CHAPTER 5

Defining Classes II

5.1 STATIC METHODS AND STATIC VARIABLES 237

Static Methods 237

Pitfall: Invoking a Nonstatic Method Within a Static Method 239

Static Variables 241

The Math Class 244

Wrapper Classes 247

Pitfall: A Wrapper Class Does Not Have a No-Argument Constructor 251

5.2 REFERENCES AND CLASS PARAMETERS 252

Variables and Memory 252

References 254

Class Parameters 259

Pitfall: Use of = and == with Variables of a Class Type 262

The Constant null 263

JOptionPane Revisited 263

Pitfall: Null Pointer Exception 264

The new Operator and Anonymous Objects 265

5.3 USING AND MISUSING REFERENCES 267

Example: A Person Class 267

Pitfall: null Can Be an Argument to a Method 275

Copy Constructors 276

Pitfall: Privacy Leaks 279

Mutable and Immutable Classes 282

Tip: Deep Copy versus Shallow Copy 283

Tip: Assume Your Coworkers Are Malicious 284

5.4 PACKAGES AND javadoc 285

Packages and import Statements 285

The Package java.lang 287

Package Names and Directories 287

Pitfall: Subdirectories Are Not Automatically Imported 290

The Default Package 290

Specifying a Class Path When You Compile ✜ 291

Pitfall: Not Including the Current Directory in Your Class Path 291

Name Clashes ✜292

Introduction to javadoc ✜ 293

Commenting Classes for javadoc ✜ 293

Running javadoc ✜ 295

CHAPTER SUMMARY 297

ANSWERS TO SELF-TEST EXERCISES 298

PROGRAMMING PROJECTS 301

5 Defining Classes II

There never are any good quotes left for Part II.
All the good quotes get used for Part I.

Anonymous

Introduction

This chapter is a continuation of Chapter 4. It covers the rest of the core material on defining classes. We start by discussing *static methods* and *static variables,* which are methods and variables that belong to the class as a whole and not to particular objects. We then go on to discuss how class type variables name objects of their class and how class type parameters are handled in Java.

This chapter also discusses *packages,* which are Java's way of grouping classes into libraries. We end this chapter with a discussion of `javadoc`, a program that automatically extracts documentation from classes and packages.

Prerequisites

This chapter uses material from Chapters 1 through 4.

Sections 5.3 and 5.4 are independent of each other and may be covered in any order. Section 5.3 covers some subtle points about references and Section 5.4 covers packages and `javadoc`. The material on `javadoc` is not used in the rest of this book. The other material in Sections 5.3 and 5.4 is not heavily used in the next few chapters, and can be digested as needed if the material seems difficult on first reading.

The material on packages in Section 5.4 assumes that you know about directories (which are called folders in some operating systems), that you know about path names for directories (folders), and that you know about PATH (environment) variables. These are not Java topics. They are part of your operating system, and the details depend on your particular operating system. If you can find out how to set the PATH variable on your operating system, you will know enough about these topics to understand the material on packages in Section 5.4.

5.1 Static Methods and Static Variables

All for one, one for all, that is our device.

Alexandre Dumas, *The Three Musketeers*

STATIC METHODS

Some methods do not require a calling object. Methods to perform simple numeric calculations are a good example of such methods. For example, a method to compute the maximum of two integers has no obvious candidate for a calling object. In Java, you can define a method so that it requires no calling object. Such methods are known as **static methods**. You define a static method in the same way as any other method, but you add the keyword `static` to the method definition heading, as in the following example:

static method

```
public static int maximum(int n1, int n2)
{
    if (n1 > n2)
        return n1;
    else
        return n2;
}
```

Although a static method requires no calling object, it still belongs to some class and its definition is given inside the class definition. When you invoke a static method, you normally use the class name in place of a calling object. So, if the above definition of the method `maximum` were in a class named `SomeClass`, then the following is a sample invocation of `maximum`:

```
int budget = SomeClass.maximum(yourMoney, myMoney);
```

where `yourMoney` and `myMoney` are variables of type `int` that contain some values.

A sample of some static method definitions as well as a program that uses the methods are given in Display 5.1.

