example, an enumerated type definition for days of the weak might appear as:

```
public enum Day { MON, TUE, WED, THU, FRI, SAT, SUN };
```

Once defined, Day becomes an official type and we may declare variables or parameters with type Day. A variable of that type can be declared as:

```
Day today;
```

and an assignment of a value to that variable can appear as:

```
today = Day.TUE;
```

1.4 Expressions

Variables and constants are used in ***expressions*** to define new values and to modify variables. In this section, we discuss how expressions work in Java in more detail. Expressions involve the use of ***literals***, ***variables***, and ***operators***. Since we have already discussed variables, let us briefly focus on literals and then discuss operators in some detail.

1.4.1 Literals

A ***literal*** is any "constant" value that can be used in an assignment or other expression. Java allows the following kinds of literals:

- The **null** object reference (this is the only object literal, and it is allowed to be any reference type).
- Boolean: **true** and **false**.
- Integer: The default for an integer like 176, or -52 is that it is of type **int**, which is a 32-bit integer. A long integer literal must end with an "L" or "l", for example, 176L or -52l, and defines a 64-bit integer.
- Floating Point: The default for floating-point numbers, such as 3.1415 and 135.23, is that they are **double**. To specify that a literal is a **float**, it must end with an "F" or "f". Floating-point literals in exponential notation are also allowed, such as 3.14E2 or .19e10; the base is assumed to be 10.
- Character: In Java, character constants are assumed to be taken from the Unicode alphabet. Typically, a character is defined as an individual symbol enclosed in single quotes. For example, `'a'` and `'?'` are character constants. In addition, Java defines the following special character constants:

`'\n'`	(newline)	`'\t'`	(tab)
`'\b'`	(backspace)	`'\r'`	(return)
`'\f'`	(form feed)	`'\\'`	(backslash)
`'\''`	(single quote)	`'\"'`	(double quote).

- String Literal: A string literal is a sequence of characters enclosed in double quotes, for example, the following is a string literal:

```
"dogs cannot climb trees"
```

1.4.2 Operators

Java expressions involve composing literals and variables with operators. We will survey the operators in Java in this section.

Arithmetic Operators

The following are binary arithmetic operators in Java:

$+$	addition
$-$	subtraction
$*$	multiplication
$/$	division
%	the modulo operator

This last operator, modulo, is also known as the "remainder" operator, because it is the remainder left after an integer division. We often use "mod" to denote the modulo operator, and we define it formally as

$$n \bmod m = r,$$

such that

$$n = mq + r,$$

for an integer q and $0 \leq r < m$.

Java also provides a unary minus ($-$), which can be placed in front of an arithmetic expression to invert its sign. Parentheses can be used in any expression to define the order of evaluation. Java also uses a fairly intuitive operator precedence rule to determine the order of evaluation when parentheses are not used. Unlike C++, Java does not allow operator overloading for class types.

String Concatenation

With strings, the ($+$) operator performs ***concatenation***, so that the code

```
String rug = "carpet";
String dog = "spot";
String mess = rug + dog;
String answer = mess + " will cost me " + 5 + " hours!";
```

would have the effect of making answer refer to the string

```
"carpetspot will cost me 5 hours!"
```

This example also shows how Java converts nonstring values (such as 5) into strings, when they are involved in a string concatenation operation.

Increment and Decrement Operators

Like C and C++, Java provides increment and decrement operators. Specifically, it provides the plus-one increment (++) and decrement (−−) operators. If such an operator is used in front of a variable reference, then 1 is added to (or subtracted from) the variable and its value is read into the expression. If it is used after a variable reference, then the value is first read and then the variable is incremented or decremented by 1. So, for example, the code fragment

```
int i = 8;
int j = i++;          // j becomes 8 and then i becomes 9
int k = ++i;          // i becomes 10 and then k becomes 10
int m = i--;          // m becomes 10 and then i becomes 9
int n = 9 + --i;      // i becomes 8 and then n becomes 17
```

assigns 8 to j, 10 to k, 10 to m, 17 to n, and returns *i* to value 8, as noted.

Logical Operators

Java supports the standard comparisons operators between numbers:

<	less than
<=	less than or equal to
==	equal to
!=	not equal to
>=	greater than or equal to
>	greater than

The type of the result of any of these comparison is a **boolean**. Comparisons may also be performed on **char** values, with inequalities determined according to the underlying character codes.

For reference types, it is important to know that the operators == and != are defined so that expression a == b is true if a and b both refer to the identical object (or are both **null**). Most object types support an equals method, such that a.equals(b) is true if a and b refer to what are deemed as "equivalent" instances for that class (even if not the same instance); see Section 3.5 for further discussion.

Operators defined for **boolean** values are the following:

!	not (prefix)
&&	conditional and
||	conditional or

The boolean operators && and || will not evaluate the second operand (to the right) in their expression if it is not needed to determine the value of the expression. This ***"short circuiting"*** feature is useful for constructing boolean expressions where we first test that a certain condition holds (such as an array index being valid) and then test a condition that could have otherwise generated an error condition had the prior test not succeeded.

Bitwise Operators

Java also provides the following bitwise operators for integers and booleans:

$\sim$	bitwise complement (prefix unary operator)
&	bitwise and
|	bitwise or
^	bitwise exclusive-or
<<	shift bits left, filling in with zeros
>>	shift bits right, filling in with sign bit
>>>	shift bits right, filling in with zeros

The Assignment Operator

The standard assignment operator in Java is "=". It is used to assign a value to an instance variable or local variable. Its syntax is as follows:

variable = *expression*

where *variable* refers to a variable that is allowed to be referenced by the statement block containing this expression. The value of an assignment operation is the value of the expression that was assigned. Thus, if j and k are both declared as type **int**, it is correct to have an assignment statement like the following:

```
j = k = 25;      // works because '=' operators are evaluated right-to-left
```

Compound Assignment Operators

Besides the standard assignment operator (=), Java also provides a number of other assignment operators that combine a binary operation with an assignment. These other kinds of operators are of the following form:

variable op= expression

where *op* is any binary operator. The above expression is generally equivalent to

variable = variable op expression

so that x *= 2 is equivalent to x = x * 2. However, if *variable* contains an expression (for example, an array index), the expression is evaluated only once. Thus, the code fragment

```
a[5] = 10;
j = 5;
a[j++] += 2;                    // not the same as a[j++] = a[j++] + 2
```

leaves a[5] with value 12 and j with value 6.

Operator Precedence

Operators in Java are given preferences, or precedence, that determine the order in which operations are performed when the absence of parentheses brings up evaluation ambiguities. For example, we need a way of deciding if the expression, "5+2*3," has value 21 or 11 (Java says it is 11). We show the precedence of the operators in Java (which, incidentally, is the same as in C and C++) in Table 1.3.

Operator Precedence		
	Type	**Symbols**
1	array index method call dot operator	[ ] ( ) .
2	postfix ops prefix ops cast	*exp*++ *exp*−− ++*exp* −−*exp* +*exp* −*exp* ~*exp* !*exp* (*type*) *exp*
3	mult./div.	* / %
4	add./subt.	+ −
5	shift	<< >> >>>
6	comparison	< <= > >= **instanceof**
7	equality	== !=
8	bitwise-and	&
9	bitwise-xor	^
10	bitwise-or	|
11	and	&&
12	or	||
13	conditional	*booleanExpression* ? *valueIfTrue* : *valueIfFalse*
14	assignment	= += −= *= /= %= <<= >>= >>>= &= ^= |=

Table 1.3: The Java precedence rules. Operators in Java are evaluated according to the ordering above if parentheses are not used to determine the order of evaluation. Operators on the same line are evaluated in left-to-right order (except for assignment and prefix operations, which are evaluated in right-to-left order), subject to the conditional evaluation rule for boolean && and || operations. The operations are listed from highest to lowest precedence (we use *exp* to denote an atomic or parenthesized expression). Without parenthesization, higher precedence operators are performed before lower precedence operators.

