CHAPTER 4

Defining Classes I

4.1 CLASS DEFINITIONS 160

Instance Variables and Methods 164

More about Methods 166

Tip: Any Method Can Be Used as a void Method 173

Local Variables 174

Blocks 174

Tip: Declaring Variables in a for Statement 175

Parameters of a Primitive Type 176

Pitfall: Use of the Terms "Parameter" and "Argument" 181

The this Parameter 182

Simple Cases with Class Parameters 185

Methods that Return a Boolean Value 185

The Methods equals and toString 187

Tip: Testing Methods 191

Recursive Methods 192

4.2 INFORMATION HIDING AND ENCAPSULATION 192

public and private Modifiers 193

Example: Yet Another Date Class 194

Accessor and Mutator Methods 199

Preconditions and Postconditions 201

Tip: A Class Has Access to Private Members of All Objects of the Class 201

4.3 OVERLOADING 202

Rules for Overloading 203

Pitfall: Overloading and Automatic Type Conversion 206

Pitfall: You Cannot Overload Based on the Type Returned 208

4.4 CONSTRUCTORS 210

Constructor Definitions 210

Tip: You Can Invoke Another Method in a Constructor 218

Example: The Final Date Class 219

Tip: Include a No-Argument Constructor 219

Default Variable Initializations 221

An Alternative Way to Initialize Instance Variables 221

The StringTokenizer Class 221

CHAPTER SUMMARY 226

ANSWERS TO SELF-TEST EXERCISES 227

PROGRAMMING PROJECTS 231

4 Defining Classes I

This is the exciting part.
This is like the Supremes.
See the way it builds up?

Attributed to Frank Zappa

Introduction

Classes are the single most important language feature that facilitates object-oriented programming (OOP), the dominant programming methodology in use today. You have already been using predefined classes. `String` and `BufferedReader` are two of the classes we have used. An object is a value of a class type and is referred to as *an instance of the class.* An object differs from a value of a primitive type in that it has methods (actions) as well as data. For example, `"Hello"` is an object of the class `String`. It has the characters in the string as its data and also has a number of methods, such as `length`.

You already know how to use classes, objects, and methods. This chapter tells you how to define classes and their methods. In Java, the act of programming consists of defining a number of classes. Every program is a class; all helping software consists of classes; all programmer-defined types are classes; classes are central to Java.

Prerequisites

This chapter uses material from Chapters 1, 2, and 3.

4.1 Class Definitions

The Time has come the walrus said
to talk of many things
of shoes and ships and sealing wax
of cabbages and kings.

Lewis Carroll, *Through the Looking-Glass*

A Java program consists of objects, from various classes, interacting with one another. Before we go into the details of how you define classes, let's review some of the general properties of classes. A value of a class type is called an **object**. An object is usually referred to as an object of the class or as an

object

instance of the class rather than as a value of the class, but it is a value of the class type. An object is a value of the class type much like a value, such as `5`, of a primitive type, like `int`, is a value of a variable of that type. However, an object typically has multiple pieces of data and has **methods** (actions) it can take. Each object can have different data but all objects of a class have the same types of data and all objects in a class have the same methods. We tend to speak of the data and methods as belonging to the object, and that is an acceptable point of view. The data certainly does belong to the object, but since all objects in a class have the same methods, it also would be correct to say the methods belong to the class. To make this abstract discussion come alive, we need a sample definition.

instance

method

A Class Is a Type

If A is a class, then the phrases "bla is of type A," "bla is an instance of the class A," and "bla is an object of the class A" mean the same thing.

Display 4.1 contains a definition for a class named `DateFirstTry` and a program that demonstrates using the class. Objects of this class represent dates like December 31, 2006 and July 4, 1776. This class is unrealistically simple, but it will serve to introduce you to the syntax for a class definition. Each object of this class has three pieces of data: a string for the month name, an integer for the day of the month, and another integer for the year. The objects have only one method, which is named `writeOutput`. Both the data items and the methods are sometimes called **members** of the object, because they belong to the object. The data items are also sometimes called **fields**. We will call the data items **instance variables** and call the methods *methods*.

member

field

instance variable

The following three lines from the start of the class definition define three instance variables (three data members):

```
public String month; //always 3 letters long, as in Jan, Feb, etc.
public int day;
public int year; //a four digit number.
```

The word `public` simply means that there are no restrictions on how these instance variables are used. Each of these lines declares one instance variable name. You can think of an object of the class as a complex item with instance variables inside of it. So, you can think of an instance variable as a smaller variable inside each object of the class. In this case, the instance variables are called `month`, `day`, and `year`.

An object of a class is typically named by a variable of the class type. For example, the program `DateFirstTryDemo` in Display 4.1 declares the two variables `date1` and `date2` to be of type `DateFirstTry`, as follows:

```
DateFirstTry date1, date2;
```

Display 4.1 A Simple Class

This class definition goes in a file named `DateFirstTry.java`.

```
1   public class DateFirstTry
2   {
3       public String month; //always 3 letters long, as in Jan, Feb, etc.
4       public int day;
5       public int year; //a four digit number.

6       public void writeOutput()
7       {
8           System.out.println(month + " " + day + ", " + year);
9       }
10  }
```

Later in this chapter we will see that these three `public` modifiers should be replaced with `private`.

This class definition (program) goes in a file named `DateFirstTryDemo.java`.

```
1   public class DateFirstTryDemo
2   {
3      public static void main(String[] args)
4      {
5          DateFirstTry date1, date2;
6          date1 = new DateFirstTry();
7          date2 = new DateFirstTry();
8          date1.month = "Dec";
9          date1.day = 31;
10         date1.year = 2006;
11         System.out.println("date1:");
12         date1.writeOutput();

13         date2.month = "Jul";
14         date2.day = 4;
15         date2.year = 1776;
16         System.out.println("date2:");
17         date2.writeOutput();
18     }
19  }
```

Sample Dialogue

```
date1
Dec 31, 2006
date2
Jul 4, 1776
```

This gives us variables of the class `DateFirstTry`, but so far there are no objects of the class. Objects are class values that are named by the variables. To obtain an object, you must use the `new` operator to create a "new" object. For example, the following creates an object of the class `DateFirstTry` and names it with the variable `date1`:

new

```
date1 = new DateFirstTry();
```

We will discuss this kind of statement in more detail later in this chapter when we discuss something called a *constructor.* For now simply note that

```
Class_Variable = new Class_Name();
```

creates a new object of the specified class and associates it with the class variable.[1] Since the class variable now names an object of the class, we will often refer to the class variable as an object of the class. (This is really the same usage as when we refer to an `int` variable `n` as "the integer `n`," even though the integer is, strictly speaking, not `n` but the value of `n`.)

Unlike what we did in Display 4.1, the declaration of a class variable and the creation of the object are more typically combined into one statement as follows:

```
DateFirstTry date1 = new DateFirstTry();
```

The new Operator

The new operator is used to create an object of a class and associate the object with a variable that names it.

Syntax:

```
Class_Variable = new Class_Name();
```

Example:

```
DateFirstTry date;
date = new DateFirstTry();
```

which is usually written in the following equivalent form:

```
DateFirstTry date = new DateFirstTry();
```

[1] For many the word "new" suggests a memory allocation. As we will see, the new operator does indeed produce a memory allocation.

INSTANCE VARIABLES AND METHODS

We will illustrate the details about instance variables using the class and program in Display 4.1. Each object of the class `DateFirstTry` has three instance variables, which can be named by giving the object name followed by a dot and the name of the instance variable. For example, the object `date1` in the program `DateFirstTryDemo` has the following three instance variables:

```
date1.month
date1.day
date1.year
```

Similarly, if you replace `date1` with `date2`, you obtain the three instance variables for the object `date2`. Note that `date1` and `date2` together have a total of six instance variables. The instance variables `date1.month` and `date2.month`, for example, are two different (instance) variables.

The instance variables in Display 4.1 can be used just like any other variables. For example, `date1.month` can be used just like any other variable of type `String`. The instance variables `date1.day` and `date1.year` can be used just like any other variables of type `int`. Thus, although the following is not in the spirit of the class definition, it is legal and would compile:

```
date1.month = "Hello friend.";
```

More likely assignments to instance variables are given in the program `DateFirstTry-Demo`.

The class `DateFirstTry` has only one method, which is named `writeOutput`. We reproduce the definition of the method here:

```
public void writeOutput()   <—— Heading
{
    System.out.println(month + " " + day + ", " + year);   > Body
}
```

heading
body

All method definitions belong to some class and all method definitions are given inside the definition of the class to which they belong. A method definition is divided into two parts, a **heading** and a **method body**, as illustrated by the annotation on the method definition. The word `void` means this is a method for performing an action as opposed to producing a value. We will say more about method definitions later in this chapter (including some indication of why the word `void` was chosen to indicate an action). You have already been using methods for predefined classes. The way you invoke a method from a class definition you write is the same as how you do it for a predefined class. For example, the following from the program `DateFirstTryDemo` is an invocation of the method `writeOutput` with `date1` as the calling object:

```
date1.writeOutput();
```

File Names and Locations

Remember a file must be named the same as the class it contains with an added `.java` at the end. For example, a class named `MyClass` must be in a file named `MyClass.java`.

We will eventually see other ways to arrange files, but at this point, your program and all the classes it uses should be in the same directory (same folder).

Class Definition

The following shows the form of a class definition that is most commonly used; however, it is legal to intermix the method definitions and the instance variable declarations.

Syntax:

```
public class Class_Name
{
    Instance_Variable_Declaration_1
    Instance_Variable_Declaration_2
         . . .
    Instance_Variable_Declaration_Last

    Method_Definition_1
    Method_Definition_2
         . . .
    Method_Definition_Last
}
```

Examples:

See Displays 4.1 and 4.2.

This invocation is equivalent to execution of the method body. So, this invocation is equivalent to

```
System.out.println(month + " " + day + ", " + year);
```

However, we need to say more about exactly how this is equivalent. If you simply replace the method invocation with this `System.out.println` statement, you will get a compiler error message. Note that within the definition for the method `writeOutput`, the names of the instance variables are used without any calling object. This is because the method will be invoked with different calling objects at different times. When an

instance variable is used in a method definition, it is understood to be the instance variable of the calling object. So in the program `DateFirstTryDemo`,

```
date1.writeOutput();
```

is equivalent to

```
System.out.println(date1.month + " " + date1.day
                                   + ", " + date1.year);
```

Similarly,

```
date2.writeOutput();
```

is equivalent to

```
System.out.println(date2.month + " " + date2.day
                                    + ", " + date2.year);
```

Self-Test Exercises

1. Write a method called `makeItNewYears` that could be added to the class `DateFirstTry` in Display 4.1. The method `makeItNewYears` has no parameters and sets the `month` instance variable to `"Jan"` and the `day` instance variable to `1`. It does not change the `year` instance variable.

2. Write a method called `yellIfNewYear` that could be added to the class `DateFirstTry` in Display 4.1. The method `yellIfNewYear` has no parameters and outputs the string `"Hurrah!"` provided the `month` instance variable has the value `"Jan"` and the `day` instance variable has the value `1`. Otherwise, it outputs the string `"Not New Year's Day."`.

MORE ABOUT METHODS

As we noted for predefined methods, methods of the classes you define are of two kinds: methods that return (compute) some value and methods that perform an action other than returning a value. For example, the method `println` of the object `System.out` is an example of a method that performs an action other than returning a value; in this case, the action is to write something to the screen. The method `readLine` of the class `BufferedReader`, introduced in Chapter 2, is a method that returns a value; in this case, the value returned is the string of characters typed in by the user. A method that performs some action other than returning a value is called a `void` method. This same distinction between `void` methods and methods that return a value applies to methods in the classes you define. The two kinds of methods require slight differences in how they are defined.

Both kinds of methods have a method heading and a method body, both of which are similar but not identical for the two kinds of methods. The method heading for a `void` method is of the form

```
public void Method_Name(Parameter_List)
```

The method heading for a method that returns a value is

```
public Type_Returned Method_Name(Parameter_List)
```

Later in the chapter we will see that `public` may sometimes be replaced by a more restricted modifier and that it is possible to add additional modifiers, but these templates will do for now. For now, our examples will have an empty *Parameter_List*.

If a method returns a value, then it can return different values in different situations, but all values returned must be of the same type, which is specified as the type returned. For example, if a method has the heading

```
public double myMethod()
```

then the method always returns a value of type `double`, and the heading

```
public String yourMethod()
```

indicates a method that always returns a value of type `String`.

The following is a `void` method heading:

```
public void ourMethod()
```

Notice that when the method returns no value at all, we use the keyword `void` in place of a type. If you think of `void` as meaning "no returned type," the word `void` begins to make sense.

invocation

An invocation of a method that returns a value can be used as an expression anyplace that a value of the *Type_Returned* can be used. For example, suppose `anObject` is an object of a class with methods having our sample heading; in that case, the following are legal:

```
double d = anObject.myMethod();
String aStringVariable = anObject.yourMethod();
```

A `void` method does not return a value, but simply performs an action, so an invocation of a `void` method is a statement. A `void` method is invoked as in the following example:

```
anObject.ourMethod();
```

Note the ending semicolon.

So far, we have avoided the topic of parameter lists by only giving examples with empty parameter lists, but note that parentheses are required even for an empty parameter list. Parameter lists will be discussed later in this chapter.

body

The body of a `void` method definition is simply a list of declarations and statements enclosed in a pair of braces, `{}`. For example, the following is a complete `void` method definition:

```
public void ourMethod()
{
    System.out.println("Hello");
    System.out.println("from our method.");
}
```

The body of a method that returns a value is the same as the body of a `void` method but with one additional requirement. The body of a method that returns a value must contain at least one `return` statement. A `return` statement is of the form

`return` statement

```
return Expression;
```

where *Expression* can be any expression that evaluates to something of the *Type_Returned*, which is listed in the method heading. For example, the following is a complete definition of a method that returns a value:

```
public String yourMethod()//not quite right yet
{
    BufferedReader keyboard = new BufferedReader(
                                  new InputStreamReader(System.in));
    System.out.println("Enter a line of text");
    String result = keyboard.readLine();
    return result + " was entered.";
}
```

Notice that a method that returns a value can do other things besides returning a value, but style rules dictate that whatever else it does should be related to the value returned.