We have already been using static methods. For example, we have used the method `parseInt`, which is a static method in the class `Integer`. The following is an invocation of the method `parseInt`, which we used in Display 4.11:

```
int monthInput = Integer.parseInt(keyboard.readLine());
```

Another static method we have used is the `exit` method of the class `System`, which is invoked as follows:

```
System.exit(0);
```

Display 5.1 Static Methods *(Part 1 of 2)*

```
1   /**
2   Class with static methods for circles and spheres.
3   */
4   public class RoundStuff
5   {
6       public static final double PI = 3.14159;
7
8       /**
9        Return the area of a circle of the given radius.
10      */
11      public static double area(double radius)
12      {
13          return (PI*radius*radius);
14      }
15
16      /**
17       Return the volume of a sphere of the given radius.
18      */
19      public static double volume(double radius)
20      {
21          return ((4.0/3.0)*PI*radius*radius*radius);
22      }
23  }
```

This is the file RoundStuff.java

```
1   public class RoundStuffDemo
2   {
3       public static void main(String[] args)
4       {
5           double radius = 2;
6
7           System.out.println("A circle of radius "
8                                                 + radius + " inches");
9           System.out.println("has an area of " +
10               RoundStuff.area(radius) + " square inches.");
11          System.out.println("A sphere of radius "
12                                                + radius + " inches");
13          System.out.println("has an volume of " +
14               RoundStuff.volume(radius) + " cubic inches.");
15      }
16  }
```

This is the file RoundStuffDemo.java

Display 5.1 Static Methods *(Part 2 of 2)*

SAMPLE DIALOGUE

```
A circle of radius 2.0 inches
has an area of 12.56636 square inches.
A sphere of radius 2.0 inches
has a volume of 33.51029333333333 cubic inches.
```

Note that with a static method, the class name serves the same purpose as a calling object. (It would be legal to create an object of the class `System` and use it to invoke the method `exit`, but that is confusing style, so we usually use the class name when invoking a static method.)

Within the definition of a static method, you cannot do anything that refers to a calling object, such as accessing an instance variable. This makes perfectly good sense, because a static method can be invoked without using any calling object and so can be invoked when there are no instance variables. (Remember instance variables belong to the calling object.) The best way to think about this restriction is in terms of the `this` parameter. In a static method, you cannot use the `this` parameter, either explicitly or implicitly. For example, the name of an instance variable by itself has an implicit `this` and a dot before it. So, you cannot use an instance variable in the definition of a static method.

Pitfall

INVOKING A NONSTATIC METHOD WITHIN A STATIC METHOD

If `myMethod()` is a nonstatic (that is, ordinary) method in a class, then within the definition of any method of this class, an invocation of the form

```
myMethod();
```

means

```
this.myMethod();
```

and so it is illegal within the definition of a static method. (A static method has no `this`.)

However, it is legal to invoke a static method within the definition of another static method.

There is one way that you can invoke a nonstatic method within a static method: If you create an object of the class and use that object of the class (rather than `this`) as the calling object. For example, suppose `myMethod()` is a nonstatic method in the class `MyClass`. Then, as we already discussed, the following is illegal within the definition of a static method in the class `MyClass`:

```
myMethod();
```

However, the following is perfectly legal in a static method or any method definition:

```
MyClass anObject = new MyClass();
anObject.myMethod();
```

The method `main` is a static method and you will often see code similar to this in the `main` method of a class.

Static Methods

A **static method** is one that can be used without a calling object. With a static method, you normally use the class name in place of the calling object.

When you define a static method, you place the keyword `static` in the heading of the definition.

Since it does not need a calling object, a static method cannot refer to an instance variable of the class, nor can it invoke a nonstatic method of the class (unless it creates a new object of the class and uses that object as the calling object). Another way to phrase it is that, in the definition of a static method, you cannot use an instance variable or method that has an implicit or explicit `this` for a calling object.

Self-Test Exercises

1. Is the following legal? The class `RoundStuff` is defined in Display 5.1.

```
RoundStuff roundObject = new RoundStuff( );
System.out.println("A circle of radius 5.5 has area"
                   + roundObject.area(5.5);
```

2. In Display 5.1 we did not define any constructors for the class `RoundStuff`. Is this poor programming style?
3. Can a class contain both static and nonstatic (that is, regular) methods?
4. Can you invoke a nonstatic method within a static method?
5. Can you invoke a static method within a nonstatic method?
6. Can you reference an instance variable within a static method? Why or why not?

STATIC VARIABLES

A class can have static variables as well as static methods. A **static variable** is a variable that belongs to the class as a whole and not just to one object. Each object has its own copies of the instance variables. However, with a static variable there is only one copy of the variable, and all the objects can use this one variable. Thus, a static variable can be used by objects to communicate between the objects. One object can change the static variable and another object can read that change. To make a variable static, you declare it like an instance variable but add the modifier `static` as follows:

static variable

```
private static int turn;
```

Or if you wish to initialize the static variable, which is typical, you might declare it as follows instead:

```
private static int turn = 0;
```

If you do not initialize a static variable, it will be automatically initialized to a default value: Static variables of type `boolean` are automatically initialized to `false`. Static variables of other primitive types are automatically initialized to the zero of their type. Static variables of a class type are automatically initialized to `null`, which is a kind of placeholder for an object that we will discuss later in this chapter. However, we prefer to explicitly initialize static variables, either as just shown or in a constructor.

default initialization

Display 5.2 gives an example of a class with a static variable along with a demonstration program. Notice that the two objects, `lover1` and `lover2`, access the same static variable `turn`.