We have now discussed almost all of the operators listed in Table 1.3. A notable exception is the conditional operator, which involves evaluating a boolean expression and then taking on the appropriate value depending on whether this boolean expression is true or false. (We discuss the use of the **instanceof** operator in the next chapter.)

1.4.3 Type Conversions

Casting is an operation that allows us to change the type of a value. In essence, we can take a value of one type and ***cast*** it into an equivalent value of another type. There are two forms of casting in Java: ***explicit casting*** and ***implicit casting***.

Explicit Casting

Java supports an explicit casting syntax with the following form:

(*type*) *exp*

where *type* is the type that we would like the expression *exp* to have. This syntax may only be used to cast from one primitive type to another primitive type, or from one reference type to another reference type. We will discuss its use between primitives here, and between reference types in Section 2.5.1.

Casting from an **int** to a **double** is known as a ***widening*** cast, as the **double** type is more broad than the **int** type, and a conversion can be performed without losing information. But a cast from a **double** to an **int** is a ***narrowing*** cast; we may lose precision, as any fractional portion of the value will be truncated. For example, consider the following:

```
double d1 = 3.2;
double d2 = 3.9999;
int i1 = (int) d1;                    // i1 gets value 3
int i2 = (int) d2;                    // i2 gets value 3
double d3 = (double) i2;              // d3 gets value 3.0
```

Although explicit casting cannot directly convert a primitive type to a reference type, or vice versa, there are other means for performing such type conversions. We already discussed, as part of Section 1.3, conversions between Java's primitive types and corresponding wrapper classes (such as **int** and Integer). For convenience, those wrapper classes also provide static methods that convert between their corresponding primitive type and String values.

For example, the Integer.toString method accepts an **int** parameter and returns a String representation of that integer, while the Integer.parseInt method accepts a String as a parameter and returns the corresponding **int** value that the string represents. (If that string does not represent an integer, a NumberFormatException results.) We demonstrate their use as follows:

```
String s1 = "2014";
int i1 = Integer.parseInt(s1);        // i1 gets value 2014
int i2 = -35;
String s2 = Integer.toString(i2);     // s2 gets value "-35"
```

Similar methods are supported by other wrapper types, such as Double.

Implicit Casting

There are cases where Java will perform an ***implicit cast*** based upon the context of an expression. For example, you can perform a *widening* cast between primitive types (such as from an **int** to a **double**), without explicit use of the casting operator. However, if attempting to do an implicit *narrowing* cast, a compiler error results. For example, the following demonstrates both a legal and an illegal implicit cast via assignment statements:

```
int i1 = 42;
double d1 = i1;          // d1 gets value 42.0
i1 = d1;                 // compile error: possible loss of precision
```

Implicit casting also occurs when performing arithmetic operations involving a mixture of numeric types. Most notably, when performing an operation with an integer type as one operand and a floating-point type as the other operand, the integer value is implicitly converted to a floating-point type before the operation is performed. For example, the expression 3 + 5.7 is implicitly converted to 3.0 + 5.7 before computing the resulting **double** value of 8.7.

It is common to combine an explicit cast and an implicit cast to perform a floating-point division on two integer operands. The expression (**double**) 7 / 4 produces the result 1.75, because operator precedence dictates that the cast happens first, as ((**double**) 7) / 4, and thus 7.0 / 4, which implicitly becomes 7.0 / 4.0. Note however that the expression, (**double**) (7 / 4) produces the result 1.0.

Incidentally, there is one situation in Java when only implicit casting is allowed, and that is in string concatenation. Any time a string is concatenated with any object or base type, that object or base type is automatically converted to a string. Explicit casting of an object or base type to a string is not allowed, however. Thus, the following assignments are incorrect:

```
String s = 22;                          // this is wrong!
String t = (String) 4.5;                // this is wrong!
String u = "Value = " + (String) 13;    // this is wrong!
```

To perform a conversion to a string, we must use the appropriate toString method or perform an implicit cast via the concatenation operation. Thus, the following statements are correct:

```
String s = Integer.toString(22);        // this is good
String t = "" + 4.5;                    // correct, but poor style
String u = "Value = " + 13;             // this is good
```

1.5 Control Flow

Control flow in Java is similar to that of other high-level languages. We review the basic structure and syntax of control flow in Java in this section, including method returns, **if** statements, **switch** statements, loops, and restricted forms of "jumps" (the **break** and **continue** statements).

1.5.1 The If and Switch Statements

In Java, conditionals work similarly to the way they work in other languages. They provide a way to make a decision and then execute one or more different statement blocks based on the outcome of that decision.

The If Statement

The syntax of a simple **if** statement is as follows:

if (*booleanExpression*)
    trueBody
else
    falseBody

where *booleanExpression* is a boolean expression and *trueBody* and *falseBody* are each either a single statement or a block of statements enclosed in braces ("{" and "}"). Note that, unlike some similar languages, the value tested by an **if** statement in Java must be a boolean expression. In particular, it is definitely not an integer expression. Nevertheless, as in other similar languages, the **else** part (and its associated statement) in a Java **if** statement is optional. There is also a way to group a number of boolean tests, as follows:

if (*firstBooleanExpression*)
    firstBody
else if (*secondBooleanExpression*)
    secondBody
else
    thirdBody

If the first boolean expression is false, the second boolean expression will be tested, and so on. An **if** statement can have an arbitrary number of **else if** parts. Braces can be used for any or all statement bodies to define their extent.

As a simple example, a robot controller might have the following logic:

```
if (door.isClosed( ))
  door.open( );
advance( );
```

Notice that the final command, advance( ), is not part of the conditional body; it will be executed unconditionally (although after opening a closed door).

We may nest one control structure within another, relying on explicit braces to mark the extent of the various bodies if needed. Revisiting our robot example, here is a more complex control that accounts for unlocking a closed door.

```
if (door.isClosed( )) {
  if (door.isLocked( ))
    door.unlock( );
  door.open( );
}
advance( );
```

The logic expressed by this example can be diagrammed as a traditional ***flowchart***, as portrayed in Figure 1.4.

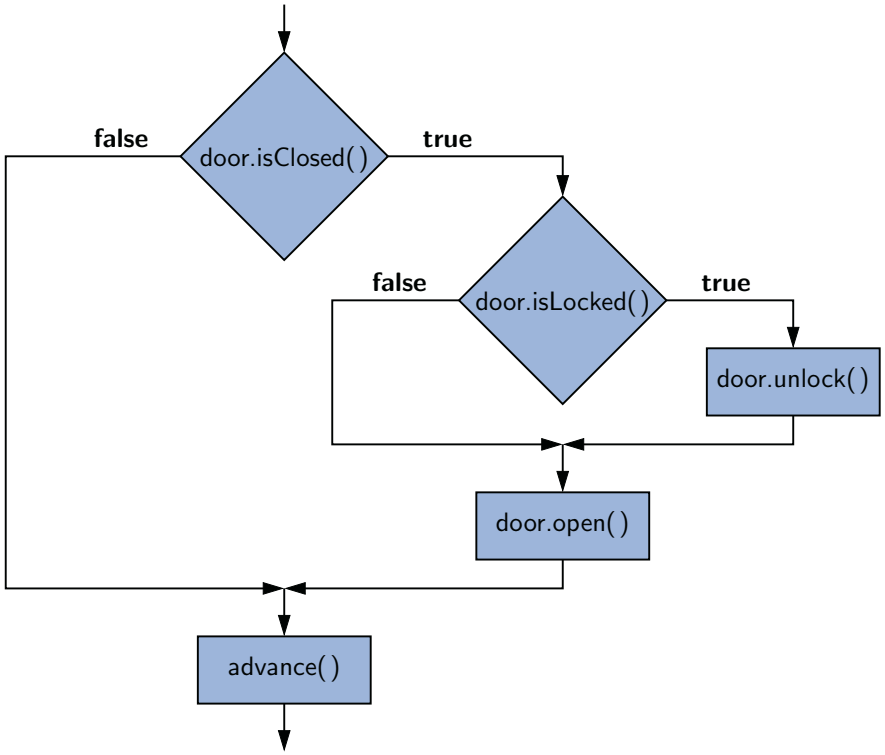


Figure 1.4: A flowchart describing the logic of nested conditional statements.