`throws IOException`

As noted in the comment, the definition of the method `yourMethod` is not quite correct yet. Just as you need the phrase `throws IOException` on the `main` method of a program that uses the method `readline` of the class `BufferedReader`, you also need the same magic formula on the heading of any other method that uses the method `readline` of the class `BufferedReader`. So, the complete and correct definition is

```
public String yourMethod() throws IOException
{
    BufferedReader keyboard = new BufferedReader(
                                  new InputStreamReader(System.in));
    System.out.println("Enter a line of text");
    String result = keyboard.readLine();
    return result + " was entered.";

}
```

We should also note that if there is an invocation of the method `yourMethod` inside the definition of another method (including the possibility of the `main` method), then that method (the one that contains an invocation of `yourMethod`) in effect contains an invocation of `readLine` and so that method also requires the phrase `throws IOException`.

`throws` IOEXCEPTION

Any method (including the method `main`) that contains an invocation of the method `readLine` of the class `BufferedReader` must have the following phrase at the end of the method heading:

```
throws IOException
```

This will be explained in Chapter 9 when we cover exceptions, and we will then see how to do things other than adding this phrase. Until then the phrase `throws IOException` is just a magic formula that must be there.

A `return` statement always ends a method invocation. Once the `return` statement is executed, the method ends and any remaining statements in the method definition are not executed.

`return` STATEMENTS

The definition of a method that returns a value must have one or more `return` statements. A `return` statement specifies the value returned by the method and it ends the method invocation.

SYNTAX:

```
return Expression;
```

EXAMPLE:

```
public int getYear()
{
    return year;
}
```

A `void` method definition need not have a `return` statement. However, a `return` statement can be used in a `void` method to cause the method to immediately end. The form for a `return` statement in a `void` method is

```
return;
```

`return` in a `void` method

If you want to end a `void` method before it runs out of statements, you can use a `return` statement without any expression, as follows:

```
return;
```

A `void` method need not have any `return` statements, but you can place a `return` statement in a `void` method if there are situations that require the method to end before all the code is executed.

Although it may seem that we have lost sight of the fact, all these method definitions must be inside of some class definition. Java does not have any stand-alone methods that are not in any class. Display 4.2 rewrites the class given in Display 4.1 but this time we have added a more diverse set of methods.

METHOD DEFINITIONS

There are two kinds of methods: methods that return a value and methods, known as `void` methods, that perform some action other than returning a value.

DEFINITION OF A METHOD THAT RETURNS A VALUE:

SYNTAX:

```
public Type_Returned Method_Name(Parameter_List)
{
    <List of statements, at least one of which
          must contain a return statement.>
}
```

If there are no *Parameters,* then the parentheses are empty.

EXAMPLE:

```
public int getDay()
{
    return day;
}
```

`void` METHOD DEFINITION:

SYNTAX:

```
public void Method_Name(Parameter_List)
{
   <List of statements>
}
```

If there are no *Parameters,* then the parentheses are empty.

Example:

```
public void writeOutput()
{
    System.out.println(month + " " + day + ", " + year);
}
```

All method definitions are inside of some class definition. See Display 4.2 to see these example method definitions in the context of a class.

When an instance variable name is used in a method definition, it refers to an instance variable of the calling object.

Display 4.2 **A Class with More Methods *(Part 1 of 2)***

The significance of the modifier `private` is discussed in the subsection "`public` and `private` Modifiers" in Section 4.2 a bit later in this chapter.

```
1   import java.io.BufferedReader;
2   import java.io.InputStreamReader;
3   import java.io.IOException;

4   public class DateSecondTry
5   {
6       private String month; //always 3 letters long, as in Jan, Feb, etc.
7       private int day;
8       private int year; //a four digit number.

9       public void writeOutput()
10      {
11          System.out.println(month + " " + day + ", " + year);
12      }

13      public void readInput() throws IOException
14      {
15          BufferedReader keyboard;
16          keyboard = new BufferedReader(
17                           new InputStreamReader(System.in));
18          System.out.println("Enter month, day, and year on three lines:");
19          month = keyboard.readLine();
20          day = Integer.parseInt(keyboard.readLine());
21          year = Integer.parseInt(keyboard.readLine());
22      }

23      public int getDay()
24      {
25          return day;
26      }
```

Display 4.2 A Class with More Methods *(Part 2 of 2)*

```
27      public int getYear()
28      {
29          return year;
30      }
31      public int getMonth()
32      {
33          if (month.equals("Jan"))
34              return 1;
35          else if (month.equals("Feb"))
36              return 2;
37          else if (month.equals("Mar"))
38              return 3;
39          else if (month.equals("Apr"))
40              return 4;
41          else if (month.equals("May"))
42              return 5;
43          else if (month.equals("Jun"))
44              return 6;
45          else if (month.equals("Jul"))
46              return 7;
47          else if (month.equals("Aug"))
48              return 8;
49          else if (month.equals("Sep"))
50              return 9;
51          else if (month.equals("Oct"))
52              return 10;
53          else if (month.equals("Nov"))
54              return 11;
55          else if (month.equals("Dec"))
56              return 12;
57          else
58          {
59              System.out.println("Fatal Error");
60              System.exit(0);
61              return 0; //Needed to keep the compiler happy
62          }
63      }

64  }
```

Tip

Any Method Can Be Used as a void Method

A method that returns a value can also perform some action besides returning a value. If you want that action, but do not need the returned value, you can invoke the method as if it were a void method and the returned value will simply be discarded. For example, the following contains two invocations of the method readLine(), which returns a value of type String. Both are legal.

```
BufferedReader keyboard =
            new BufferedReader(new InputStreamReader(System.in));
     . . .
String inputString = keyboard.readLine();
     . . .
System.out.println("Press Enter to continue with program.");
keyboard.readLine(); //Reads a line and discards it.
```

Self-Test Exercises

3. Write a method called getNextYear that could be added to the class DateSecondTry in Display 4.2. The method getNextYear returns an int value equal to the value of the year instance variable plus one.

4. Consider the following method definition that might occur in some class:

```
public void echoLine() throws IOException
{
    BufferedReader keyboard = new BufferedReader(
                                   new InputStreamReader(System.in));
    System.out.println("Enter a line of text");
    String line = keyboard.readLine();
    System.out.println("You entered " + line);
}
```

Now suppose the same class also has the following method:

```
public void echo2Lines()
{
    echoLine();
    echoLine():
}
```

Is the definition of echo2Lines correct as written or do you need to add the phrase throws IOException to the first line of echo2Lines?

LOCAL VARIABLES

local variable

Look at the definition of the method `readInput()` given in Display 4.2. That method definition includes the declaration of a variable called `keyboard`. A variable declared within a method is called a **local variable**. It is called *local* because its meaning is local to—that is, confined to—the method definition. If you have two methods and each of them declares a variable of the same name—for example, if both were named `keyboard`—they would be two different variables that just happen to have the same name. Any change that is made to the variable named `keyboard` within one method would have no effect upon the variable named `keyboard` in the other method.

As we noted in Chapter 1, the `main` part of a program is itself a method. All variables declared in `main` are variables local to the method `main`. If a variable declared in `main` happens to have the same name as a variable declared in some other method, then they are two different variables that just happen to have the same name. Thus, all the variables we have seen so far are either local variables or instance variables. There is only one more kind of variable in Java, which is known as a *static variable.* Static variables will be discussed in Chapter 5.

Local Variable

A variable declared within a method definition is called a **local variable**. If two methods each have a local variable of the same name, they are two different variables that just happen to have the same name.

Global Variables

Thus far, we have discussed two kinds of variables: instance variables, whose meaning is confined to an object of a class, and local variables, whose meaning is confined to a method definition. Some other programming languages have another kind of variable called a **global variable**, whose meaning is confined only to the program. Java does not have these global variables.

BLOCKS

compound statement

block

The terms **block** and **compound statement** mean the same thing, namely, a set of Java statements enclosed in braces, `{}`. However, programmers tend to use the two terms in different contexts. When you declare a variable within a compound statement, the compound statement is usually called a *block.*

If you declare a variable within a block, that variable is local to the block. This means that when the block ends, all variables declared within the block disappear. In

Blocks

A **block** is another name for a compound statement—that is, a list of statements enclosed in braces. However, programmers tend to use the two terms in different contexts. When you declare a variable within a compound statement, the compound statement is usually called a *block*. The variables declared in a block are local to the block, and so these variables disappear when the execution of the block is completed. However, even though the variables are local to the block, their names cannot be used for anything else within the same method definition.

many programming languages, you can even use that variable's name to name some other variable outside of the block. However, *in Java, you cannot have two variables with the same name inside of a single method definition.* Local variables within blocks can sometimes create problems in Java. It is sometimes easier to declare the variables outside the block. If you declare a variable outside of a block, you can use it both inside and outside the block, and it will have the same meaning both inside the block and outside the block.

Tip

Declaring Variables in a `for` Statement

You can declare a variable (or variables) within the initialization portion of a `for` statement, as in the following:

```
int sum = 0;
for (int n = 1; n < 10; n++)
    sum = sum + n;
```

If you declare n in this way, the variable n will be *local to the* `for` *loop.* This means that n cannot be used outside of the `for` loop. For example, the following use of n in the `System.out.println` statement is illegal:

```
for (int n = 1; n < 10; n++)
    sum = sum + n;
System.out.println(n);//Illegal
```

Declaring variables inside a `for` loop can sometimes be more of a nuisance than a helpful feature. We tend to avoid declaring variables inside a `for` loop except for very simple cases that have no potential for confusion.

Self-Test Exercises

5. Write a method called `happyGreeting` that could be added to the class `DateSecondTry` in Display 4.2. The method `happyGreeting` writes the string `"Happy Days!"` to the screen a number of times equal to the value of the instance variable `day`. For example, if the value of `day` is `3`, then it writes the following to the screen:

```
Happy Days!
Happy Days!
Happy Days!
```

   Use a local variable.

PARAMETERS OF A PRIMITIVE TYPE

parameter

argument

All the method definitions we have seen thus far had no parameters, which was indicated by an empty set of parentheses in the method heading. **Parameters** are like a blank that is filled in with a particular value when the method is invoked. (What we are calling *parameters* are also called *formal parameters*.) The value that is plugged in for the parameter is called an **argument**. (Some programmers use the term *actual parameters* for what we are calling *arguments*.) We have already used arguments with predefined methods. For example, the string `"Hello"` is the argument to the method `println` in the following method invocation:

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

Display 4.3 contains the definition of a method named `setDate` that has the three parameters `newMonth`, `newDay`, and `newYear`. It also contains the definition of a method named `monthString` that has one parameter of type `int`.

The items plugged in for the parameters are called *arguments* and are given in parentheses at the end of the method invocation. For example, in the following call from Display 4.3, the integers `6` and `17` and the variable `year` are the arguments plugged in for `newMonth`, `newDay`, and `newYear`, respectively:

```
date.setDate(6, 17, year);
```

When you have a method invocation like the preceding, the argument (such as `6`) is plugged in for the corresponding formal parameter (such as `newMonth`) *everywhere that the parameter occurs in the method definition.* After all the arguments have been plugged in for their corresponding parameters, the code in the body of the method definition is executed.

The following invocation of the method `monthString` occurs within the definition of the method `setDate` in Display 4.3:

```
month = monthString(newMonth);
```

The argument is `newMonth`, which is plugged in for the parameter `monthNumber` in the definition of the method `monthString`.

Display 4.3 Methods with Parameters *(Part 1 of 2)*

The significance of the modifier `private` is discussed in the subsection "`public` and `private` Modifiers" in Section 4.2.

```
1  import java.io.BufferedReader;
2  import java.io.InputStreamReader;
3  import java.io.IOException;

4  public class DateThirdTry
5  {
6      private String month; //always 3 letters long, as in Jan, Feb, etc.
7      private int day;
8      private int year; //a four digit number.

9      public void setDate(int newMonth, int newDay, int newYear)
10     {
11         month = monthString(newMonth);
12         day = newDay;
13         year = newYear;
14     }

15     public String monthString(int monthNumber)
16     {
17         switch (monthNumber)
18         {
19         case 1:
20             return "Jan";
21         case 2:
22             return "Feb";
23         case 3:
24             return "Mar";
25         case 4:
26             return "Apr";
27         case 5:
28             return "May";
29         case 6:
30             return "Jun";
31         case 7:
32             return "Jul";
33         case 8:
34             return "Aug";
35          case 9:
36             return "Sep";
37         case 10:
38             return "Oct";
39         case 11:
40             return "Nov";
41         case 12:
42             return "Dec";
```

A better version of `setDate` will be given later in this chapter when we define `DateFourthTry`.

This is the file `DateThirdTry.java`.

Display 4.3 Methods with Parameters *(Part 2 of 2)*

```
43          default:
44              System.out.println("Fatal Error");
45              System.exit(0);
46              return "Error"; //to keep the compiler happy
47          }
48      }

        <The rest of the method definitions are identical to the ones given in Display 4.2.>
49  }
```

This is the file DateThirdTry.java.

```
1   public class DateThirdTryDemo
2   {
3       public static void main(String[] args)
4       {
5           DateThirdTry date = new DateThirdTry( );
6           int year = 1882;
7           date.setDate(6, 17, year);
8           date.writeOutput( );
9       }
10  }
```

This is the file DateThirdTryDemo.java.

The variable `year` is NOT plugged in for the parameter `newYear` in the definition of the method `setDate`. Only the value of `year`, namely `1882`, is plugged in for the parameter `newYear`.

Sample Dialogue

```
Jun 17, 1882
```

Note that each of the formal parameters must be preceded by a type name, even if there is more than one parameter of the same type. Corresponding arguments must match the type of their corresponding formal parameter, although in some simple cases an automatic type cast might be performed by Java. For example, if you plug in an argument of type `int` for a parameter of type `double`, then Java will automatically type cast the `int` value to a value of type `double`. The following list shows the type casts that Java will automatically perform for you. An argument in a method invocation that is of any of these types will automatically be type cast to any of the types that appear to its right if that is needed to match a formal parameter.[2]

```
byte --> short --> int --> long --> float --> double
```

[2] An argument of type `char` will also be converted to a matching number type, if the formal parameter is of type `int` or any type to the right of `int` in our list of types.

Note that this is exactly the same as the automatic type casting we discussed in Chapter 1 for storing values of one type in a variable of another type. The more general rule is that you can use a value of any of the listed types anywhere that Java expects a value of a type further down on the list.

Note that the correspondence of the parameters and arguments is determined by their order in the lists in parentheses. In a method invocation, there must be exactly the same number of arguments in parentheses as there are formal parameters in the method definition heading. The first argument in the method invocation is plugged in for the first parameter in the method definition heading, the second argument in the method invocation is plugged in for the second parameter in the heading of the method definition, and so forth. This is diagrammed in Display 4.4.