As we already noted, you cannot directly access an instance variable within the definition of a static method. However, it is perfectly legal to access a static variable within a static method, because a static variable belongs to the class as a whole. This is illustrated by the method `getTurn` in Display 5.2. When we write `turn` in the definition of the static method `getTurn`, it does not mean `this.turn`; it means `TurnTaker.turn`. If the static variable `turn` were marked `public` instead of `private`, it would even be legal to use `TurnTaker.turn` outside of the definition of the class `TurnTaker`.

Defined constants that we have already been using, such as the following, are a special kind of static variable:

```
public static final double PI = 3.14159;
```

The modifier `final` in the above means that the static variable `PI` cannot be changed. Such defined constants are normally public and can be used outside the class. This defined constant appears in the class `RoundStuff` in Display 5.1. To use this constant outside of the class `RoundStuff`, you write the constant in the form `RoundStuff.PI`.

Good programming style dictates that static variables should normally be marked `private` unless they are marked `final`, that is, unless they are defined constants. The reason is the same as the reason for making instance variables `private`.

Display 5.2 A Static Variable *(Part 1 of 2)*

```
1   public class TurnTaker
2   {
3       private static int turn = 0;

4       private int myTurn;
5       private String name;

6       public TurnTaker(String theName, int theTurn)
7       {
8           name = theName;
9           if (theTurn >= 0)
10              myTurn = theTurn;
11          else
12          {
13              System.out.println("Fatal Error.");
14              System.exit(0);
15          }
16      }

17      public TurnTaker()
18      {
19          name = "No name yet";
20          myTurn = 0;//Indicating no turn.
21      }

22      public String getName()
23      {
24          return name;
25      }

26      public static int getTurn( )
27      {
28          turn++;
29          return turn;
30      }

31      public boolean isMyTurn( )
32      {
33          return (turn == myTurn);
34      }
35  }
```

This is the file TurnTaker.java

You cannot access an instance variable in a static method, but you can access a static variable in a static method.

Display 5.2 A Static Variable *(Part 2 of 2)*

```
1   public class StaticDemo                                 This is the file
2   {                                                       StaticDemo.java
3       public static void main(String[] args)
4       {
5           TurnTaker lover1 = new TurnTaker("Romeo", 1);
6           TurnTaker lover2 = new TurnTaker("Juliet", 3);
7           for (int i = 1; i < 5; i++)
8           {
9               System.out.println("Turn = " + TurnTaker.getTurn());
10              if (lover1.isMyTurn())
11                  System.out.println("Love from " + lover1.getName());
12              if (lover2.isMyTurn())
13                  System.out.println("Love from " + lover2.getName());
14          }
15      }
16  }
```

SAMPLE DIALOGUE

```
Turn = 1
Love from Romeo
Turn = 2
Turn = 3
Love from Juliet
Turn = 4
```

STATIC VARIABLES

A **static variable** belongs to the class as a whole. All objects of the class can read and change the static variable. Static variables should normally be private, unless they happen to be defined constants.

SYNTAX:

```
private static Type Variable_Name;
private static Type Variable_Name = Initial_Value;
public static final Type Variable_Name = Constant_Value;
```

EXAMPLES:

```
private static int turn = 0;
public static final double PI = 3.14159;
```

Self-Test Exercises

7. What is the difference between a static variable and an instance variable?
8. Can you use an *instance variable* (without a class name and dot) in the definition of a *static method* of the same class? Can you use an *instance variable* (without an object name and dot) in the definition of a *nonstatic (ordinary) method* of the same class?
9. Can you use a *static variable* in the definition of a *static method* of the same class? Can you use a *static variable* in the definition of a *nonstatic (ordinary) method* of the same class?
10. Can you use the `this` parameter in the definition of a static method?
11. When we defined the class `Date` in Display 4.11 we had not yet discussed static methods, so we did not mark any of the methods `static`. However, some of the methods could have been marked `static` (and should have been marked `static`, if only we'd known what that meant). Which of the methods can be marked `static`? (If you omit the modifier `static` when it is appropriate, then the method cannot be invoked with the class name; it must be invoked with a calling object.)
12. Following the style guidelines given in this book, when should a static variable be marked `private`?
13. What do static methods and static variables have in common? After all, they are both called *static,* so it sounds like they have something in common.