The following is an example of the nesting of **if** and **else** clauses.

```
if (snowLevel < 2) {
  goToClass();
  comeHome();
} else if (snowLevel < 5) {
  goSledding();
  haveSnowballFight();
} else
  stayAtHome();              // single-statement body needs no { } braces
```

Switch Statements

Java provides for multiple-value control flow using the **switch** statement, which is especially useful with enum types. The following is an indicative example (based on a variable d of the Day enum type of Section 1.3).

```
switch (d) {
  case MON:
    System.out.println("This is tough.");
    break;
  case TUE:
    System.out.println("This is getting better.");
    break;
  case WED:
    System.out.println("Half way there.");
    break;
  case THU:
    System.out.println("I can see the light.");
    break;
  case FRI:
    System.out.println("Now we are talking.");
    break;
  default:
    System.out.println("Day off!");
}
```

The **switch** statement evaluates an integer, string, or enum expression and causes control flow to jump to the code location labeled with the value of this expression. If there is no matching label, then control flow jumps to the location labeled "**default**." This is the only explicit jump performed by the **switch** statement, however, so flow of control "falls through" to the next case if the code for a case is not ended with a **break** statement (which causes control flow to jump to the end).

1.5.2 Loops

Another important control flow mechanism in a programming language is looping. Java provides for three types of loops.

While Loops

The simplest kind of loop in Java is a **while** loop. Such a loop tests that a certain condition is satisfied and will perform the body of the loop each time this condition is evaluated to be **true**. The syntax for such a conditional test before a loop body is executed is as follows:

while (*booleanExpression*)
    loopBody

As with an **if** statement, *booleanExpression*, can be an arbitrary boolean expression, and the body of the loop can be an arbitrary block of code (including nested control structures). The execution of a **while** loop begins with a test of the boolean condition. If that condition evaluates to **true**, the body of the loop is performed. After each execution of the body, the loop condition is retested and if it evaluates to **true**, another iteration of the body is performed. If the condition evaluates to **false** when tested (assuming it ever does), the loop is exited and the flow of control continues just beyond the body of the loop.

As an example, here is a loop that advances an index through an array named data until finding an entry with value target or reaching the end of the array.

```
int j = 0;
while ((j < data.length) && (data[j] != target))
  j++;
```

When this loop terminates, variable j's value will be the index of the leftmost occurrence of target, if found, or otherwise the length of the array (which is recognizable as an invalid index to indicate failure of the search). The correctness of the loop relies on the short-circuiting behavior of the logical && operator, as described on page 25. We intentionally test j < data.length to ensure that j is a valid index, prior to accessing element data[j]. Had we written that compound condition with the opposite order, the evaluation of data[j] would eventually throw an ArrayIndexOutOfBoundsException if the target is not found. (See Section 2.4 for discussion of exceptions.)

We note that a **while** loop will execute its body zero times in the case that the initial condition fails. For example, our above loop will not increment the value of j if data[0] matches the target (or if the array has length 0).

Do-While Loops

Java has another form of the **while** loop that allows the boolean condition to be checked at the *end* of each pass of the loop rather than before each pass. This form is known as a **do-while** loop, and has syntax shown below:

do
    loopBody
while (*booleanExpression*)

A consequence of the **do-while** loop is that its body always executes at least once. (In contrast, a **while** loop will execute zero times if the initial condition fails.) This form is most useful for a situation in which the condition is ill-defined until after at least one pass. Consider, for example, that we want to prompt the user for input and then do something useful with that input. (We discuss Java input and output in more detail in Section 1.6.) A possible condition, in this case, for exiting the loop is when the user enters an empty string. However, even in this case, we may want to handle that input and inform the user that he or she has quit. The following example illustrates this case:

```
String input;
do {
  input = getInputString( );
  handleInput(input);
} while (input.length( ) > 0);
```

For Loops

Another kind of loop is the **for** loop. Java supports two different styles of **for** loop. The first, which we will refer to as the "traditional" style, is patterned after a similar syntax as **for** loops in the C and C++ languages. The second style, which is known as the "for-each" loop, was introduced into Java in 2004 as part of the SE 5 release. This style provides a more succinct syntax for iterating through elements of an array or an appropriate container type.

The traditional **for**-loop syntax consists of four sections—an initialization, a boolean condition, an increment statement, and the body—although any of those can be empty. The structure is as follows:

for (*initialization*; *booleanCondition*; *increment*)
    loopBody

For example, the most common use of a **for** loop provides repetition based on an integer index, such as the following:

```
for (int j=0; j < n; j++)
  // do something
```

The behavior of a **for** loop is very similar to the following **while** loop equivalent:

```
{
    initialization;
    while (booleanCondition) {
        loopBody;
        increment;
    }
}
```

The *initialization* section will be executed once, before any other portion of the loop begins. Traditionally, it is used to either initialize existing variables, or to declare and initialize new variables. Note that any variables declared in the initialization section only exist in scope for the duration of the **for** loop.

The *booleanCondition* will be evaluated immediately before each potential iteration of the loop. It should be expressed similar to a **while**-loop condition, in that if it is **true**, the loop body is executed, and if **false**, the loop is exited and the program continues to the next statement beyond the **for**-loop body.

The *increment* section is executed immediately after each iteration of the formal loop body, and is traditionally used to update the value of the primary loop variable. However, the incrementing statement can be any legal statement, allowing significant flexibility in coding.

As a concrete example, here is a method that computes the sum of an array of **double** values using a **for** loop:

```
public static double sum(double[ ] data) {
  double total = 0;
  for (int j=0; j < data.length; j++)      // note the use of length
    total += data[j];
  return total;
}
```

As one further example, the following method computes the maximum value within a (nonempty) array.

```
public static double max(double[ ] data) {
  double currentMax = data[0];        // assume first is biggest (for now)
  for (int j=1; j < data.length; j++)      // consider all other entries
    if (data[j] > currentMax)              // if data[j] is biggest thus far...
      currentMax = data[j];                // record it as the current max
  return currentMax;
}
```

Notice that a conditional statement is nested within the body of the loop, and that no explicit "{" and "}" braces are needed for the loop body, as the entire conditional construct serves as a single statement.

For-Each Loop

Since looping through elements of a collection is such a common construct, Java provides a shorthand notation for such loops, called the ***for-each loop***. The syntax for such a loop is as follows:

```
for (elementType name : container)
    loopBody
```

where *container* is an array of the given *elementType* (or a collection that implements the Iterable interface, as we will later discuss in Section 7.4.1).

Revisiting a previous example, the traditional loop for computing the sum of elements in an array of **double** values can be written as:

```
public static double sum(double[ ] data) {
  double total = 0;
  for (double val : data)                    // Java's for-each loop style
    total += val;
  return total;
}
```

When using a for-each loop, there is no explicit use of array indices. The loop variable represents one particular element of the array. However, within the body of the loop, there is no designation as to which element it is.