It is important to note that only the value of the argument is used in this substitution process. If an argument in a method invocation is a variable (such as `year` *in Display 4.3), it is*

Display 4.4 Correspondence Between Formal Parameters and Arguments

This is in the file DateThirdTry.java.

```
public class DateThirdTry
{
    private String month; //always 3 letters long, as in Jan, Feb, etc.
    private int day;
    private int year; //a four digit number.

    public void setDate(int newMonth, int newDay, int newYear)
    {
        month = monthString(newMonth);
        day = newDay;
        year = newYear;
    }
    ...
```

Only the value of year, namely 1882, is plugged in for the parameter newYear.

This is in the file DateThirdTryDemo.java. This is the file for a program that uses the class DateThirdTry.

```
public class DateThirdTryDemo
{
   public static void main(String[] args)
   {
        DateThirdTry date = new DateThirdTry( );
        int year = 1882;
        date.setDate(6, 17, year);
        date.writeOutput( );
    }
}
```

The arrows show which argument is plugged in for which formal parameter.

Parameters of a Primitive Type

Parameters are given in parentheses after the method name in the heading of a method definition. A parameter of a primitive type, such as `int`, `double`, or `char`, is a local variable. When the method is invoked, the parameter is initialized to the value of the corresponding argument in the method invocation. This mechanism is known as the **call-by-value** parameter mechanism. The argument in a method invocation can be a literal constant, like 2 or `'A'`; a variable; or any expression that yields a value of the appropriate type. This is the only kind of parameter that Java has for parameters of a primitive type. (Parameters of a class type are discussed in Chapter 5.)

`main` Is a `void` Method

The `main` part of a program is a `void` method, as indicated by its heading:

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

The word `static` will be explained in Chapter 5. The identifier `args` is a parameter of type `String[]`, which is the type for an array of strings. Arrays are discussed in Chapter 6, and you need not be concerned about them until then. In what we are doing in this book, we never use the parameter `args`. Since `args` is a parameter, you may replace it with any other non-keyword identifier and your program will have the same meaning. Aside from possibly changing the name of the parameter `args`, the heading of the `main` method must be exactly as shown above. Although we will not be using the parameter `args`, we will tell you how to use it in Chapter 6.

A program in Java is just a class that has a `main` method. When you give a command to run a Java program, the run-time system invokes the method `main`.

the value of the variable that is plugged in, not the variable name. For example, in Display 4.3 the value of the variable `year` (that is, `1882`) is plugged in for the parameter `newYear`. The variable `year` is not plugged in to the body of the method `setDate`. Because only the value of the argument is used, this method of plugging in arguments for formal parameters is known as the **call-by-value** mechanism. In Java, this is the only method of substitution that is used with parameters of a primitive type, such as `int`, `double`, and `char`. As you will eventually see, this is, strictly speaking, also the only method of substitution that is used with parameters of a class type. However, there are other differences that make parameters of a class type appear to use a different substitution mechanism. But for now, we are concerned only with parameters and arguments of primitive types, such as `int`, `double`, and `char`. (Although the type `String` is a class type, you will not go far wrong if you consider it to behave like a primitive type when an argument of type `String` is plugged in for its corresponding parameter. However,

call-by-value

for most class types, you need to think a bit differently about how arguments are plugged in for parameters. We discuss parameters of a class type in Chapter 5.)

parameters as local variables

In most cases, you can think of a parameter as a kind of blank, or placeholder, that is filled in by the value of its corresponding argument in the method invocation. However, parameters are more than just blanks that are filled in with the argument values for the method. A parameter is actually a local variable. When the method is invoked, the value of an argument is computed and the corresponding parameter, which is a local variable, is initialized to this value. Occasionally, it is useful to use a parameter as a local variable.

Pitfall

Use of the Terms "Parameter" and "Argument"

The use of the terms *parameter* and *argument* that we follow in this book is consistent with common usage, but people also often use the terms *parameter* and *argument* interchangeably. When you see the terms *parameter* and *argument,* you must determine their exact meaning from context. Many people use the term *parameter* for both what we call *parameters* and what we call *arguments.* Other people use the term *argument* both for what we call *parameters* and what we call *arguments.* Do not expect consistency in how people use these two terms.

formal parameter
actual parameter

The term **formal parameter** is often used for what we describe as a *parameter.* We will sometimes use the term *formal parameter* for emphasis. The term **actual parameter** is often used for what we call an *argument*. We do not use the term *actual parameter* in this book, but you will encounter it in other books.

Self-Test Exercises

6. Write a method called `fractionDone` that could be added to the class `DateThirdTry` in Display 4.3. The method `fractionDone` has a parameter `targetDay` of type `int` (for a day of the month) and returns a value of type `double`. The value returned is the value of the `day` instance variable divided by the `int` parameter `targetDay`. (So it returns the fraction of the time passed so far this month where the goal is reaching the `targetDay`.) Do floating-point division, not integer division. To get floating-point division, copy the value of the `day` instance variable into a local variable of type `double` and use this local variable in place of the `day` instance variable in the division. (You may assume the parameter `targetDay` is a valid day of the month that is greater than the value of the `day` instance variable.)

7. Write a method called `advanceYear` that could be added to the class `DateThirdTry` in Display 4.3. The method `advanceYear` has one parameter of type `int`. The method `advanceYear` increases the value of the `year` instance variable by the amount of this one parameter.

8. Suppose we redefine the method `setDate` in Display 4.3 to the following:

```
public void setDate(int newMonth, int newDay, int newYear)
{
    month = monthString(newMonth);
    day = newDay;
    year = newYear;
    System.out.println("Date changed to "
                       + newMonth + " " + newDay + ", " + newYear);
}
```

Indicate all instances of `newMonth` that have their value changed to `6` in the following invocation (also from Display 4.3):

```
date.setDate(6, 17, year);
```

9. Is the following a legal method definition that could be added to the class `DateThirdTry` in Display 4.3?

```
public void multiWriteOutput(int count)
{
    while (count > 0)
    {
        writeOutput();
        count--;
    }
}
```

10. Consider the definition of the method `monthString` in Display 4.3. Why are there no `break` statements in the `switch` statement?

THE this PARAMETER

As we noted earlier, if `today` is of type `DateSecondTry` (Display 4.2), then

```
today.writeOutput();
```

is equivalent to

```
System.out.println(today.month + " " + today.day
                                     + ", " + today.year);
```

This is because, although the definition of `writeOutput` reads

```
public void writeOutput()
{
    System.out.println(month + " " + day + ", " + year);
}
```

it really means

```
public void writeOutput()
{
    System.out.println(<the calling object>.month + " "
        + <the calling object>.day + ", " + <the calling object>.year);
}
```

The instance variables are understood to have `<the calling object>.` in front of them. Sometimes it is handy, and on rare occasions even necessary, to have an explicit name for the calling object. Inside a Java method definition, you can use the keyword `this` as a name for the calling object. So, the following is a valid Java method definition that is equivalent to the one we are discussing:

```
public void writeOutput()
{
    System.out.println(this.month + " " + this.day
                                  + ", " + this.year);
}
```

The definition of `writeOutput` in Display 4.2 could be replaced by this completely equivalent version. Moreover, this version is in some sense the true version. The version without the `this` and a dot in front of each instance variable is just an abbreviation for this version. However, the abbreviation of omitting the `this` is used frequently. The keyword `this` is known as the **`this` parameter**.

`this` parameter

The `this` Parameter

Within a method definition, you can use the keyword `this` as a name for the calling object. If an instance variable or another method in the class is used without any calling object, then `this` is understood to be the calling object.

There is one common situation that requires the use of the `this` parameter. You often want to have the parameters in a method such as `setDate` to be the same as the instance variables. A first, incorrect, try at doing this is the following rewriting of the method `setDate` from Display 4.3:

```
public void setDate(int month, int day, int year) //Not corrrect
{
    month = monthString(month);
    day = day;
    year = year;
}
```

This rewritten version does not do what we want. When you declare a local variable in a method definition, then within the method definition that name always refers to the local variable. A parameter is a local variable, so this rule applies to parameters. Consider the following assignment statement in our rewritten method definition:

```
day = day;
```

mask a variable

Both the identifiers `day` refer to the parameter named `day`. The identifier `day` does not refer to the instance variable `day`. All occurrences of the identifier `day` refer to the parameter `day`. This is often described by saying the parameter `day` **masks** or hides the instance variable `day`. Similar remarks apply to the parameters `month` and `year`.

This rewritten method definition of the method `setDate` will produce a compiler error message because the following attempts to assign a `String` value to the `int` variable (the parameter) `month`:

```
month = monthString(month);
```

However, in many situations, this sort of rewriting will produce a method definition that will compile but that will not do what it is supposed to do.

To correctly rewrite the method `setDate`, we need some way to say "the instance variable `month`" as opposed to the parameter `month`. The way to say "the instance variable `month`" is `this.month`. Similar remarks apply to the other two parameters. So, the correct rewriting of the method `setDate` is as follows:

```
public void setDate(int month, int day, int year)
{
    this.month = monthString(month);
    this.day = day;
    this.year = year;
}
```

This version is completely equivalent to the version in Display 4.3.

Self-Test Exercises

11. The method `writeOutput` in Display 4.2 uses the instance variables `month`, `day`, and `year`, but gives no object name for these instance variables. Every instance variable must belong to some object. To what object or objects do these instance variables in the definition of `writeOutput` belong?

12. Rewrite the definitions of the methods `getDay` and `getYear` in Display 4.2 using the `this` parameter.

13. Rewrite the method `getMonth` in Display 4.2 using the `this` parameter.

SIMPLE CASES WITH CLASS PARAMETERS

Methods can have parameters of a class type. Parameters of a class type are more subtle and more powerful than parameters of a primitive type. We will discuss parameters of class types in detail in Chapter 5. In the meantime, we will occasionally use a class type parameter in very simple situations. For these very simple cases, you need not know any details about class type parameters except that, in some sense or another, the class argument is plugged in for the class parameter.

METHODS THAT RETURN A BOOLEAN VALUE

There is nothing special about methods that return a value of type `boolean`. The type `boolean` is a primitive type, just like the types `int` and `double`. A method that returns a value of type `boolean` must have a `return` statement of the form

```
return Boolean_Expression;
```

So, an invocation of a method that returns a value of type `boolean` returns either `true` or `false`. It thus makes sense to use an invocation of such a method to control an `if-else` statement, to control a `while` loop, or anyplace else that a Boolean expression is allowed. Although there is nothing new here, people who have not used `boolean` valued methods before sometimes find them to be uncomfortable. So, we will go through one small example.

The following is a method definition that could be added to the class `DateThirdTry` in Display 4.3:

```
public boolean isBetween(int lowYear, int highYear)
{
     return ( (year > lowYear) && (year < highYear) );
}
```

Consider the following lines of code:

```
DateThirdTry date = new DateThirdTry();
date.setDate(1, 2, 3001);
if (date.isBetween(2000, 4000))
    System.out.println("The date is between the years 2000 and 4000");
else
    System.out.println(
                  "The date is not between the years 2000 and 4000");
```

The expression `date.isBetween(2000, 4000)` is an invocation of a method that returns a `boolean` value—that is, returns one of the two values `true` and `false`. So, it makes perfectly good sense to use it as the controlling Boolean expression in an `if-else` statement. The expression `year` in the definition of `isBetween` really means

this.year and this stands for the calling object. In date.isBetween(2000, 4000) the calling object is date. So, this returns the value

```
(date.year > lowYear) && (date.year < highYear)
```

But, 2000 and 4000 are plugged in for the parameters lowYear and highYear, respectively. So, this expression is equivalent to

```
(date.year > 2000) && (date.year < 4000)
```

Thus, the if-else statement is equivalent to[3]

```
if ((date.year > 2000) && (date.year < 4000))
    System.out.println("The date is between the years 2000 and 4000.");
else
    System.out.println(
                  "The date is not between the years 2000 and 4000.");
```

So, the output produced is

```
The date is between the years 2000 and 4000.
```

Another example of a boolean valued method, which we will in fact add to our date class, is shown below:

```
public boolean precedes(DateFourthTry otherDate)
{
    return ( (year < otherDate.year) ||
      (year == otherDate.year && getMonth() < otherDate.getMonth()) ||
      (year == otherDate.year && month.equals(otherDate.month)
                                    && day < otherDate.day) );
}
```

The version of our date class with this method is given in Display 4.5. The other new methods in that class will be discussed shortly in the subsection entitled "The Methods equals and toString." Right now, let's discuss this new method named precedes.

An invocation of the method precedes has the following form, where date1 and date2 are two objects of our date class:

```
date1.precedes(date2)
```

[3] Later in this chapter we will see that: Since year is marked private, it is not legal to write date.year in a program, but the meaning of such an expression is clear even if you cannot include it in a program.

This is a Boolean expression that returns `true` if `date1` comes before `date2`. Since it is a Boolean expression it can be used anyplace a Boolean expression is allowed, such as to control an `if-else` or `while` statement. For example,

```
if (date1.precedes(date2))
    System.out.println("date1 comes before date2.");
else
    System.out.println("date2 comes before or is equal to date1.");
```

The `return` statement in the definition of the method `precedes` may look intimidating, but is really straightforward. It says that `date1.precedes(date2)` returns `true`, provided one of the following three conditions is satisfied:

`date1.year < date2.year`

`date1.year` equals `date2.year` and `date1.month` comes before `date2.month`

`date1` and `date2` have the same year and month and also
`date1.day < date2.day`.

If you give it a bit of thought, you will realize that `date1` precedes `date2` in time precisely when one of these three conditions is satisfied.