It is also worth emphasizing that making an assignment to the loop variable has no effect on the underlying array. Therefore, the following method is an invalid attempt to scale all values of a numeric array.

```
public static void scaleBad(double[ ] data, double factor) {
  for (double val : data)
    val *= factor;                           // changes local variable only
}
```

In order to overwrite the values in the cells of an array, we must make use of indices. Therefore, this task is best solved with a traditional **for** loop, such as the following:

```
public static void scaleGood(double[ ] data, double factor) {
  for (int j=0; j < data.length; j++)
    data[j] *= factor;                       // overwrites cell of the array
}
```

1.5.3 Explicit Control-Flow Statements

Java also provides statements that cause explicit change in the flow of control of a program.

Returning from a Method

If a Java method is declared with a return type of **void**, then flow of control returns when it reaches the last line of code in the method or when it encounters a **return** statement (with no argument). If a method is declared with a return type, however, the method must exit by returning an appropriate value as an argument to a **return** statement. It follows that the **return** statement ***must*** be the last statement executed in a method, as the rest of the code will never be reached.

Note that there is a significant difference between a statement being the last line of code that is ***executed*** in a method and the last line of code in the method itself. The following (correct) example illustrates returning from a method:

```
public double abs(double value) {
  if (value < 0)                  // value is negative,
    return -value;                // so return its negation
  return value;                   // return the original nonnegative value
}
```

In the example above, the line **return** −value; is clearly not the last line of code that is written in the method, but it may be the last line that is executed (if the original value is negative). Such a statement explicitly interrupts the flow of control in the method. There are two other such explicit control-flow statements, which are used in conjunction with loops and switch statements.

The break Statement

We first introduced use of the **break** command, in Section 1.5.1, to exit from the body of a switch statement. More generally, it can be used to "break" out of the innermost **switch**, **for**, **while**, or **do-while** statement body. When it is executed, a break statement causes the flow of control to jump to the next line after the loop or **switch** to the body containing the **break**.

The continue Statement

A **continue** statement can be used within a loop. It causes the execution to skip over the remaining steps of the *current iteration* of the loop body, but then, unlike the **break** statement, the flow of control returns to the top of the loop, assuming its condition remains satisfied.

1.6 Simple Input and Output

Java provides a rich set of classes and methods for performing input and output within a program. There are classes in Java for doing graphical user interface design, complete with pop-up windows and pull-down menus, as well as methods for the display and input of text and numbers. Java also provides methods for dealing with graphical objects, images, sounds, Web pages, and mouse events (such as clicks, mouse overs, and dragging). Moreover, many of these input and output methods can be used in either stand-alone programs or in applets.

Unfortunately, going into the details on how all of the methods work for constructing sophisticated graphical user interfaces is beyond the scope of this book. Still, for the sake of completeness, we describe how simple input and output can be done in Java in this section.

Simple input and output in Java occurs within the Java console window. Depending on the Java environment we are using, this window is either a special pop-up window that can be used for displaying and inputting text, or a window used to issue commands to the operating system (such windows are referred to as shell windows, command windows, or terminal windows).

Simple Output Methods

Java provides a built-in static object, called System.out, that performs output to the "standard output" device. Most operating system shells allow users to redirect standard output to files or even as input to other programs, but the default output is to the Java console window. The System.out object is an instance of the java.io.PrintStream class. This class defines methods for a buffered output stream, meaning that characters are put in a temporary location, called a ***buffer***, which is then emptied when the console window is ready to print characters.

Specifically, the java.io.PrintStream class provides the following methods for performing simple output (we use *baseType* here to refer to any of the possible base types):

print(String *s*): Print the string *s*.

print(Object *o*): Print the object *o* using its toString method.

print(*baseType b*): Print the base type value *b*.

println(String *s*): Print the string *s*, followed by the newline character.

println(Object *o*): Similar to print(*o*), followed by the newline character.

println(*baseType b*): Similar to print(*b*), followed by the newline character.

An Output Example

Consider, for example, the following code fragment:

```
System.out.print("Java values: ");
System.out.print(3.1416);
System.out.print(',');
System.out.print(15);
System.out.println(" (double,char,int).");
```

When executed, this fragment will output the following in the Java console window:

```
Java values: 3.1416,15 (double,char,int).
```

Simple Input Using the java.util.Scanner Class

Just as there is a special object for performing output to the Java console window, there is also a special object, called System.in, for performing input from the Java console window. Technically, the input is actually coming from the "standard input" device, which by default is the computer keyboard echoing its characters in the Java console. The System.in object is an object associated with the standard input device. A simple way of reading input with this object is to use it to create a Scanner object, using the expression

new Scanner(System.in)

The Scanner class has a number of convenient methods that read from the given input stream, one of which is demonstrated in the following program:

```
import java.util.Scanner;                    // loads Scanner definition for our use

public class InputExample {
  public static void main(String[ ] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter your age in years: ");
    double age = input.nextDouble( );
    System.out.print("Enter your maximum heart rate: ");
    double rate = input.nextDouble( );
    double fb = (rate − age) ∗ 0.65;
    System.out.println("Your ideal fat-burning heart rate is " + fb);
  }
}
```

When executed, this program could produce the following on the Java console:

```
Enter your age in years: 21
Enter your maximum heart rate: 220
Your ideal fat-burning heart rate is 129.35
```

java.util.Scanner Methods

The Scanner class reads the input stream and divides it into ***tokens***, which are strings of characters separated by ***delimiters***. A delimiter is a special separating string, and the default delimiter is whitespace. That is, tokens are separated by strings of spaces, tabs, and newlines, by default. Tokens can either be read immediately as strings or a Scanner object can convert a token to a base type, if the token has the right form. Specifically, the Scanner class includes the following methods for dealing with tokens:

hasNext( ): Return **true** if there is another token in the input stream.

next( ): Return the next token string in the input stream; generate an error if there are no more tokens left.

hasNext*Type*( ): Return **true** if there is another token in the input stream and it can be interpreted as the corresponding base type, *Type*, where *Type* can be Boolean, Byte, Double, Float, Int, Long, or Short.

next*Type*( ): Return the next token in the input stream, returned as the base type corresponding to *Type*; generate an error if there are no more tokens left or if the next token cannot be interpreted as a base type corresponding to *Type*.

Additionally, Scanner objects can process input line by line, ignoring delimiters, and even look for patterns within lines while doing so. The methods for processing input in this way include the following:

hasNextLine( ): Returns **true** if the input stream has another line of text.

nextLine( ): Advances the input past the current line ending and returns the input that was skipped.

findInLine(String *s*): Attempts to find a string matching the (regular expression) pattern *s* in the current line. If the pattern is found, it is returned and the scanner advances to the first character after this match. If the pattern is not found, the scanner returns **null** and doesn't advance.

These methods can be used with those above, as in the following:

```
Scanner input = new Scanner(System.in);
System.out.print("Please enter an integer: ");
while (!input.hasNextInt( )) {
  input.nextLine( );
  System.out.print("Invalid integer; please enter an integer: ");
}
int i = input.nextInt( );
```

1.7 An Example Program

In this section, we present another example of a Java class, which illustrates many of the constructs defined thus far in this chapter. This CreditCard class defines credit card objects that model a simplified version of traditional credit cards. They store information about the customer, issuing bank, account identifier, credit limit, and current balance. They do not charge interest or late payments, but they do restrict charges that would cause a card's balance to go over its credit limit. We also provide a static main method as part of this class to demonstrate its use.

The primary definition of the CreditCard class is given in Code Fragment 1.5. We defer until Code Fragment 1.6 the presentation of the main method, and in Code Fragment 1.7 we show the output produced by the main method. Highlights of this class, and underlying techniques that are demonstrated, include:

- The class defines five instance variables (lines 3–7), four of which are declared as **private** and one that is **protected**. (We will take advantage of the **protected** balance member when introducing inheritance in the next chapter.)
- The class defines two different constructor forms. The first version (beginning at line 9) requires five parameters, including an explicit initial balance for the account. The second constructor (beginning at line 16) accepts only four parameters; it relies on use of the special **this** keyword to invoke the five-parameter version, with an explicit initial balance of zero (a reasonable default for most new accounts).
- The class defines five basic accessor methods (lines 20–24), and two update methods (charge and makePayment). The charge method relies on conditional logic to ensure that a charge is rejected if it would have resulted in the balance exceeding the credit limit on the card.
- We provide a **static** utility method, named printSummary, in lines 37–43.
- The main method includes an array, named wallet, storing CreditCard instances. The main method also demonstrates a **while** loop, a traditional **for** loop, and a for-each loop over the contents of the wallet.
- The main method demonstrates the syntax for calling traditional (nonstatic) methods—charge, getBalance, and makePayment—as well as the syntax for invoking the static printSummary method.

```
1  public class CreditCard {
2    // Instance variables:
3    private String customer;        // name of the customer (e.g., "John Bowman")
4    private String bank;            // name of the bank (e.g., "California Savings")
5    private String account;         // account identifier (e.g., "5391 0375 9387 5309")
6    private int limit;              // credit limit (measured in dollars)
7    protected double balance;       // current balance (measured in dollars)
8    // Constructors:
9    public CreditCard(String cust, String bk, String acnt, int lim, double initialBal) {
10     customer = cust;
11     bank = bk;
12     account = acnt;
13     limit = lim;
14     balance = initialBal;
15   }
16   public CreditCard(String cust, String bk, String acnt, int lim) {
17     this(cust, bk, acnt, lim, 0.0);                  // use a balance of zero as default
18   }
19   // Accessor methods:
20   public String getCustomer( ) { return customer; }
21   public String getBank( ) { return bank; }
22   public String getAccount( ) { return account; }
23   public int getLimit( ) { return limit; }
24   public double getBalance( ) { return balance; }
25   // Update methods:
26   public boolean charge(double price) {            // make a charge
27     if (price + balance > limit)                   // if charge would surpass limit
28       return false;                                // refuse the charge
29     // at this point, the charge is successful
30     balance += price;                              // update the balance
31     return true;                                   // announce the good news
32   }
33   public void makePayment(double amount) {         // make a payment
34     balance -= amount;
35   }
36   // Utility method to print a card's information
37   public static void printSummary(CreditCard card) {
38     System.out.println("Customer = " + card.customer);
39     System.out.println("Bank = " + card.bank);
40     System.out.println("Account = " + card.account);
41     System.out.println("Balance = " + card.balance);   // implicit cast
42     System.out.println("Limit = " + card.limit);       // implicit cast
43   }
44   // main method shown on next page...
45 }
```

Code Fragment 1.5: The CreditCard class.

```
1   public static void main(String[ ] args) {
2     CreditCard[ ] wallet = new CreditCard[3];
3     wallet[0] = new CreditCard("John Bowman", "California Savings",
4                                "5391 0375 9387 5309", 5000);
5     wallet[1] = new CreditCard("John Bowman", "California Federal",
6                                "3485 0399 3395 1954", 3500);
7     wallet[2] = new CreditCard("John Bowman", "California Finance",
8                                "5391 0375 9387 5309", 2500, 300);
9
10    for (int val = 1; val <= 16; val++) {
11      wallet[0].charge(3*val);
12      wallet[1].charge(2*val);
13      wallet[2].charge(val);
14    }
15
16    for (CreditCard card : wallet) {
17      CreditCard.printSummary(card);          // calling static method
18      while (card.getBalance( ) > 200.0) {
19        card.makePayment(200);
20        System.out.println("New balance = " + card.getBalance( ));
21      }
22    }
23  }
```

Code Fragment 1.6: The main method of the CreditCard class.

```
Customer = John Bowman
Bank = California Savings
Account = 5391 0375 9387 5309
Balance = 408.0
Limit = 5000
New balance = 208.0
New balance = 8.0
Customer = John Bowman
Bank = California Federal
Account = 3485 0399 3395 1954
Balance = 272.0
Limit = 3500
New balance = 72.0
Customer = John Bowman
Bank = California Finance
Account = 5391 0375 9387 5309
Balance = 436.0
Limit = 2500
New balance = 236.0
New balance = 36.0
```

Code Fragment 1.7: Output from the Test class.

1.8 Packages and Imports

The Java language takes a general and useful approach to the organization of classes into programs. Every stand-alone public class defined in Java must be given in a separate file. The file name is the name of the class with a `.java` extension. So a class declared as **public class** Window is defined in a file `Window.java`. That file may contain definitions for other stand-alone classes, but none of them may be declared with public visibility.

To aid in the organization of large code repository, Java allows a group of related type definitions (such as classes and enums) to be grouped into what is known as a ***package***. For types to belong to a package named *packageName*, their source code must all be located in a directory named *packageName* and each file must begin with the line:

package *packageName*;

By convention, most package names are lowercased. For example, we might define an architecture package that defines classes such as Window, Door, and Room. Public definitions within a file that does not have an explicit **package** declaration are placed into what is known as the ***default package***.

To refer to a type within a named package, we may use a fully qualified name based on dot notation, with that type treated as an attribute of the package. For example, we might declare a variable with architecture.Window as its type.

Packages can be further organized hierarchically into ***subpackages***. Code for classes in a subpackage must be located within a subdirectory of the package's directory, and qualified names for subpackages rely on further use of dot notation. For example, there is a java.util.zip subpackage (with support for working with ZIP compression) within the java.util package, and the Deflater class within that subpackage is fully qualified as java.util.zip.Deflater.

There are many advantages to organizing classes into packages, most notably:

- Packages help us avoid the pitfalls of name conflicts. If all type definitions were in a single package, there could be only one public class named Window. But with packages, we can have an architecture.Window class that is independent from a gui.Window class for graphical user interfaces.
- It is much easier to distribute a comprehensive set of classes for other programmers to use when those classes are packaged.
- When type definitions have a related purpose, it is often easier for other programmers to find them in a large library and to better understand their coordinated use when they are grouped as a package.
- Classes within the same package have access to any of each others' members having **public**, **protected**, or default visibility (i.e., anything but **private**).

Import Statements

As noted on the previous page, we may refer to a type within a package using its fully qualified name. For example, the Scanner class, introduced in Section 1.6, is defined in the java.util package, and so we may refer to it as java.util.Scanner. We could declare and construct a new instance of that class in a project using the following statement:

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

However, all the extra typing needed to refer to a class outside of the current package can get tiring. In Java, we can use the **import** keyword to include external classes or entire packages in the current file. To import an individual class from a specific package, we type the following at the beginning of the file:

import *packageName.className*;

For example, in Section 1.6 we imported the Scanner class from the java.util package with the command:

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

and then we were allowed to use the less burdensome syntax:

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

Note that it is illegal to import a class with the above syntax if a similarly named class is defined elsewhere in the current file, or has already been imported from another package. For example, we could not simultaneously import both architecture.Window and gui.Window to use with the unqualified name Window.

Importing a Whole Package

If we know we will be using many definitions from the same package, we can import all of them using an asterisk character (*) to denote a wildcard, as in the following syntax:

import *packageName*.*;

If a locally defined name conflicts with one in a package being imported in this way, the locally defined one retains the unqualified name. If there is a name conflict between definitions in two different packages being imported this way, *neither* of the conflicting names can be used without qualification. For example, if we import the following hypothetical packages:

```
import architecture.*;      // which we assume includes a Window class
import gui.*;               // which we assume includes a Window class
```

we must still use the qualified names architecture.Window and gui.Window in the rest of our program.