THE METHODS equals AND toString

There are certain methods that Java expects to be in all, or almost all, classes. This is because some of the standard Java libraries have software that assumes such methods are defined. Two of these methods are `equals` and `toString`. Therefore, you should include such methods and be certain to spell their names exactly as we have done. Use `equals`, not `same` or `areEqual`. Do not even use `equal` without the `s`. Similar remarks apply to the `toString` method. After we have developed more material, we will explain this in more detail. In particular, we will then explain how to give a better method definition for `equals`. For now, just get in the habit of including them.

equals

The method `equals` is a `boolean` valued method to compare two objects of the class to see if they satisfy the intuitive notion of "being equal." So, the heading should be

```
public boolean equals(Class_Name Parameter_Name)
```

Display 4.5 contains definitions of the methods `equals` and `toString` that we might add to our date class, which is now named `DateFourthTry`. The heading of that `equals` method is

```
public boolean equals(DateFourthTry otherDate)
```

When you use the method `equals` to compare two objects of the class `DateFourthTry`, one object is the calling object and the other object is the argument, like so

```
date1.equals(date2)
```

Display 4.5 A Class with Methods equals and toString

```
1   import java.io.BufferedReader;
2   import java.io.InputStreamReader;
3   import java.io.IOException;

4   public class DateFourthTry
5   {
6       private String month; //always 3 letters long, as in Jan, Feb, etc.
7       private int day;
8       private int year; //a four digit number.

9       public String toString()
10      {
11          return (month + " " + day + ", " + year);
12      }

13      public void writeOutput()
14      {
15          System.out.println(month + " " + day + ", " + year);
16      }

17      public boolean equals(DateFourthTry otherDate)
18      {
19          return ( (month.equals(otherDate.month))
20                   && (day == otherDate.day) && (year == otherDate.year) );
21      }

22      public boolean precedes(DateFourthTry otherDate)
23      {
24          return ( (year < otherDate.year) ||
25             (year == otherDate.year && getMonth() < otherDate.getMonth()) ||
26             (year == otherDate.year && month.equals(otherDate.month)
27                                           && day < otherDate.day) );
28      }
```

This is the method equals in the class DateFourthTry.

This is the method equals in the class String.

<The rest of the method definitions are identical to the ones in DateThirdTry in Display 4.3.>

```
29  }
```

Display 4.6 **Using the Methods equals and toString**

```
1   public class EqualsAndToStringDemo
2   {
3       public static void main(String[] args)
4       {
5           DateFourthTry date1 = new DateFourthTry(),
6                         date2 = new DateFourthTry();
7           date1.setDate(6, 17, 1882);
8           date2.setDate(6, 17, 1882);

9           if (date1.equals(date2))
10              System.out.println(date1 + " equals " + date2);
11          else
12              System.out.println(date1 + " does not equal " + date2);

13          date1.setDate(7, 28, 1750);

14          if (date1.precedes(date2))
15              System.out.println(date1 + " comes before " + date2);
16          else
17              System.out.println(date2 + " comes before or is equal to "
18                                 + date1);
19      }
20  }
```

These are equivalent to `date1.toString()`.

These are equivalent to `date2.toString()`.

Sample Dialogue

```
Jun 17, 1882 equals Jun 17, 1882
Jul 28, 1750 comes before Jun 17, 1882
```

or equivalently

```
date2.equals(date1)
```

Since the method `equals` returns a value of type `boolean`, you can use an invocation of `equals` as the Boolean expression in an `if-else` statement, as shown in Display 4.6. Similarly, you can also use it anyplace else that a Boolean expression is allowed.

There is no absolute notion of "equality" that you must follow in your definition of `equals`. You can define the method `equals` any way you wish, but to be useful it should reflect some notion of "equality" that is useful for the software you are designing. A common way to define `equals` for simple classes of the kind we are looking at now is to say `equals` returns `true` if each instance variable of one object equals the corresponding instance variable of the other object. This is how we defined `equals` in Display 4.5.

If the definition of `equals` in Display 4.5 seems less than clear, it may help to rewrite it as follows using the `this` parameter:

```
public boolean equals(DateFourthTry otherDate)
{
    return ( ((this.month).equals(otherDate.month))
      && (this.day == otherDate.day) && (this.year == otherDate.year) );
}
```

So if `date1` and `date2` are objects of the class `DateFourthTry`, then `date1.equals(date2)` returns `true` provided the three instance variables in `date1` have values that are equal to the three instance variables in `date2`.

Also, note that the method in the definition of `equals` that is used to compare months is not the `equals` for the class `DateFourthTry` but the `equals` for the class `String`. You know this because the calling object, which is `this.month`, is of type `String`.

(Remember we use the `equals` method of the class `String` because `==` does not work correctly for comparing `String` values. This was discussed in the Pitfall section of Chapter 3 entitled "Using == with Strings.")

(In Chapter 7, you will see that there are reasons to make the definition of the `equals` method a bit more involved. But, the spirit of what an `equals` method should be is very much like what we are now doing, and it is the best we can do with what we know so far.)

`toString`

The method `toString` should be defined so that it returns a `String` value that represents the data in the object. One nice thing about the method `toString` is that it makes it easy to output an object to the screen. If `date` is of type `DateFourthTry`, then you can output the date to the screen as follows:

```
System.out.println(date.toString());
```

`println` used with objects

In fact, `System.out.println` was written so that it will automatically invoke `toString()` if you do not include it. So, the object `date` can also be output by the following simpler and equivalent statement:

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

This means that the method `writeOutput` in Display 4.5 is superfluous and could safely be omitted from the class definition.

If you look at Display 4.6, you will see that `toString` is called automatically even if the object is connected to some other string with a `+`, as in

```
System.out.println(date1 + " equals " + date2);
```

\+ used with objects

In this case, it is really the plus operator that causes the automatic invocation of `toString()`. So, the following is also legal:

```
String s = date1 + " equals " + date2;
```

The preceding is equivalent to

```
String s = date1.toString() + " equals " + date2.toString();
```

THE METHODS equals AND toString

Usually, your class definitions should contain an `equals` method and a `toString` method.

An `equals` method compares the calling object to another object and should return `true` when the two objects are intuitively equal. When comparing objects of a class type, you normally use the method `equals`, not `==`.

The `toString` method should return a string representation of the data in the calling object. If a class has a `toString` method, then you can use an object of the class as an argument to the methods `System.out.println` and `System.out.print`.

See Display 4.5 for an example of a class with `equals` and `toString` methods.

Tip

TESTING METHODS

Each method should be tested in a program in which it is the only untested program. If you test methods this way, then when you find an error, you will know which method contains the error. A program that does nothing but test a method is called a **driver program**.

driver program

If one method contains an invocation of another method in the same class, this can complicate the testing task. One way to test a method is to first test all the methods invoked by that method and then test the method itself. This is called **bottom-up testing**.

bottom-up testing

It is sometimes impossible or inconvenient to test a method without using some other method that has not yet been written or has not yet been tested. In this case, you can use a simplified version of the missing or untested method. These simplified methods are called **stubs**. These stubs will not necessarily perform the correct calculation, but they will deliver values that suffice for testing, and they are simple enough that you can have confidence in their performance. For example, the following is a possible stub:

stub

```
/**
 Computes the probability of rain based on temperature, barometric pressure,
 and relative humidity. Returns the probability as a fraction between 0 and 1.
*/
public double rainChance(double temperature,
                               double pressure, double humidity)
{
    return 0.5;//Not correct but good enough for a stub.
}
```

THE FUNDAMENTAL RULE FOR TESTING METHODS

Every method should be tested in a program in which every other method in the testing program has already been fully tested and debugged.

RECURSIVE METHODS

recursive method

Java does allow recursive method definitions. Recursive methods are covered in Chapter 11. If you do not know what recursive methods are, there is no need to be concerned until you reach that chapter. If you want to read about recursive methods early, you can read Sections 11.1 and 11.2 of Chapter 11 after you complete Chapter 5.

Self-Test Exercises

14. In the definition of `precedes` in Display 4.5, we used

```
month.equals(otherDate.month)
```

to test whether two months are equal, but we used

```
getMonth() < otherDate.getMonth()
```

to test whether one month comes before another. Why did we use `month` in one case and `getMonth` in another case?

15. What is the fundamental rule for testing methods?

4.2 Information Hiding and Encapsulation

We all know—the Times *knows—but we pretend we don't.*

Virginia Woolf, *Monday or Tuesday*

information hiding

Information hiding means that you separate the description of how to use a class and the implementation details such as how the class methods are defined. You do this so that a programmer who uses the class does not need to know the implementation details of the class definition. The programmer who uses the class can consider the implementation details as hidden since he or she need not look at them. Information hiding is a way of avoiding information overloading. It keeps the information needed by a programmer using the class within reasonable bounds. Another term for information hiding is **abstraction**. The use of the term *abstraction* for information hiding makes sense if you think about it a bit. When you abstract something you are discarding some of the details.

abstraction

Encapsulation means grouping software into a unit in such a way that it is easy to use because there is a well-defined simple interface. So, encapsulation and information hiding are two sides of the same coin.

encapsulation

Java has a way of officially hiding details of a class definition. To hide details, you mark them as `private`, a concept we discuss next.

ENCAPSULATION

Encapsulation means that the data and the actions are combined into a single item (in our case, a class object) and that the details of the implementation are hidden. The terms *information hiding* and *encapsulation* deal with the same general principle: If a class is well designed, a programmer who uses a class need not know all the details of the implementation of the class but need only know a much simpler description of how to use the class.

API

The term **API** stands for *application programming interface.* The API for a class is a description of how to use the class. If your class is well designed, using the encapsulation techniques we discuss in this book, then a programmer who uses your class need only read the API and need not look at the details of your code for the class definition.

ADT

The term **ADT** is short for *abstract data type.* An ADT is a data type that is written using good information-hiding techniques.

public AND private MODIFIERS

Compare the instance variables in Displays 4.1 and 4.2. In Display 4.1 each instance variable is prefaced with the modifier `public`. In Display 4.2 each instance variable is prefaced with the modifier `private`. The modifier `public` means that there are no restrictions on where the instance variable can be used. The modifier `private` means that the instance variable cannot be accessed by name outside of the class definition.

public

private

For example, the following would produce a compiler error message if used in a program:

```
DateSecondTry date = new DateSecondTry();
date.month = "Jan";
date.day = 1;
date.year = 2006;
```

In fact, any one of the three assignments would be enough to trigger a compiler error. This is because, as shown in Display 4.2, each of the instance variables `month`, `day`, and `year` is labeled `private`.

If, on the other hand, we had used the class `DateFirstTry` from Display 4.1 instead of the class `DateSecondTry` in the preceding code, then the code would be legal and would compile and run with no error messages. This is because, in the definition of `DateFirstTry` (Display 4.1), each of the instance variables `month`, `day`, and `year` is labeled `public`.

It is considered good programming practice to make all instance variables private. As we will explain a little later in this chapter, this is intended to simplify the task of any programmer using the class. But before we say anything about how, on balance, this simplifies the job of a programmer who uses the class, let's see how it complicates the job of a programmer who uses the class.

Once you label an instance variable as `private`, there is then no way to change its value (nor to reference the instance variable in any other way), except by using one of the methods belonging to the class. Note that even when an instance variable is private, you can still access it through methods of the class. For the class `DateSecondTry`, you can change the values of the instance variables with the method `readInput` and you can obtain the values of the instance variables with the methods whose names start with `get`. So, the qualifier `private` does not make it impossible to access the instance variables. It just makes it illegal to use their names, which can be a minor nuisance.

The modifiers `public` and `private` before a method definition have a similar meaning. If the method is labeled `public`, there are no restrictions on its usage. If the method is labeled `private`, the method can only be used in the definition of another method of the same class.

Any instance variable can be labeled either `public` or `private`. Any method can be `public` or `private`. However, normal good programming practices require that *all* instance variables be private and typically most methods be public. Normally, a method is private only if it is being used solely as a helping method in the definition of other methods.

Example

YET ANOTHER DATE CLASS

Display 4.7 contains another, much improved, definition of a class for a date. Note that all instance variables are private and that two methods are private. We made the methods `dateOK` and `monthString` private because they are just helping methods used in the definitions of other methods. A user of the class `DateFifthTry` would not (in fact, cannot) use either of the methods `dateOK` or `monthString`. This is all hidden information that need not concern a programmer

using the class. The method `monthString` was public in previous versions of our date classes because we had not yet discussed the `private` modifier. It is now marked `private` because it is just a helping method.

Note that the class `DateFifthTry` uses the method `dateOK` to make sure that any changes to instance variables make sense. You cannot use any methods, such as `readInput` or `setDate`, to set the instance variables so that they represent an impossible date like January 63, 2005. If you try to do so, your program would end with an error message. (To make our definition of the method `dateOK` simple, we did not check for certain impossible dates, such as February 31, but it would be easy to exclude these dates as well.)

The methods `dateOK` and `equals` each return a value of type `boolean`. That means they return a value that is either `true` or `false` and so can be used as the Boolean expression in an `if-else` statement, `while` statement, or other loop statement. This is illustrated by the following, which is taken from the definition of the method `setDate` in Display 4.7:

```
if (dateOK(month, day, year))
{
    this.month = monthString(month);
    this.day = day;
    this.year = year;
}
else
{
    System.out.println("Fatal Error");
    System.exit(0);
}
```

Note that, although all the instance variables are private, a programmer using the class can still change or access the value of an instance variable using the methods that start with `set` or `get`. This is discussed more fully in the next subsection, "Accessor and Mutator Methods."

Note that there is a difference between what we might call the *inside view* and the *outside view* of the class `DateFifthTry`. A date like July 4, 1776 is represented inside the class object as the string value `"Jul"` and the two `int` values `4` and `1776`. But, if a programmer using the same class object asks for the date using `getMonth`, `getDay`, and `getYear`, he or she will get the three `int` values `7`, `4`, and `1776`. From inside the class, a month is a string value, but from outside the class, a month is an integer. The description of the data in a class object need not be a simple direct description of the instance variables.

Note that the method definitions in a class need not be given in any particular order. In particular, it is perfectly acceptable to give the definition the method `dateOK` after the definitions of methods that use `dateOK`. Indeed, any ordering of the method definitions is acceptable. Use whatever order seems to make the class easiest to read. (Those who come to Java from certain other programming languages should note that there is no kind of forward reference needed when a method is used before it is defined.)

Display 4.7 Yet Another Date Class *(Part 1 of 4)*

```
1   import java.io.BufferedReader;
2   import java.io.InputStreamReader;
3   import java.io.IOException;

4   public class DateFifthTry
5   {
6       private String month; //always 3 letters long, as in Jan, Feb, etc.
7       private int day;
8       private int year; //a four digit number.


9       public void writeOutput()
10      {
11          System.out.println(month + " " + day + ", " + year);
12      }

13      public void readInput() throws IOException
14      {
15          boolean tryAgain = true;
16          BufferedReader keyboard = new BufferedReader(
17                                     new InputStreamReader(System.in));
18          while (tryAgain)
19          {
20              System.out.println(
21                          "Enter month, day, and year on three lines.");
22              System.out.println(
23                          "Enter month, day, and year as three integers.");

24              int monthInput = Integer.parseInt(keyboard.readLine());
25              int dayInput = Integer.parseInt(keyboard.readLine());
26              int yearInput = Integer.parseInt(keyboard.readLine());
27              if (dateOK(monthInput, dayInput, yearInput) )
28              {
29                  setDate(monthInput, dayInput, yearInput);
30                  tryAgain = false;
31              }
32              else
33                  System.out.println("Illegal date. Reenter input.");
34           }
35      }
36      public void setDate(int month, int day, int year)
37      {
38          if (dateOK(month, day, year))
39          {
40              this.month = monthString(month);
41              this.day = day;
42              this.year = year;
43          }
```

Note that this version of readInput checks to see that the input is reasonable.

Display 4.7 **Yet Another Date Class** ***(Part 2 of 4)***

```
44          else
45          {
46              System.out.println("Fatal Error");
47              System.exit(0);
48          }
49      }

50      public void setMonth(int monthNumber)
51      {
52          if ((monthNumber <= 0) || (monthNumber > 12))
53          {
54              System.out.println("Fatal Error");
55              System.exit(0);
56          }
57          else
58              month = monthString(monthNumber);
59      }

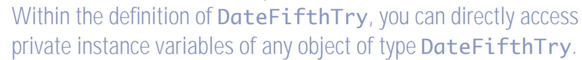
60      public void setDay(int day)
61      {
62          if ((day <= 0) || (day > 31))
63          {
64              System.out.println("Fatal Error");
65              System.exit(0);
66          }
67          else
68              this.day = day;
69      }
70      public void setYear(int year)
71      {
72          if ( (year < 1000) || (year > 9999) )
73          {
74              System.out.println("Fatal Error");
75              System.exit(0);
76          }
77          else
78              this.year = year;
79      }

80      public boolean equals(DateFifthTry otherDate)
81      {
82          return ( (month.equals(otherDate.month))
83                    && (day == otherDate.day) && (year == otherDate.year) );
84      }
```

Within the definition of `DateFifthTry`, you can directly access private instance variables of any object of type `DateFifthTry`.

Display 4.7 Yet Another Date Class *(Part 3 of 4)*

```
85        public boolean precedes(DateFifthTry otherDate)
86        {
87            return ( (year < otherDate.year) ||
88               (year == otherDate.year && getMonth() < otherDate.getMonth()) ||
89               (year == otherDate.year && month.equals(otherDate.month)
90                                              && day < otherDate.day) );
91        }
```



<The definitions of the following methods are the same as in Display 4.2 and Display 4.5:
getMonth, getDay, getYear, and toString.>

```
92        private boolean dateOK(int monthInt, int dayInt, int yearInt)
93        {
94            return ( (monthInt >= 1) && (monthInt <= 12) &&
95                     (dayInt >= 1) && (dayInt <= 31) &&
96                     (yearInt >= 1000) && (yearInt <= 9999) );
97        }

98        private String monthString(int monthNumber)
99        {
100           switch (monthNumber)
101           {
102           case 1:
103               return "Jan";
104           case 2:
105               return "Feb";
106           case 3:
107               return "Mar";
108           case 4:
109               return "Apr";
110           case 5:
111               return "May";
112           case 6:
113               return "Jun";
114           case 7:
115               return "Jul";
116           case 8:
117               return "Aug";
118            case 9:
119               return "Sep";
120           case 10:
121               return "Oct";
122           case 11:
123               return "Nov";
```

Display 4.7 Yet Another Date Class ***(Part 4 of 4)***

```
124             case 12:
125                 return "Dec";
126             default:
127                 System.out.println("Fatal Error");
128                 System.exit(0);
129                 return "Error"; //to keep the compiler happy
130             }
131         }

132     }
```

Self-Test Exercises

16. Following the style guidelines given in this book, when should an instance variable be marked `private`?

17. Following the style guidelines given in this book, when should a method be marked `pri-vate`?

ACCESSOR AND MUTATOR METHODS

You should always make all instance variables in a class private. But, you may sometimes need to do something with the data in a class object. The special-purpose methods, such as `toString`, `equals`, and any input methods, will allow you to do many things with the data in an object. But, sooner or later you will want to do something with the data for which there are no special-purpose methods. How can you do anything new with the data in an object? The answer is that you can do anything that you might reasonably want (and that the class design specifications consider to be legitimate), provided you equip your classes with suitable *accessor* and *mutator* methods. These are methods that allow you to access and change the data in an object, usually in a very general way. **Accessor methods** allow you to obtain the data. In Display 4.7, the methods `getMonth`, `getDay`, and `getYear` are accessor methods. The accessor methods need not literally return the values of each instance variable, but they must return something equivalent to those values. For example, the method `getMonth` returns the number of the month, even though the month is stored in a `String` instance variable. Although it is not required by the Java language, it is a generally accepted good programming practice to spell the names of accessor methods starting with `get`.

accessor methods

Mutator methods allow you to change the data in a class object. In Display 4.7, the methods whose names begin with the word `set` are mutator methods. It is a generally accepted good programming practice to use names that begin with the word `set` for mutator methods. Your class definitions will typically provide a complete set of public

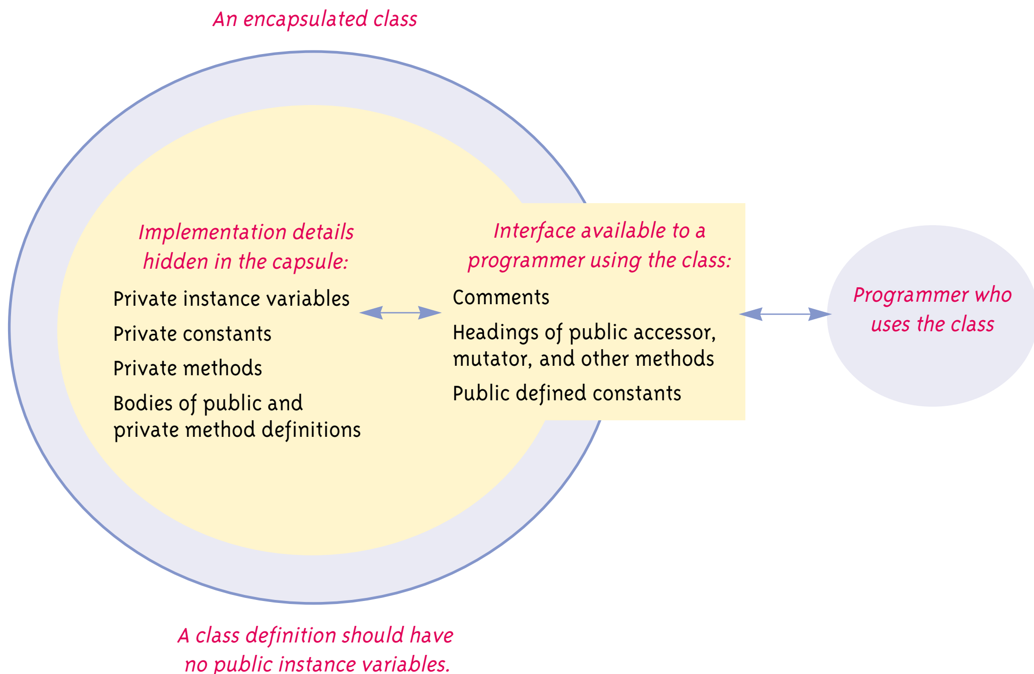
mutator methods

accessor methods and typically at least some public mutator methods. There are, however, important classes, such as the class `String`, that have no public mutator methods.

At first glance, it may look as if accessor and mutator methods defeat the purpose of making instance variables private, but if you look carefully at the mutator methods in Display 4.7, you will see that the mutator and accessor methods are not equivalent to making the instance variables public. Notice the mutator methods, that is, the ones that begin with `set`. They all test for an illegal date and end the program with an error message if there is an attempt to set the instance variables to any illegal values. If the variables were public you could set the data to values that do not make sense for a date, such as January 42, 1930. With mutator methods, you can control and filter changes to the data. (As it is, you can still set the data to values that do not represent a real date, such as February 31, but as we already noted, it would be easy to exclude these dates as well. We did not exclude these dates to keep the example simple. See Self-Test Exercise 20 for a more complete date check method.)

The way that a well-designed class definition uses private instance variables and public accessor and mutator methods to implement the principle of encapsulation is diagrammed in Display 4.8.

Display 4.8 Encapsulation



Tip

A Class Has Access to Private Members of All Objects of the Class

Consider the definition of the method `equals` for the class `DateFifthTry`, given in Display 4.7 and repeated below:

```
public boolean equals(DateFifthTry otherDate)
{
    return ( (month.equals(otherDate.month))
               && (day == otherDate.day) && (year == otherDate.year) );
}
```

You might object that `otherDate.month`, `otherDate.day`, and `otherDate.year` are illegal since `month`, `day`, and `year` are private instance variables of some object other than the calling object. Normally that objection would be correct. However, the object `otherDate` is of the same type as the class being defined, so this is legal. In the definition of a class, you can access private members of any object of the class, not just private members of the calling object.

Similar remarks apply to the method `precedes` in the same class. In one place in the definition of `precedes` we used `otherDate.getMonth()` rather than `otherDate.month` only because we wanted the month as an integer instead of a string. We did, in fact, use `otherDate.month` elsewhere in the definition of `precedes`.

PRECONDITIONS AND POSTCONDITIONS

One good way to write a method comment is to break it down into two kinds of information, called the *precondition* and the *postcondition.* The **precondition** states what is assumed to be true when the method is called. The method should not be used and cannot be expected to perform correctly unless the precondition holds. The **postcondition** describes the effect of the method call; that is, the postcondition tells what will be true after the method is executed in a situation in which the precondition holds. For a method that returns a value, the postcondition will describe the value returned by the method.

precondition

postcondition

For example, the following is an example of a method heading from Display 4.7 with a precondition and postcondition added:

```
/**
 Precondition: All instance variables of the calling object have values.
 Postcondition: The data in the calling object has been written to the screen.
*/
public void writeOutput()
```

You do not need to know the definition of the method `writeOutput` to use this method. All that you need to know to use this method is given by the precondition and

postcondition. (The importance of this is more dramatic when the definition of the method is longer than that of writeOutput.)

When the only postcondition is a description of the value returned, programmers usually omit the word Postcondition, as in the following example:

```
/**
 Precondition: All instance variables of the calling object have values.
 Returns a string describing the data in the calling object.
*/
public String toString()
```

Some programmers choose not to use the words *precondition* and *postcondition* in their method comments. However, whether you use the words or not, you should always think in terms of precondition and postcondition when designing a method and when deciding what to include in the method comment.

Self-Test Exercises

18. List all the accessor methods in the class DateFifthTry in Display 4.7.

19. List all the mutator methods in the class DateFifthTry in Display 4.7.

20. Write a better version of the method dateOK with three int parameters (Display 4.7). This better version checks for the correct number of days in each month and does not just allow 31 days in any month. It will help to define another helping method named leapYear, which takes an int argument for a year and returns true if the year is a leap year. February has 29 days in leap years and only 28 days in other years. Use the following rule for determining if the year is a leap year: A year is a leap year if it is divisible by 4 but is not divisible by 100 or if it is divisible by 400.

4.3 Overloading

A good name is better than precious ointment...

Ecclesiastes 7:1

Two (or more) different classes can have methods with the same name. For example, many classes have a method named toString. It is easy to see why this is acceptable. The type of the calling object allows Java to decide which definition of the method toString to use. It uses the definition of toString given in the definition of the class for the calling object. You may be more surprised to learn that two or more methods *in the same class* can have the same method name. This is called **overloading** and is the topic of this section.

overloading

RULES FOR OVERLOADING

In Display 4.9 we have added two methods named `setDate` to our date class so that there is a total of three methods named `setDate`. This is an example of overloading the method name `setDate`. On the following three lines we display the headings of these three methods:

```
public void setDate(int month, int day, int year)
public void setDate(String month, int day, int year)
public void setDate(int year)
```

Notice that each method has a different parameter list. The first two differ in the type of their first parameter. The last one differs from the other two by having a different number of parameters.

The name of a method and the list of parameter types in the heading of the method definition is called the **method signature**. The signatures for these three method definitions are

method signature

```
setDate(int, int, int)
setDate(String, int, int)
setDate(int)
```

When you overload a method name, each of the method definitions in the class must have a different **signature**.

SIGNATURE

The **signature** of a method consists of the method name and the list of types for parameters that are listed in the heading of the method name.

EXAMPLE:

If a method has the heading

```
public int computeSomething(int n1, double x1,
                                  double x2, String name);
```

then the signature is

```
computeSomething(int, double, double, String)
```

Note that the return type is not part of the method signature.

Display 4.9 Overloading Method Names (*Part 1 of 2*)

```
1   import java.io.BufferedReader;
2   import java.io.InputStreamReader;
3   import java.io.IOException;

4   public class DateSixthTry
5   {
6       private String month; //always 3 letters long, as in Jan, Feb, etc.
7       private int day;
8       private int year; //a four digit number.

9       public void setDate(int monthInt, int day, int year)
10      {
11          if (dateOK(monthInt, day, year))
12          {
13              this.month = monthString(monthInt);
14              this.day = day;
15              this.year = year;
16          }
17          else
18          {
19              System.out.println("Fatal Error");
20              System.exit(0);
21          }
22      }

23      public void setDate(String monthString, int day, int year)
24      {
25          if (dateOK(monthString, day, year))
26          {
27              this.month = monthString;
28              this.day = day;
29              this.year = year;
30          }
31          else
32          {
33              System.out.println("Fatal Error");
34              System.exit(0);
35          }
36      }
```

There are three different methods named `setDate`. (One is in Part 2 of this Display.)

In Display 4.9 we also overloaded the method name `dateOK` so that there are two different methods named `dateOK`. The two signatures for the two methods named `dateOK` are

```
dateOK(int, int, int)
dateOK(String, int, int)
```

Display 4.9 Overloading Method Names *(Part 2 of 2)*

```
37      public void setDate(int year)
38      {
39          setDate(1, 1, year);
40      }

41      private boolean dateOK(int monthInt, int dayInt, int yearInt)
42      {
43          return ( (monthInt >= 1) && (monthInt <= 12) &&
44                   (dayInt >= 1) && (dayInt <= 31) &&
45                   (yearInt >= 1000) && (yearInt <= 9999) );
46      }

47      private boolean dateOK(String monthString, int dayInt, int yearInt)
48      {
49          return ( monthOK(monthString) &&
50                   (dayInt >= 1) && (dayInt <= 31) &&
51                   (yearInt >= 1000) && (yearInt <= 9999) );
52      }

53      private boolean monthOK(String month)
54      {
55          return (month.equals("Jan") || month.equals("Feb") ||
56                  month.equals("Mar") || month.equals("Apr") ||
57                  month.equals("May") || month.equals("Jun") ||
58                  month.equals("Jul") || month.equals("Aug") ||
59                  month.equals("Sep") || month.equals("Oct") ||
60                  month.equals("Nov") || month.equals("Dec") );
61      }

<The rest of the methods are the same as in Display 4.7, except that
        the parameter to equals and precedes is, of course, of type DateSixthTry.>

62  }
```

Two different methods named `setDate`

Two different methods named `dateOK`

Overloading

Within one class, you can have two (or more) definitions of a single method name. This is called **overloading** the method name. When you overload a method name, any two definitions of the method name must have different signatures; that is, any two definitions of the method name must either have different numbers of parameters or some parameter position must be of differing types in the two definitions.