1.9 Software Development

Traditional software development involves several phases. Three major steps are:

1. Design
2. Coding
3. Testing and Debugging

In this section, we briefly discuss the role of these phases, and we introduce several good practices for programming in Java, including coding style, naming conventions, formal documentation, and testing.

1.9.1 Design

For object-oriented programming, the design step is perhaps the most important phase in the process of developing software. It is in the design step that we decide how to divide the workings of our program into classes, when we decide how these classes will interact, what data each will store, and what actions each will perform. Indeed, one of the main challenges that beginning programmers face is deciding what classes to define to do the work of their program. While general prescriptions are hard to come by, there are some rules of thumb that we can apply when determining how to define our classes:

- ***Responsibilities***: Divide the work into different ***actors***, each with a different responsibility. Try to describe responsibilities using action verbs. These actors will form the classes for the program.
- ***Independence***: Define the work for each class to be as independent from other classes as possible. Subdivide responsibilities between classes so that each class has autonomy over some aspect of the program. Give data (as instance variables) to the class that has jurisdiction over the actions that require access to this data.
- ***Behaviors***: Define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it. These behaviors will define the methods that this class performs, and the set of behaviors for a class form the ***protocol*** by which other pieces of code will interact with objects from the class.

Defining the classes, together with their instance variables and methods, are key to the design of an object-oriented program. A good programmer will naturally develop greater skill in performing these tasks over time, as experience teaches him or her to notice patterns in the requirements of a program that match patterns that he or she has seen before.

A common tool for developing an initial high-level design for a project is the use of ***CRC cards***. Class-Responsibility-Collaborator (CRC) cards are simple index cards that subdivide the work required of a program. The main idea behind this tool is to have each card represent a component, which will ultimately become a class in the program. We write the name of each component on the top of an index card. On the left-hand side of the card, we begin writing the responsibilities for this component. On the right-hand side, we list the collaborators for this component, that is, the other components that this component will have to interact with to perform its duties.

The design process iterates through an action/actor cycle, where we first identify an action (that is, a responsibility), and we then determine an actor (that is, a component) that is best suited to perform that action. The design is complete when we have assigned all actions to actors. In using index cards for this process (rather than larger pieces of paper), we are relying on the fact that each component should have a small set of responsibilities and collaborators. Enforcing this rule helps keep the individual classes manageable.

As the design takes form, a standard approach to explain and document the design is the use of UML (Unified Modeling Language) diagrams to express the organization of a program. UML diagrams are a standard visual notation to express object-oriented software designs. Several computer-aided tools are available to build UML diagrams. One type of UML figure is known as a ***class diagram***.

An example of a class diagram is given in Figure 1.5, corresponding to our CreditCard class from Section 1.7. The diagram has three portions, with the first designating the name of the class, the second designating the recommended instance variables, and the third designating the recommended methods of the class. The type declarations of variables, parameters, and return values are specified in the appropriate place following a colon, and the visibility of each member is designated on its left, with the "+" symbol for **public** visibility, the "#" symbol for **protected** visibility, and the "−" symbol for **private** visibility.

class:	CreditCard	
fields:	− customer : String − bank : String − account : String	− limit : int # balance : double
methods:	+ getCustomer( ) : String + getBank( ) : String + charge(price : double) : boolean + makePayment(amount : double)	+ getAccount( ) : String + getLimit( ) : int + getBalance( ) : double

Figure 1.5: A UML Class diagram for the CreditCard class from Section 1.7.

1.9.2 Pseudocode

As an intermediate step before the implementation of a design, programmers are often asked to describe algorithms in a way that is intended for human eyes only. Such descriptions are called ***pseudocode***. Pseudocode is not a computer program, but is more structured than usual prose. It is a mixture of natural language and high-level programming constructs that describe the main ideas behind a generic implementation of a data structure or algorithm. Because pseudocode is designed for a human reader, not a computer, we can communicate high-level ideas without being burdened by low-level implementation details. At the same time, we should not gloss over important steps. Like many forms of human communication, finding the right balance is an important skill that is refined through practice.

There really is no precise definition of the pseudocode language. At the same time, to help achieve clarity, pseudocode mixes natural language with standard programming language constructs. The programming language constructs that we choose are those consistent with modern high-level languages such as C, C++, and Java. These constructs include the following:

- ***Expressions:*** We use standard mathematical symbols to express numeric and boolean expressions. To be consistent with Java, we use the equal sign "=" as the assignment operator in assignment statements, and the "==" relation to test equivalence in boolean expressions.
- ***Method declarations:*** **Algorithm** name$(param1, param2, \ldots)$ declares new method "name" and its parameters.
- ***Decision structures:*** **if** condition **then** true-actions [**else** false-actions]. We use indentation to indicate what actions should be included in the true-actions and false-actions.
- ***While-loops:*** **while** condition **do** actions. We use indentation to indicate what actions should be included in the loop actions.
- ***Repeat-loops:*** **repeat** actions **until** condition. We use indentation to indicate what actions should be included in the loop actions.
- ***For-loops:*** **for** variable-increment-definition **do** actions. We use indentation to indicate what actions should be included among the loop actions.
- ***Array indexing:*** $A[i]$ represents the i^{th} cell in the array A. The cells of an n-celled array A are indexed from $A[0]$ to $A[n-1]$ (consistent with Java).
- ***Method calls:*** object.method(args); object is optional if it is understood.
- ***Method returns:*** **return** value. This operation returns the value specified to the method that called this one.
- ***Comments:*** { Comment goes here. }. We enclose comments in braces.

1.9.3 Coding

One of the key steps in implementing an object-oriented program is coding the descriptions of classes and their respective data and methods. In order to accelerate the development of this skill, we will discuss various ***design patterns*** for designing object-oriented programs (see Section 2.1.3) at various points throughout this text. These patterns provide templates for defining classes and the interactions between these classes.

Once we have settled on a design for the classes or our program and their responsibilities, and perhaps drafted pseudocode for their behaviors, we are ready to begin the actual coding on a computer. We type the Java source code for the classes of our program by using either an independent text editor (such as emacs, WordPad, or vi), or the editor embedded in an ***integrated development environment*** (IDE), such as Eclipse.

Once we have completed coding for a class (or package), we compile this file into working code by invoking a compiler. If we are not using an IDE, then we compile our program by calling a program, such as `javac`, on our file. If we are using an IDE, then we compile our program by clicking the appropriate compilation button. If we are fortunate, and our program has no syntax errors, then this compilation process will create files with a "`.class`" extension.

If our program contains syntax errors, then these will be identified, and we will have to go back into our editor to fix the offending lines of code. Once we have eliminated all syntax errors, and created the appropriate compiled code, we can run our program by either invoking a command, such as "`java`" (outside an IDE), or by clicking on the appropriate "run" button (within an IDE). When a Java program is run in this way, the runtime environment locates the directories containing the named class and any other classes that are referenced from this class according to a special operating system environment variable named "CLASSPATH." This variable defines an order of directories in which to search, given as a list of directories, which are separated by colons in Unix/Linux or semicolons in DOS/Windows. An example CLASSPATH assignment in the DOS/Windows operating system could be the following:

```
SET CLASSPATH=.;C:\java;C:\Program Files\Java\
```

Whereas an example CLASSPATH assignment in the Unix/Linux operating system could be the following:

```
setenv CLASSPATH ".:/usr/local/java/lib:/usr/netscape/classes"
```

In both cases, the dot (".") refers to the current directory in which the runtime environment is invoked.