Display 4.10 gives a simple example of a program using the overloaded method name `setDate`. Note that for each invocation of a method named `setDate`, only one of the definitions of `setDate` has a signature that matches the types of the arguments.

Pitfall

Overloading and Automatic Type Conversion

Automatic type conversion of arguments (such as converting an `int` to a `double` when the parameter is of type `double`) and overloading can sometimes interact in unfortunate ways. So, you need to know how these two things interact.

For example, consider the following method that might be added to the class `DateSixthTry` in Display 4.9:

```
public void increase(double factor)
{
    year = (int)(year + factor*year);
}
```

Display 4.10 Using an Overloaded Method Name

```
1   public class OverloadingDemo
2   {
3       public static void main(String[] args)
4       {
5           DateSixthTry date1 = new DateSixthTry(),
6                        date2 = new DateSixthTry(),
7                        date3 = new DateSixthTry();
8           date1.setDate(1, 2, 2007);
9           date2.setDate("Feb", 2, 2007);
10          date3.setDate(2007);
11          System.out.println(date1);
12          System.out.println(date2);
13          System.out.println(date3);
14      }
15  }
```

Sample Dialogue

```
Jan 2, 2007
Feb 2, 2007
Jan 1, 2007
```

If you add this method to the class `DateSixthTry`, then the following presents no problems, where `date` is an object of type `DateSixthTry` that has been set to some date:

```
date.increase(2);
```

The `int` value of 2 is type cast to the `double` value `2.0` and the value of `date.year` is changed as follows:

```
date.year = (int)(date.year + 2.0*date.year);
```

(Since `year` is private in the class `DateSixthTry`, you cannot write this in a program that uses the class `DateSixthTry`, but the meaning of this expression is clear.)

So far, so good. But, now suppose we also add the following method definition to the class `DateSixthTry`:

```
public void increase(int term)
{
    year = year + term;
}
```

This is a valid overloading because the two methods named `increase` take parameters of different types.

With both of these methods named `increase` added to the class, the following now behaves differently:

```
date.increase(2);
```

If Java can find an exact match of types, it will use the method definition with an exact match before it tries to do any automatic type casts. So now, the displayed invocation of `date.increase` is equivalent to

```
date.year = date.year + 2;
```

However, if you meant to use an argument of `2.0` for `date.increase` and instead used `2`, counting on an automatic type cast, then this is not what you want.

Overloading and Automatic Type Conversion

Java always looks for a method signature that exactly matches the method invocation before it tries to use automatic type conversion. If Java can find a definition of a method that exactly matches the types of the arguments, it will use that definition. Only after it fails to find an exact match will Java try automatic type conversions to find a method definition that matches the (type cast) types of the method invocation.

It is best to avoid overloading where there is a potential for interacting dangerously with automatic type casting, as in the examples discussed in this Pitfall section.

In some cases of overloading, a single method invocation can be resolved in two different ways, depending on how overloading and type conversion interact. Such ambiguous method invocations are not allowed in Java and will produce an error message. For example, you can overload a method named `doSomething` by giving two definitions that have the following two method headings in a `SampleClass`:

```
public class SampleClass
{
    public void doSomething(double n1, int n2)
        .
        .
        .
    public void doSomething(int n1, double n2)
        .
        .
        .
```

Such overloading is legal, but there is a problem. Suppose `aSampleObject` is an object of type `SampleClass`. An invocation such as the following will produce an error message, because Java cannot decide which overloaded definition of `doSomething` to use:

```
aSampleObject.doSomething(5, 10);
```

Java cannot decide whether it should convert the `int` value `5` to a `double` value and use the first definition of `doSomething`, or convert the `int` value `10` to a `double` value and use the second definition. In this situation, the Java compiler issues an error message indicating that the method invocation is ambiguous.

The following two method invocations are allowed:

```
aSampleObject.doSomething(5.0, 10);
aSampleObject.doSomething(5, 10.0);
```

However, such situations, while legal, are confusing and should be avoided.

Pitfall

You Cannot Overload Based on the Type Returned

Note that the signature of a method lists only the method name and the types of the parameters and does not include the type returned. When you overload a method name, any two methods must have different signatures. The type returned has nothing to do with the signature of a method. For example, a class could not have two method definitions with the following headings:

```
public class SampleClass2
{
```

```
public int computeSomething(int n)
    .
    .
    .
public double computeSomething(int n)
    .
    .
    .
```

If you think about it, there is no way that Java could allow this sort of overloading. Suppose `anObject` is an object of the class `SampleClass2`, then in the following assignment, Java could not decide which of the above two method definitions to use:

```
double answer = anObject.computeSomething(10);
```

Either a value of type `int` or a value of type `double` can legally be assigned to the variable `answer`. So, either method definition could be used. Because of such problems, Java says it is illegal to have both of these method headings in the same class.

Self-Test Exercises

21. What is the signature of each of the following method headings?

```
public void doSomething(int p1, char p2, int p3)
public void setMonth(int newMonth)
public void setMonth(String newMonth)
public int amount(int balance, double duration)
public double amount(int balance, double duration)
```

22. Consider the class `DateSixthTry` in Display 4.9. Would it be legal to add two method definitions with the following two method headings to the class `DateSixthTry`?

```
public void setMonth(int newMonth)
public void setMonth(String newMonth)
```

23. Consider the class `DateSixthTry` in Display 4.9. Would it be legal to add two method definitions with the following two method headings to the class `DateSixthTry`?

```
public void setMonth(int newMonth)
private void setMonth(int newMonth)
```

24. Consider the class `DateSixthTry` in Display 4.9. Would it be legal to add two method definitions with the following two method headings to the class `DateSixthTry`?

```
public int getMonth()
public String getMonth()
```

YOU CANNOT OVERLOAD OPERATORS IN JAVA

Many programming languages, such as C++, allow you to overload operators, such as +, so that the operator can be used with objects of some class you define, as well as being used for such things as numbers. You cannot do this in Java. If you want to have an "addition" in your class, you must use a method name, such as add, and ordinary method syntax; you cannot define operators, such as the + operator, to work with objects of a class you define .

4.4 Constructors

Well begun is half done.

Proverb

You often want to initialize the instance variables for an object when you create the object. As we will see later in this book, there are other initializing actions you might also want to take, but initializing instance variables is the most common sort of initialization. A *constructor* is a special variety of method that is designed to perform such initialization. In this section, we tell you how to define and use constructors.

CONSTRUCTOR DEFINITIONS

Although you may not have realized it, you have already been using constructors every time you used the `new` operator to create an object, as in the following example:

```
DateSixthTry date1 = new DateSixthTry();
```

constructor

The expression `new DateSixthTry()` is an invocation of a constructor. A **constructor** is a special variety of method that, among other things, must have the same name as the class. So, the first occurrence of `DateSixthTry` in the above code is a class name and the second occurrence of `DateSixthTry` is the name of a constructor. If you add no constructor definitions to your class, then Java automatically creates a constructor that takes no arguments. We have been using this automatically provided constructor up until now. The automatically provided constructor creates the object but does little else. It is preferable to define your own constructors so that you can have the constructor initialize instance variables as you want or do whatever other initialization actions you want.

In Display 4.11 we have rewritten our date class one last time by adding five constructors. Since this is our final date class, we have included all method definitions in the display so you can see the entire class definition. (We have omitted `writeOutput` because it would be superfluous, as noted in the earlier subsection entitled "The Methods `equals` and `toString`.")

Display 4.11 A Class with Constructors *(Part 1 of 5)*

This is our final definition of a class whose objects are dates.

```
1   import java.io.BufferedReader;
2   import java.io.InputStreamReader;
3   import java.io.IOException;

4   public class Date
5   {
6       private String month; //always 3 letters long, as in Jan, Feb, etc.
7       private int day;
8       private int year; //a four digit number.

9       public Date()
10      {
11          month = "Jan";
12          day = 1;
13          year = 1000;
14      }

15      public Date(int monthInt, int day, int year)
16      {
17          setDate(monthInt, day, year);
18      }

19      public Date(String monthString, int day, int year)
20      {
21          setDate(monthString, day, year);
22      }

23      public Date(int year)
24      {
25          setDate(1, 1, year);
26      }

27      public Date(Date aDate)
28      {
29          if (aDate == null)//Not a real date.
30          {
31              System.out.println("Fatal Error.");
32              System.exit(0);
33          }
```

No-argument constructor

You can invoke another method inside a constructor definition.

A constructor usually initializes all instance variables, even if there is not a corresponding parameter.

We will have more to say about this constructor in Chapter 5. Although you have had enough material to use this constructor, you need not worry about it until Section 5.3 of Chapter 5.

<Definition of this constructor continues in Part 2.>

```
34          month = aDate.month;
35          day = aDate.day;
36          year = aDate.year;
37      }
```

Display 4.11 A Class with Constructors *(Part 2 of 5)*

```
38      public void setDate(int monthInt, int day, int year)
39      {
40          if (dateOK(monthInt, day, year))
41          {
42              this.month = monthString(monthInt);
43              this.day = day;
44              this.year = year;
45          }
46          else
47          {
48              System.out.println("Fatal Error");
49              System.exit(0);
50          }
51      }

52      public void setDate(String monthString, int day, int year)
53      {
54          if (dateOK(monthString, day, year))
55          {
56              this.month = monthString;
57              this.day = day;
58              this.year = year;
59          }
60          else
61          {
62              System.out.println("Fatal Error");
63              System.exit(0);
64          }
65      }

66      public void setDate(int year)
67      {
68          setDate(1, 1, year);
69      }
70      public void setYear(int year)
71      {
72          if ( (year < 1000) || (year > 9999) )
73          {
74              System.out.println("Fatal Error");
75              System.exit(0);
76          }
77          else
78              this.year = year;
79      }
```

The mutator methods, whose names begin with `set`*, are used to reset the data in an object after the object has been created using* `new` *and a constructor.*

Display 4.11 **A Class with Constructors** ***(Part 3 of 5)***

```
80      public void setMonth(int monthNumber)
81      {
82          if ((monthNumber <= 0) || (monthNumber > 12))
83          {
84              System.out.println("Fatal Error");
85              System.exit(0);
86          }
87          else
88              month = monthString(monthNumber);
89      }

90      public void setDay(int day)
91      {
92          if ((day <= 0) || (day > 31))
93          {
94              System.out.println("Fatal Error");
95              System.exit(0);
96          }
97          else
98              this.day = day;
99      }

100     public int getMonth()
101     {
102         if (month.equals("Jan"))
103             return 1;
104         else if (month.equals("Feb"))
105             return 2;
106         else if (month.equals("Mar"))
107             return 3;
              . . .
```

<The omitted cases are obvious, but if need be, you can see all the cases in Display 4.2.>

```
              . . .
108         else if (month.equals("Oct"))
109             return 10;
110         else if (month.equals("Nov"))
111             return 11;
112         else if (month.equals("Dec"))
113             return 12;
114         else
115         {
116             System.out.println("Fatal Error");
117             System.exit(0);
118             return 0; //Needed to keep the compiler happy
119         }
120     }
```

Display 4.11 A Class with Constructors *(Part 4 of 5)*

```
121        public int getDay()
122        {
123            return day;
124        }

125        public int getYear()
126        {
127            return year;
128        }

129        public String toString()
130        {
131            return (month + " " + day + ", " + year);
132        }
133        public boolean equals(Date otherDate)
134        {
135            return ( (month.equals(otherDate.month))
136                      && (day == otherDate.day) && (year == otherDate.year) );
137        }

138        public boolean precedes(Date otherDate)
139        {
140            return ( (year < otherDate.year) ||
141               (year == otherDate.year && getMonth() < otherDate.getMonth()) ||
142               (year == otherDate.year && month.equals(otherDate.month)
143                                             && day < otherDate.day) );
144        }

145        public void readInput() throws IOException
146        {
147            boolean tryAgain = true;
148            BufferedReader keyboard = new BufferedReader(
149                                           new InputStreamReader(System.in));
150            while (tryAgain)
151            {
152                System.out.println(
153                            "Enter month, day, and year on three lines.");
154                System.out.println(
155                            "Enter month, day, and year as three integers.");

156                int monthInput = Integer.parseInt(keyboard.readLine());
157                int dayInput = Integer.parseInt(keyboard.readLine());
158                int yearInput = Integer.parseInt(keyboard.readLine());
159                if (dateOK(monthInput, dayInput, yearInput) )
160                {
```

We have omitted the method `writeOutput` because it would be superfluous, as noted in the subsection entitled "The Methods `equals` and `toString`."

The method `equals` of the class `String`

Display 4.11 A Class with Constructors *(Part 5 of 5)*

```
161                     setDate(monthInput, dayInput, yearInput);
162                    tryAgain = false;
163                }
164                else
165                    System.out.println("Illegal date. Reenter input.");
166             }
167        }

168        private boolean dateOK(int monthInt, int dayInt, int yearInt)
169        {
170            return ( (monthInt >= 1) && (monthInt <= 12) &&
171                     (dayInt >= 1) && (dayInt <= 31) &&
172                     (yearInt >= 1000) && (yearInt <= 9999) );
173        }

174        private boolean dateOK(String monthString, int dayInt, int yearInt)
175        {
176            return ( monthOK(monthString) &&
177                     (dayInt >= 1) && (dayInt <= 31) &&
178                     (yearInt >= 1000) && (yearInt <= 9999) );
179        }

180        private boolean monthOK(String month)
181        {
182            return (month.equals("Jan") || month.equals("Feb") ||
183                    month.equals("Mar") || month.equals("Apr") ||
184                    month.equals("May") || month.equals("Jun") ||
185                    month.equals("Jul") || month.equals("Aug") ||
186                    month.equals("Sep") || month.equals("Oct") ||
187                    month.equals("Nov") || month.equals("Dec") );
188        }

189        private String monthString(int monthNumber)
190        {
191            switch (monthNumber)
192            {
193            case 1:
194                return "Jan";
               . . .
```

<The omitted cases are obvious, but if need be, you can see all the cases in Display 4.7.>

```
                . . .
195            default:
196                System.out.println("Fatal Error");
197                System.exit(0);
198                return "Error"; //to keep the compiler happy
199            }
200        }
201    }
```

The private methods need not be last, but that's as good a place as any.

In Display 4.11 we have used overloading to create five constructors for the class `Date`. It is normal to have more than one constructor. Since every constructor must have the same name as the class, all the constructors in a class must have the same name. So, when you define multiple constructors, you must use overloading.

Note that when you define a constructor, you do not give any return type for the constructor; you do not even use `void` in place of a return type. Also notice that constructors are normally public.

All the constructor definitions in Display 4.11 initialize all the instance variables, even if there is no parameter corresponding to that instance variable. This is normal. In a constructor definition you can do pretty much anything that you can do in any ordinary method definition, but normally you only do initialization tasks like initialization of instance variables.

constructor arguments

When you create a new object with the operator `new`, you must always include the name of a constructor after the operator `new`. This is the way you invoke a constructor. As with any method invocation, you list any arguments in parentheses after the constructor name (which is the same as the class name). For example, suppose you want to use `new` to create a new object of the class `Date` defined in Display 4.11. You might do so as follows:

```
Date birthday = new Date("Dec", 16, 1770);
```

This is a call to the constructor for the class `Date` that takes three arguments: one of type `String` and two of type `int`. This creates a new object to represent the date December 16, 1770 and sets the variable `birthday` so that it names this new object. Another example is the following:

```
Date newYearsDay = new Date(3000);
```

This creates a new object to represent the date January 1, 3000 and sets the variable `newYearsDay` so that it names this new object.

resetting object values

A constructor can be called only when you create a new object with the operator `new`. An attempt to call a constructor in any other way, such as the following, is illegal:

```
birthday.Date("Jan", 27, 1756); //Illegal!
```

Since you cannot call a constructor for an object after it is created, you need some other way to change the values of the instance variables of an object. That is the purpose of the `setDate` methods and other methods that begin with `set` in Display 4.11. If `birthday` already names an object that was created with `new`, you can change the values of the instance variables as follows:

```
birthday.setDate("Jan", 27, 1756);
```

Although it is not required, such methods that reset instance variables normally are given names that start with `set`.

Constructor

A **constructor** is a variety of method that is called when an object of the class is created using new. Constructors are used to initialize objects. A constructor must have the same name as the class to which it belongs. Arguments for a constructor are given in parentheses after the class name, as in the following examples:

Examples:

```
Date birthday = new Date("Dec", 16, 1770),
     theDate = new Date(2008);
```

A constructor is defined very much like any ordinary method except that it does not have a type returned and does not even include a void in the constructor heading. See Display 4.11 for examples of constructor definitions.

Although you cannot use a constructor to reset the instance variables of an already created object, you can do something that looks very similar to that. The following is legal:

```
Date birthday = new Date("Dec", 16, 1770);
                    .
                    .
                    .
birthday = new Date("Jan", 27, 1756);
```

However, the second invocation of the constructor does not simply change the values of instance variables for the object. Instead, it discards the old object and allocates storage for a new object before setting the instance variables. So, for efficiency (and occasionally for other reasons we have not yet discussed) it is preferable to use a method like setDate to change the data in the instance variables of an already created object.

Display 4.12 contains a demonstration program for the constructors defined in Display 4.11.

Is a Constructor Really a Method?

There are differing opinions on whether or not a constructor should be called a *method.* Most authorities call a constructor a method but emphasize that it is a very special kind of method with many properties not shared with other kinds of methods. Some authorities say a constructor is a method-like entity but not, strictly speaking, a method. All authorities agree about what a constructor is; the only disagreement is over whether or not it should be referred to as a *method.* Thus, this is not a major issue. However, whenever you hear a phrase like "all methods" you should make sure it does or does not include constructors. To avoid confusion we try to use the phrase "constructors and methods" when we want to include constructors.

Display 4.12 Use of Constructors

```
1   public class ConstructorsDemo
2   {
3       public static void main(String[] args)
4       {
5           Date date1 = new Date("Dec", 16, 1770),
6                date2 = new Date(1, 27, 1756),
7                date3 = new Date(1882),
8                date4 = new Date();

9           System.out.println("Whose birthday is " + date1 + "?");
10          System.out.println("Whose birthday is " + date2 + "?");
11          System.out.println("Whose birthday is " + date3 + "?");
12          System.out.println("The default date is " + date4 + ".");
13      }
14  }
```

Sample Dialogue

```
Whose birthday is Dec 16, 1770?
Whose birthday is Jan 27, 1756?
Whose birthday is Jan 1, 1882?
The default date is Jan 1, 1000.
```

Tip

You Can Invoke Another Method in a Constructor

It is perfectly legal to invoke another method within the definition of a constructor. For example, several of the constructors in Display 4.11 invoke a mutator method to set the values of the instance variables. This is legal because the first action taken by a constructor is to (automatically) create an object with instance variables. You do not write any code to create this object. Java creates it automatically when the constructor is invoked. Any method invocation in the body of the constructor definition has this object as its calling object.

You can even include an invocation of one constructor within the definition of another constructor. However, we will not discuss the syntax for doing that in this chapter. It will be covered in Chapter 7.

Tip

Include a No-Argument Constructor

A constructor that takes no arguments is called a **no-argument constructor** or **no-arg constructor**. If you define a class and include absolutely no constructors of any kind, then a no-argument constructor will be automatically created. This no-argument constructor does not do much else but it does give you an object of the class type. So, if the definition of the class `MyClass` contains absolutely no constructor definitions, then the following is legal:

no-argument constructor

```
MyClass myObject = new MyClass();
```

If your class definition includes one or more constructors of any kind, then no constructor is generated automatically. So, for example, suppose you define a class called `YourClass`. If you include one or more constructors that each take one or more arguments, but you do not include a no-argument constructor in your class definition, then there is not a no-argument constructor and the following is illegal:

```
YourClass yourObject = new YourClass();
```

The problem with the above declaration is that it asks the compiler to invoke the no-argument constructor, but there is no no-argument constructor in this case.

To avoid problems, you should normally include a no-argument constructor in any class you define. If you do not want the no-argument constructor to initialize any instance variables, you can simply give it an empty body when you implement it. The following constructor definition is perfectly legal. It does nothing but create an object (and, as we will see later in this chapter, set the instance variables equal to default values):

```
public MyClass()
{/*Do nothing.*/}
```

A no-argument constructor is also known as a **default constructor**. However, the term *default constructor* is misleading since, as we have explained, a no-argument constructor is not always provided by default. There is now a movement to replace the term *default constructor* with the term *no-argument constructor,* but you will frequently encounter the term *default constructor.*

default constructor

Example

The Final Date Class

The final version of our class for a date is given in Display 4.11. We will be using this class `Date` again in Chapter 5.

No-Argument Constructor

A constructor with no parameters is called a **no-argument constructor**. If your class definition contains absolutely no constructor definitions, then Java will automatically create a no-argument constructor. If your class definition contains one or more constructor definitions, then Java does not automatically generate any constructor; in this case, what you define is what you get. Most of the classes you define should include a definition of a no-argument constructor.

Self-Test Exercises

25. If a class is named `CoolClass`, what names are allowed as names for constructors in the class `CoolClass`?

26. Suppose you have defined a class like the following for use in a program:

```
public class YourClass
{
    private int information;
    private char moreInformation;

    public YourClass(int newInfo, char moreNewInfo)
    {
            <Details not shown.>
    }
    public YourClass()
    {
            <Details not shown.>
    }
    public void doStuff()
    {
            <Details not shown.>
    }
}
```

Which of the following are legal in a program that uses this class?

```
YourClass anObject = new YourClass(42, 'A');
YourClass anotherObject = new YourClass(41.99, 'A');
YourClass yetAnotherObject = new YourClass();
yetAnotherObject.doStuff();
YourClass oneMoreObject;
oneMoreObject.doStuff();
oneMoreObject.YourClass(99, 'B');
```

27. What is a no-argument constructor? Does every class have a no-argument constructor? What is a default constructor?

DEFAULT VARIABLE INITIALIZATIONS

Local variables are not automatically initialized in Java, so you must explicitly initialize a local variable before using it. Instance variables, on the other hand, are automatically initialized. Instance variables of type `boolean` are automatically initialized to `false`. Instance variables of other primitive types are automatically initialized to the zero of their type. Instance variables of a class type are automatically initialized to `null`, which is a kind of placeholder for an object that will be filled in later. We will discuss `null` in Chapter 5. Although instance variables are automatically initialized, we prefer to always explicitly initialize them in a constructor, even if the initializing value is the same as the default initialization. That makes the code clearer.

AN ALTERNATIVE WAY TO INITIALIZE INSTANCE VARIABLES

Instance variables are normally initialized in constructors, and that is where we prefer to initialize them. However, there is an alternative. You can initialize instance variables when you declare them in a class definition, as illustrated by the following:

```
public class Date
{
    private String month = "Jan";
    private int day = 1;
    private int year = 1000;
```

If you initialize instance variables in this way, you may or may not want to define constructors. But, if you do define any constructors, it is usually best to define a no-argument constructor even if the body of the no-argument constructor is empty.

THE StringTokenizer CLASS

The `StringTokenizer` class is used to recover the words in a multi-word string. It is often used when reading input. However, when we covered input in Chapter 2 we could not cover the `StringTokenizer` class because its use normally involves knowledge of loops and constructors, two topics that we had not yet covered. We now have covered enough material to explain the `StringTokenizer` class.

When reading keyboard input either with `JOptionPane` or with `BufferedReader` and `readLine`, the input is always produced as a string value corresponding to a complete line of input. The class `StringTokenizer` can be used to decompose this string into words so that you can treat input as multiple items on a single line.

The class `StringTokenizer` is in the standard Java package (library) `java.util`. To tell Java where to find the class `StringTokenizer`, any class or program that uses the class `StringTokenizer` must contain the following (or something similar) at the start of the file:

```
import java.util.StringTokenizer;
```

`import`

Perhaps the most common use of the `StringTokenizer` class is to decompose a line of input. However, the `StringTokenizer` class can be used to decompose any string. The following example illustrates a typical way that the class `StringTokenizer` is used:

```
StringTokenizer wordFactory =
                new StringTokenizer("A single word can be critical.");
while (wordFactory.hasMoreTokens())
{
    System.out.println(wordFactory.nextToken());
}
```

This will produce the following output:

```
A
single
word
can
be
critical.
```

The constructor invocation

```
new StringTokenizer("A single word can be critical.")
```

produces a new object of the class `StringTokenizer`. The assignment statement

```
StringTokenizer wordFactory =
                new StringTokenizer("A single word can be critical.");
```

gives this `StringTokenizer` object the name `wordFactory`. You may use any string in place of `"A single word can be critical."` and any variable name in place of `word-Factory`. The `StringTokenizer` object created in this way can be used to produce the individual words in the string used as the argument to the `StringTokenizer` constructor. These individual words are called **tokens**.

tokens

nextToken

The method `nextToken` returns the first token (word) when it is invoked for the first time, returns the second token when it is invoked the second time, and so forth. If your code invokes `nextToken` after it has returned all the tokens in its string, then your program will halt and issue an error message.

hasMore-Tokens

The method `hasMoreTokens` is a method that returns a value of type `boolean`; that is, it returns either `true` or `false`. Thus, an invocation of `hasMoreTokens`, such as

```
wordFactory.hasMoreTokens()
```

is a Boolean expression, and so can be used to control a `while` loop. The method `hasMoreTokens` returns `true` as long as `nextToken` has not yet returned all the tokens in the string, and it returns `false` after the method `nextToken` has returned all the tokens in the string.

When the constructor for `StringTokenizer` is used with a single argument, as in the preceding example, the tokens are substrings of nonwhitespace characters, and the whitespace characters are used as the separators for the tokens. Any string of one or more whitespace characters is considered a separator. Thus, in the preceding example, the last token produced by the method `nextToken` is `"critical."`, including the period, because the period is not a whitespace character and so is not a separator.

choosing delimiters

You can specify your own set of separator characters. When you give your own set of separator characters, you give a second argument to the constructor for `StringTokenizer`. The second argument is a string consisting of all the separator characters. Thus, if you want your separators to consist of the blank, new-line character, period, and comma, you could proceed as in the following example:

```
StringTokenizer wordfactory2 =
        new StringTokenizer("Give me the word, my friend.", " \n.,");
while (wordfactory2.hasMoreTokens())
{
    System.out.println(wordfactory2.nextToken());
}
```

This will produce the output

```
Give
me
the
word
my
friend
```

Notice that the period and comma are not part of the tokens produced, because they are now token separators. Also note that the string of token separators is the second argument to the constructor.

Some of the methods for the class `StringTokenizer` are summarized in Display 4.13. A sample use of `StringTokenizer` is given in Display 4.14 .

Display 4.13 Some Methods in the Class StringTokenizer

The class `StringTokenizer` is in the `java.util` package.

```
public StringTokenizer(String theString)
```

Constructor for a tokenizer that will use whitespace characters as separators when finding tokens in `theString`.

```
public StringTokenizer(String theString, String delimiters)
```

Constructor for a tokenizer that will use the characters in the string `delimiters` as separators when finding tokens in `theString`.

```
public boolean hasMoreTokens()
```

Tests whether there are more tokens available from this tokenizer's string. When used in conjunction with `nextToken`, it returns `true` as long as `nextToken` has not yet returned all the tokens in the string; returns `false` otherwise.

```
public String nextToken()
```

Returns the next token from this tokenizer's string. (Throws `NoSuchElementException` if there are no more tokens to return.)[1]

```
public int countTokens()
```

Returns the number of tokens remaining to be returned by `nextToken`.

[1] Exceptions are covered in Chapter 9. You can ignore any reference to `NoSuchElementException` until you reach Chapter 9. We include it here for reference value only.