1.9.4 Documentation and Style

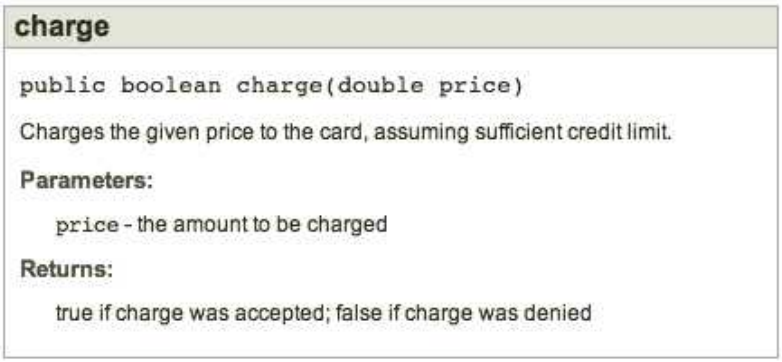
Javadoc

In order to encourage good use of block comments and the automatic production of documentation, the Java programming environment comes with a documentation production program called ***javadoc***. This program takes a collection of Java source files that have been commented using certain keywords, called ***tags***, and it produces a series of HTML documents that describe the classes, methods, variables, and constants contained in these files. As an example, Figure 1.6 shows a portion of the documentation generated for our CreditCard class.

Each javadoc comment is a block comment that starts with "`/**`" and ends with "`*/`", and each line between these two can begin with a single asterisk, "`*`", which is ignored. The block comment is assumed to start with a descriptive sentence, which is followed by special lines that begin with javadoc tags. A block comment that comes just before a class definition, instance variable declaration, or method definition is processed by javadoc into a comment about that class, variable, or method. The primary javadoc tags that we use are the following:

- `@author` *text*: Identifies each author (one per line) for a class.
- `@throws` *exceptionName description*: Identifies an error condition that is signaled by this method (see Section 2.4).
- `@param` *parameterName description*: Identifies a parameter accepted by this method.
- `@return` *description*: Describes the return type and its range of values for a method.

There are other tags as well; the interested reader is referred to online documentation for javadoc for further information. For space reasons, we cannot always include javadoc-style comments in all the example programs included in this book, but we include such a sample in Code Fragment 1.8, and within the online code at the website that accompanies this book.



charge

`public boolean charge(double price)`

Charges the given price to the card, assuming sufficient credit limit.

Parameters:

`price` - the amount to be charged

Returns:

true if charge was accepted; false if charge was denied

Figure 1.6: Documentation rendered by javadoc for the CreditCard.charge method.

```
/**
 * A simple model for a consumer credit card.
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwasser
 */
public class CreditCard {
  /**
   * Constructs a new credit card instance.
   * @param cust        the name of the customer (e.g., "John Bowman")
   * @param bk          the name of the bank (e.g., "California Savings")
   * @param acnt        the account identifier (e.g., "5391 0375 9387 5309")
   * @param lim         the credit limit (measured in dollars)
   * @param initialBal  the initial balance (measured in dollars)
   */
  public CreditCard(String cust, String bk, String acnt, int lim, double initialBal) {
    customer = cust;
    bank = bk;
    account = acnt;
    limit = lim;
    balance = initialBal;
  }

  /**
   * Charges the given price to the card, assuming sufficient credit limit.
   * @param price  the amount to be charged
   * @return true  if charge was accepted; false if charge was denied
   */
  public boolean charge(double price) {          // make a charge
    if (price + balance > limit)                 // if charge would surpass limit
      return false;                              // refuse the charge
    // at this point, the charge is successful
    balance += price;                            // update the balance
    return true;                                 // announce the good news
  }

  /**
   * Processes customer payment that reduces balance.
   * @param amount  the amount of payment made
   */
  public void makePayment(double amount) {     // make a payment
    balance -= amount;
  }
  // remainder of class omitted...
```

Code Fragment 1.8: A portion of the CreditCard class definition, originally from Code Fragment 1.5, with javadoc-style comments included.

Readability and Programming Conventions

Programs should be made easy to read and understand. Good programmers should therefore be mindful of their coding style, and develop a style that communicates the important aspects of a program's design for both humans and computers. Much has been written about good coding style, with some of the main principles being the following:

- Use meaningful names for identifiers. Try to choose names that can be read aloud, and choose names that reflect the action, responsibility, or data each identifier is naming. The tradition in most Java circles is to capitalize the first letter of each word in an identifier, except for the first word for a variable or method name. By this convention, "Date," "Vector," "DeviceManager" would identify classes, and "isFull( )," "insertItem( )," "studentName," and "studentHeight" would respectively identify methods and variables.
- Use named constants or enum types instead of literals. Readability, robustness, and modifiability are enhanced if we include a series of definitions of named constant values in a class definition. These can then be used within this class and others to refer to special values for this class. The tradition in Java is to fully capitalize such constants, as shown below:

```
public class Student {
  public static final int MIN_CREDITS = 12;   // min credits per term
  public static final int MAX_CREDITS = 24;  // max credits per term
  public enum Year {FRESHMAN, SOPHOMORE, JUNIOR, SENIOR};

  // Instance variables, constructors, and method definitions go here...
}
```

- Indent statement blocks. Typically programmers indent each statement block by 4 spaces; in this book we typically use 2 spaces, however, to avoid having our code overrun the book's margins.
- Organize each class in the following order:
  1. Constants
  2. Instance variables
  3. Constructors
  4. Methods

  We note that some Java programmers prefer to put instance variable definitions last. We put them earlier so that we can read each class sequentially and understand the data each method is working with.
- Use comments that add meaning to a program and explain ambiguous or confusing constructs. In-line comments are good for quick explanations and do not need to be sentences. Block comments are good for explaining the purpose of a method and complex code sections.

1.9.5 Testing and Debugging

Testing is the process of experimentally checking the correctness of a program, while debugging is the process of tracking the execution of a program and discovering the errors in it. Testing and debugging are often the most time-consuming activity in the development of a program.

Testing

A careful testing plan is an essential part of writing a program. While verifying the correctness of a program over all possible inputs is usually infeasible, we should aim at executing the program on a representative subset of inputs. At the very minimum, we should make sure that every method of a program is tested at least once (method coverage). Even better, each code statement in the program should be executed at least once (statement coverage).

Programs often tend to fail on ***special cases*** of the input. Such cases need to be carefully identified and tested. For example, when testing a method that sorts (that is, puts in order) an array of integers, we should consider the following inputs:

- The array has zero length (no elements).
- The array has one element.
- All the elements of the array are the same.
- The array is already sorted.
- The array is reverse sorted.

In addition to special inputs to the program, we should also consider special conditions for the structures used by the program. For example, if we use an array to store data, we should make sure that boundary cases, such as inserting or removing at the beginning or end of the subarray holding data, are properly handled.

While it is essential to use handcrafted test suites, it is also advantageous to run the program on a large collection of randomly generated inputs. The Random class in the `java.util` package provides several means for generating pseudorandom numbers.

There is a hierarchy among the classes and methods of a program induced by the caller-callee relationship. Namely, a method A is above a method B in the hierarchy if A calls B. There are two main testing strategies, ***top-down*** testing and ***bottom-up*** testing, which differ in the order in which methods are tested.

Top-down testing proceeds from the top to the bottom of the program hierarchy. It is typically used in conjunction with ***stubbing***, a boot-strapping technique that replaces a lower-level method with a ***stub***, a replacement for the method that simulates the functionality of the original. For example, if method A calls method B to get the first line of a file, when testing A we can replace B with a stub that returns a fixed string.

Bottom-up testing proceeds from lower-level methods to higher-level methods. For example, bottom-level methods, which do not invoke other methods, are tested first, followed by methods that call only bottom-level methods, and so on. Similarly a class that does not depend upon any other classes can be tested before another class that depends on the former. This form of testing is usually described as ***unit testing***, as the functionality of a specific component is tested in isolation of the larger software project. If used properly, this strategy better isolates the cause of errors to the component being tested, as lower-level components upon which it relies should have already been thoroughly tested.

Java provides several forms of support for automated testing. We have already discussed how a class's static main method can be repurposed to perform tests of the functionality of that class (as was done in Code 1.6 for the CreditCard class). Such a test can be executed by invoking the Java virtual machine directly on this secondary class, rather than on the primary class for the entire application. When Java is started on the primary class, any code within such secondary main methods will be ignored.

More robust support for automation of unit testing is provided by the `JUnit` framework, which is not part of the standard Java toolkit but freely available at `www.junit.org`. This framework allows the grouping of individual test cases into larger test suites, and provides support for executing those suites, and reporting or analyzing the results of those tests. As software is maintained, ***regression testing*** should be performed, whereby automation is used to re-execute all previous tests to ensure that changes to the software do not introduce new bugs in previously tested components.

Debugging

The simplest debugging technique consists of using ***print statements*** to track the values of variables during the execution of the program. A problem with this approach is that eventually the print statements need to be removed or commented out, so they are not executed when the software is finally released.

A better approach is to run the program within a ***debugger***, which is a specialized environment for controlling and monitoring the execution of a program. The basic functionality provided by a debugger is the insertion of ***breakpoints*** within the code. When the program is executed within the debugger, it stops at each breakpoint. While the program is stopped, the current value of variables can be inspected. In addition to fixed breakpoints, advanced debuggers allow specification of ***conditional breakpoints***, which are triggered only if a given expression is satisfied.

The standard Java toolkit includes a basic debugger named `jdb`, which has a command-line interface. Most IDEs for Java programming provide advanced debugging environments with graphical user interfaces.

1.10 Exercises

Reinforcement

R-1.1 Write a short Java method, inputAllBaseTypes, that inputs a different value of each base type from the standard input device and prints it back to the standard output device.

R-1.2 Suppose that we create an array A of GameEntry objects, which has an integer scores field, and we clone A and store the result in an array B. If we then immediately set $A[4]$.score equal to 550, what is the score value of the GameEntry object referenced by $B[4]$?

R-1.3 Write a short Java method, isMultiple, that takes two **long** values, n and m, and returns true if and only if n is a multiple of m, that is, $n = mi$ for some integer i.

R-1.4 Write a short Java method, isEven, that takes an **int** i and returns true if and only if i is even. Your method cannot use the multiplication, modulus, or division operators, however.

R-1.5 Write a short Java method that takes an integer n and returns the sum of all positive integers less than or equal to n.

R-1.6 Write a short Java method that takes an integer n and returns the sum of all the odd positive integers less than or equal to n.

R-1.7 Write a short Java method that takes an integer n and returns the sum of the squares of all positive integers less than or equal to n.

R-1.8 Write a short Java method that counts the number of vowels in a given character string.

R-1.9 Write a short Java method that uses a StringBuilder instance to remove all the punctuation from a string s storing a sentence, for example, transforming the string `"Let's try, Mike!"` to `"Lets try Mike"`.

R-1.10 Write a Java class, Flower, that has three instance variables of type **String**, **int**, and **float**, which respectively represent the name of the flower, its number of petals, and price. Your class must include a constructor method that initializes each variable to an appropriate value, and your class should include methods for setting the value of each type, and getting the value of each type.

R-1.11 Modify the CreditCard class from Code Fragment 1.5 to include a method that updates the credit limit.

R-1.12 Modify the CreditCard class from Code Fragment 1.5 so that it ignores any request to process a negative payment amount.

R-1.13 Modify the declaration of the first **for** loop in the main method in Code Fragment 1.6 so that its charges will cause exactly one of the three credit cards to attempt to go over its credit limit. Which credit card is it?

Creativity

C-1.14 Write a pseudocode description of a method that reverses an array of n integers, so that the numbers are listed in the opposite order than they were before, and compare this method to an equivalent Java method for doing the same thing.

C-1.15 Write a pseudocode description of a method for finding the smallest and largest numbers in an array of integers and compare that to a Java method that would do the same thing.

C-1.16 Write a short program that takes as input three integers, a, b, and c, from the Java console and determines if they can be used in a correct arithmetic formula (in the given order), like "$a+b=c$," "$a=b-c$," or "$a*b=c$."

C-1.17 Write a short Java method that takes an array of **int** values and determines if there is a pair of distinct elements of the array whose product is even.

C-1.18 The ***p-norm*** of a vector $v=(v_1,v_2,\ldots,v_n)$ in n-dimensional space is defined as

$$\|v\| = \sqrt[p]{v_1^p + v_2^p + \cdots + v_n^p}.$$

For the special case of $p=2$, this results in the traditional ***Euclidean norm***, which represents the length of the vector. For example, the Euclidean norm of a two-dimensional vector with coordinates $(4,3)$ has a Euclidean norm of $\sqrt{4^2+3^2} = \sqrt{16+9} = \sqrt{25} = 5$. Give an implementation of a method named norm such that norm(v, p) returns the p-norm value of v and norm(v) returns the Euclidean norm of v, where v is represented as an array of coordinates.

C-1.19 Write a Java program that can take a positive integer greater than 2 as input and write out the number of times one must repeatedly divide this number by 2 before getting a value less than 2.

C-1.20 Write a Java method that takes an array of **float** values and determines if all the numbers are different from each other (that is, they are distinct).

C-1.21 Write a Java method that takes an array containing the set of all integers in the range 1 to 52 and shuffles it into random order. Your method should output each possible order with equal probability.

C-1.22 Write a short Java program that outputs all possible strings formed by using the characters `'c'`, `'a'`, `'t'`, `'d'`, `'o'`, and `'g'` exactly once.

C-1.23 Write a short Java program that takes two arrays a and b of length n storing **int** values, and returns the dot product of a and b. That is, it returns an array c of length n such that $c[i] = a[i] \cdot b[i]$, for $i = 0,\ldots,n-1$.

C-1.24 Modify the CreditCard class from Code Fragment 1.5 so that printSummary becomes a *nonstatic* method, and modify the main method from Code Fragment 1.6 accordingly.

C-1.25 Modify the CreditCard class to add a toString( ) method that returns a String representation of the card (rather than printing it to the console, as done by printSummary). Modify the main method from Code Fragment 1.6 accordingly to use the standard println command.

Projects

P-1.26 Write a short Java program that takes all the lines input to standard input and writes them to standard output in reverse order. That is, each line is output in the correct order, but the ordering of the lines is reversed.

P-1.27 Write a Java program that can simulate a simple calculator, using the Java console as the exclusive input and output device. That is, each input to the calculator, be it a number, like `12.34` or `1034`, or an operator, like `+` or `=`, can be done on a separate line. After each such input, you should output to the Java console what would be displayed on your calculator.

P-1.28 A common punishment for school children is to write out a sentence multiple times. Write a Java stand-alone program that will write out the following sentence one hundred times: "I will never spam my friends again." Your program should number each of the sentences and it should make eight different random-looking typos.

P-1.29 The ***birthday paradox*** says that the probability that two people in a room will have the same birthday is more than half, provided n, the number of people in the room, is more than 23. This property is not really a paradox, but many people find it surprising. Design a Java program that can test this paradox by a series of experiments on randomly generated birthdays, which test this paradox for $n = 5, 10, 15, 20, \ldots, 100$.

P-1.30 *(For those who know Java graphical user interface methods:)* Define a GraphicalTest class that tests the functionality of the CreditCard class from Code Fragment 1.5 using text fields and buttons.

Chapter Notes

For more detailed information about the Java programming language, we refer the reader to the Java website (http://www.java.com), as well as some of the fine books about Java, including the books by Arnold, Gosling and Holmes [8], Flanagan [33], and Horstmann and Cornell [47, 48].