Self-Test Exercises

28. What would be the last line in the dialog in Display 4.14 if the user entered the following input line instead of the one shown in Display 4.14? (The comma is omitted.)

```
41.98  42
```

29. What would be the last line in the dialog in Display 4.14 if the user entered the following input line instead of the one shown in Display 4.14?

```
1, 2, 3, 4
```

30. What would be the last line in the dialog in Display 4.14 if the user entered the following input line instead of the one shown in Display 4.14?

```
1, 2, buckle my shoe.
```

31. What would be the last line in the dialog in Display 4.14 if the user entered the following input line instead of the one shown in Display 4.14?

```
one, two, buckle my shoe.
```

Display 4.14 Use of the StringTokenizer Class

```
1   import java.io.BufferedReader;
2   import java.io.InputStreamReader;
3   import java.io.IOException;
4   import java.util.StringTokenizer;

1   public class StringTokenizerDemo
2   {
3       public static void main(String[] args) throws IOException
4       {
5           BufferedReader keyboard = new BufferedReader(
6                                         new InputStreamReader(System.in));

7           System.out.println("Enter two numbers on a line.");
8           System.out.println("Place a comma between the numbers.");
9           System.out.println("Extra blank space is OK.");
10          String inputLine = keyboard.readLine();

11          String delimiters = ", "; //Comma and blank space
12          StringTokenizer numberFactory =
13                           new StringTokenizer(inputLine, delimiters);

14          double number1 = 0,
15                 number2 = 0; //Initialized to keep compiler happy
16          if (numberFactory.countTokens() >= 2)
17          {
18              number1 = Double.parseDouble(numberFactory.nextToken());
19              number2 = Double.parseDouble(numberFactory.nextToken());
20          }
21          else
22          {
23              System.out.println("Fatal Error.");
24              System.exit(0);
25          }
26          System.out.print("You input is ");
27          System.out.println(number1 + " and " + number2);
28      }
29  }
```

Sample Dialogue

```
Enter two numbers on a line.
Place a comma between the numbers.
Extra blank space is OK.
    41.98,  42
You input is 41.98 and 42.0
```

Note that the comma and space are delimiters, but the period is not.

Chapter Summary

- Objects have both instance variables and methods. A class is a type whose values are objects. All objects in a class have the same methods and the same types of instance variables.
- There are two main kinds of methods: methods that return a value and `void` methods. (Some specialized methods, such as constructors, are neither `void` methods nor methods that return a value.)
- When defining a method, the `this` parameter is a name used for the calling object.
- Normally, your classes should each have both an `equals` method and a `toString` method.
- If an instance variable or method is marked `private`, then it cannot be directly referenced anyplace except in the definition of a method of the same class.
- Outside of the class in which it is defined, a private instance variable can be accessed via accessor methods and changed via mutator methods.
- A variable declared in a method is said to be a *local variable*. The meaning of a local variable is confined to the method in which it is declared. The local variable goes away when a method invocation ends. The name of a local variable can be reused for something else outside of the method in which it is declared.
- A parameter is like a blank in a method definition that is filled in with an argument when the method is invoked. A parameter is actually a local variable that is initialized to the value of the corresponding argument. This is known as the *call-by-value* parameter-passing mechanism.
- If a variable is used as an argument to a method, then only the value of the variable, not the variable itself, is plugged in to the corresponding parameter.
- *Encapsulation* means that the data and the actions are combined into a single item (in our case, a class object) and that the *details of the implementation are hidden.* Making all instance variables private is part of the encapsulation process.
- A class can have two (or more) different definitions for the same method name, provided the two definitions have different numbers of parameters or some parameters of differing types. This is called *overloading* the method name.
- A constructor is a variety of method that is called when you create an object of the class using `new`. A constructor is intended to be used to perform initialization tasks such as initializing instance variables. A constructor must have the same name as the class to which it belongs.
- A constructor with no parameters is called a *no-argument constructor.* If your class definition includes no constructor definitions at all, then Java will automatically provide a no-argument constructor. If your class definition contains any constructor definitions at all, then no additional constructors are provided by Java. Your class definitions should usually include a no-argument constructor.
- The class `StringTokenizer` can be used to extract the tokens (words) from a string.

Answers to Self-Test Exercises

1.
```
public void makeItNewYears()
{
    month = "Jan";
    day = 1;
}
```

2.
```
public void yellIfNewYear()
{
    if ( (month == "Jan") && (day == 1) )
        System.out.println("Hurrah!");
    else
        System.out.println("Not New Year's Day.");
}
```

3.
```
public int getNextYear()
{
    return year++;
}
```

4. You need to add the phrase `throws IOException` to the first line of `echo2Lines`. The correct definition is

```
public void echo2Lines() throws IOException
{
    echoLine();
    echoLine():
}
```

5.
```
public void happyGreeting()
{
    int count;
    for (count = 1; count <= day; count++)
        System.out.println("Happy Days!");
}
```

6.
```
public double fractionDone(int targetDay)
{
    double doubleDay = day;
    return doubleDay/targetDay;
}
```

7.
```
public void advanceYear(int increase)
{
    year = year + increase;
}
```

8. The instances of `newMonth` that have their values changed to `6` are indicated in color below:

```
public void setDate(int newMonth, int newDay, int newYear)
{
    month = monthString(newMonth);
    day = newDay;
    year = newYear;
    System.out.println("Date changed to "
                  + newMonth + " " + newDay + ", " + newYear);
}
```

   The point being emphasized here is that all instances of `newMonth` have their values changed to `6`. Technically speaking, the parameter `newMonth` is a local variable. So, there is only one local variable named `newMonth` whose value is changed to `6`, but the net effect, in this case, is the same as replacing all occurrences of `newMonth` with `6`.

9. Yes, it is legal. The point being emphasized here is that the parameter `count` is a local variable and so can have its value changed, in this case by the decrement operator.

10. Each case has a `return` statement. A `return` statement always ends the method invocation, and hence ends the execution of the `switch` statement. So, a `break` statement would be redundant.

11. They are assumed to be instance variables of the calling object.

12.
```
public int getDay()
{
     return this.day;
}

public int getYear()
{
    return this.year;
}
```

13.
```
public int getMonth()
{
    if (this.month.equals("Jan"))
        return 1;
    else if (this.month.equals("Feb"))
        return 2;
    else if (this.month.equals("Mar"))
        return 3;
    else if (this.month.equals("Apr"))
        return 4;
    else if (this.month.equals("May"))
        return 5;
    else if (this.month.equals("Jun"))
        return 6;
    else if (this.month.equals("Jul"))
        return 7;
```

```
        else if (this.month.equals("Aug"))
            return 8;
        else if (this.month.equals("Sep"))
            return 9;
        else if (this.month.equals("Oct"))
            return 10;
        else if (this.month.equals("Nov"))
            return 11;
        else if (this.month.equals("Dec"))
            return 12;
        else
        {
            System.out.println("Fatal Error");
            System.exit(0);
            return 0; //Needed to keep the compiler happy
        }
    }
```

14. The instance variable `month` contains a string, so we used `month` with `equals`. It would have been just as good to use

    ```
    getMonth() == otherDate.getMonth()
    ```

    We used `getMonth()` with the less-than sign because it is of type `int` and so works with the less-than sign. The instance variable `month` is of type `String` and does not work with the less-than sign.

15. Every method should be tested in a program in which every other method in the testing program has already been fully tested and debugged.

16. All instance variables should be marked `private`.

17. Normally, a method is private only if it is being used solely as a helping method in the definition of other methods.

18. `getMonth`, `getDay`, and `getYear`.

19. `setDate`, `setMonth`, `setDay`, and `setYear`.

20.
```
    private boolean dateOK(int monthInt, int dayInt, int yearInt)
    {
        if ((yearInt < 1000) || (yearInt > 9999))
            return false;

        switch (monthInt)
        {
        case 1:
            return (dayInt >= 1) && (dayInt <= 31);
        case 2:
            if (leapYear(yearInt))
                return (dayInt >= 1) && (dayInt <= 29);
            else
```

```
                return (dayInt >= 1) && (dayInt <= 28);
        case 3:
            return (dayInt >= 1) && (dayInt <= 31);
        case 4:
            return (dayInt >= 1) && (dayInt <= 30);
        case 5:
            return (dayInt >= 1) && (dayInt <= 31);
        case 6:
            return (dayInt >= 1) && (dayInt <= 30);
        case 7:
            return (dayInt >= 1) && (dayInt <= 31);
        case 8:
            return (dayInt >= 1) && (dayInt <= 31);
        case 9:
            return (dayInt >= 1) && (dayInt <= 30);
        case 10:
            return (dayInt >= 1) && (dayInt <= 31);
        case 11:
            return (dayInt >= 1) && (dayInt <= 30);
        case 12:
            return (dayInt >= 1) && (dayInt <= 31);
        default:
            System.out.println("Fatal Error");
            System.exit(0);
            return false; //to keep the compiler happy
        }
    }

    /**
      Returns true if yearInt is a leap year.
    */
    private boolean leapYear(int yearInt)
    {
         return ((yearInt % 4 == 0) && (yearInt % 100 != 0))
                  || (yearInt % 400 == 0);
    }
```

21. ```
    doSomething(int, char, int)
    setMonth(int)
    setMonth(String)
    amount(int, double)
    amount(int, double)
    ```

22. Yes, it is legal because they have different signatures. This is a valid example of overloading.

23. No, it would be illegal because they have the same signature.

24. No, it would be illegal. You cannot overload on the basis of the type of the returned value.

25. If a class is named `CoolClass`, then all constructors must be named `CoolClass`.
    ```

26.
```
YourClass anObject = new YourClass(42, 'A');//Legal
YourClass anotherObject = new YourClass(41.99, 'A');//Not legal
YourClass yetAnotherObject = new YourClass();//Legal
yetAnotherObject.doStuff();//Legal
YourClass oneMoreObject;//Legal
oneMoreObject.doStuff();//Not legal
oneMoreObject.YourClass(99, 'B');//Not legal
```

27. A no-argument constructor is a constructor with no parameters. If you define a class and define some constructors but do not define a no-argument constructor, then the class will have no no-argument constructor. *Default constructor* is another name for no-argument constructor.

28. The last line would be the same. Since the blank space is a delimiter, a blank space is enough to separate the tokens `"41.98"` and `"42"`.

29. `You input 1.0 and 2.0`
    The extra tokens in the input line are just not used.

30. `You input 1.0 and 2.0`
    The extra tokens in the input line are not used, so it does not matter what they are.

31. The first two tokens are `"one"` and `"two"`, but the following line of the program ends the program with an error message:

```
number1 = Double.parseDouble(numberFactory.nextToken());
```

The token `"one"` cannot be converted to a value of type `double` by the method `Double.parseDouble`.

Programming Projects

1. Define a class called `Counter` whose objects count things. An object of this class records a count that is a nonnegative integer. Include methods to set the counter to 0, to increase the count by 1, and to decrease the count by 1. Be sure that no method allows the value of the counter to become negative. Include an accessor method that returns the current count value and a method that outputs the count to the screen. There will be no input method or other mutator methods. The only method that can set the counter is the one that sets it to zero. Also, include a `toString` method and an `equals` method. Write a program (or programs) to test all the methods in your class definition.

2. Write a grading program for a class with the following grading policies:

   a. There are three quizzes, each graded on the basis of 10 points.

   b. There is one midterm exam, graded on the basis of 100 points.

   c. There is one final exam, graded on the basis of 100 points.

The final exam counts for 40 percent of the grade. The midterm counts for 35 percent of the grade. The three quizzes together count for a total of 25 percent of the grade. (Do not forget to convert the quiz scores to percentages before they are averaged in.)

Any grade of 90 or more is an A, any grade of 80 or more (but less than 90) is a B, any grade of 70 or more (but less than 80) is a C, any grade of 60 or more (but less than 70) is a D, and any grade below 60 is an F. The program will read in the student's scores and output the student's record, which consists of three quiz scores and two exam scores as well as the student's overall numeric score for the entire course and final letter grade.

Define and use a class for the student record. The class should have instance variables for the quizzes, midterm, final, overall numeric score for the course, and final letter grade. The overall numeric score is a number in the range 0 to 100, which represents the weighted average of the student's work. The class should have methods to compute the overall numeric grade and the final letter grade. These last methods will be `void` methods that set the appropriate instance variables. Your class should have a reasonable set of accessor and mutator methods, an `equals` method, and a `toString` method, whether or not your program uses them. You may add other methods if you wish.

3. Write a `Temperature` class that has two instance variables: a temperature value (a floating-point number) and a character for the scale, either `'C'` for Celsius or `'F'` for Fahrenheit. The class should have four constructor methods: one for each instance variable (assume zero degrees if no value is specified and Celsius if no scale is specified), one with two parameters for the two instance variables, and a no-argument constructor (set to zero degrees Celsius). Include (1) two accessor methods to return the temperature, one to return the degrees Celsius, the other to return the degrees Fahrenheit—use the following formulas to write the two methods, and round to the nearest tenth of a degree:

$$degreesC = 5(degreesF - 32)/9$$
$$degreesF = (9(degreesC)/5) + 32$$

(2) three mutator methods, one to set the value, one to set the scale (`'F'` or `'C'`), and one to set both; (3) three comparison methods, an `equals` method to test whether two temperatures are equal, one method to test whether one temperature is greater than another, and one method to test whether one temperature is less than another (note that a Celsius temperature can be equal to a Fahrenheit temperature as indicated by the above formulas); and (4) a suitable `toString` method. Then write a driver program (or programs) that tests all the methods. Be sure to use each of the constructors, to include at least one true and one false case for each of the comparison methods, and to test at least the following temperature equalities: 0.0 degrees C = 32.0 degrees F, –40.0 degrees C = –40.0 degrees F, and 100.0 degrees C = 212.0 degrees F.

4. Redefine the class `Date` in Display 4.11 so that the instance variable for the month is of type `int` instead of type `String`. None of the method headings will change in any way. In particular, no `String` type parameters will change to `int` type parameters. You must redefine the methods to make things work out. Any program that uses the `Date` class from Display 4.11 should be able to use your `Date` class without any changes in the program. In

particular, the program in Display 4.12 should work the same whether the `Date` class is defined as in Display 4.11 or is defined as you do it for this project. Write a test program (or programs) that tests each method in your class definition.

5. Define a class whose objects are records on animal species. The class will have instance variables for the species name, population, and growth rate. The growth rate is a percentage that can be positive or negative and can exceed 100 percent. Include a suitable collection of constructors, mutator methods, and accessor methods. Also, include a `toString` method and an `equals` method. Also, include a `boolean` valued method named `endangered` that returns `true` when the growth rate is negative and returns `false` otherwise. Write a test program (or programs) that tests each method in your class definition.