THE Math CLASS

`Math` methods

The class `Math` provides a number of standard mathematical methods. The class `Math` is provided automatically and requires no `import` statement. Some of the methods in the class `Math` are described in Display 5.3. A more complete list of methods is given in Appendix 4. All of these methods are static, which means that you normally use the class name `Math` in place of a calling object.

Display 5.3 Some Methods in the Class Math *(Part 1 of 3)*

The `Math` class is in the `java.lang` package, so it requires no `import` statement.

```
public static double pow(double base, double exponent)
```

Returns base to the power exponent.

EXAMPLE:

`Math.pow(2.0,3.0)` returns `8.0`.

Display 5.3 Some Methods in the Class Math *(Part 2 of 3)*

```
public static double abs(double argument)
public static float abs(float argument)
public static long abs(long argument)
public static int abs(int argument)
```

Returns the absolute value of the argument. (The method name abs is overloaded to produce four similar methods.)

EXAMPLE:

Math.abs(−6) and Math.abs(6) both return 6. Math.abs(−5.5) and Math.abs(5.5) both return 5.5.

```
public static double min(double n1, double n2)
public static float min(float n1, float n2)
public static long min(long n1, long n2)
public static int min(int n1, int n2)
```

Returns the minimum of the arguments n1 and n2. (The method name min is overloaded to produce four similar methods.)

EXAMPLE:

Math.min(3, 2) returns 2.

```
public static double max(double n1, double n2)
public static float max(float n1, float n2)
public static long max(long n1, long n2)
public static int max(int n1, int n2)
```

Returns the maximum of the arguments n1 and n2. (The method name max is overloaded to produce four similar methods.)

EXAMPLE:

Math.max(3, 2) returns 3.

```
public static long round(double argument)
public static int round(float argument)
```

Rounds its argument.

EXAMPLE:

Math.round(3.2) returns 3; Math.round(3.6) returns 4.

```
public static double ceil(double argument)
```

Returns the smallest whole number greater than or equal to the argument.

EXAMPLE:

Math.ceil(3.2) and Math.ceil(3.9) both return 4.0.

Display 5.3 Some Methods in the Class Math *(Part 3 of 3)*

```
public static double floor(double argument)
```

Returns the largest whole number less than or equal to the argument.

EXAMPLE:

`Math.floor(3.2)` and `Math.floor(3.9)` both return `3.0`.

```
public static double sqrt(double argument)
```

Returns the square root of its argument.

EXAMPLE:

`Math.sqrt(4)` returns `2.0`.

The class `Math` has three similar methods named `round`, `floor`, and `ceil`. Some of these return a value of type `double`, but they all return a value that is intuitively a whole number that is close to the value of their arguments. The method `round` rounds a number to the nearest whole number, and (if the argument is a `double`) it returns that whole number as a value of type `long`. If you want that whole number as a value of type `int`, you must use a type cast as in the following:

```
double exact = 7.56;
int roundedValue = (int)Math.round(exact);
```

You cannot assign a `long` value to a variable of type `int`, even if it is a value like `8`, which could just as well have been an `int`. A value like `8` can be of type either `int` or `long` (or even of type `short` or `byte`) depending on how it was created.

`floor` and `ceil`

The methods `floor` and `ceil` are similar to, but not identical to, `round`. Neither one rounds, although they both yield a whole number that is close to their argument. They both return a whole number as a value of type `double` (not of type `int` or `long`). The method `floor` returns the nearest whole number that is less than or equal to its argument. So, `Math.floor(5.9)` returns `5.0`, not `6.0`. `Math.floor(5.2)` also returns `5.0`.

The method `ceil` returns the nearest whole number that is greater than or equal to its argument. The word `ceil` is short for "ceiling." `Math.ceil(5.1)` returns `6.0`, not `5.0`. `Math.ceil(5.9)` also returns `6.0`.

If you want to store the value returned by either `floor` or `ceil` in a variable of type `int`, you must use a type cast as in the following example:

```
double exact = 7.56;
int lowEstimate = (int)Math.floor(exact);
int highEstimate = (int)Math.ceil(exact);
```

`Math.floor(exact)` returns the `double` value `7.0`, and the variable `lowEstimate` receives the `int` value 7. `Math.ceil(exact)` returns the `double` value `8.0`, and the variable `highEstimate` receives the `int` value `8`.

(Since values of type `double` are effectively approximate values, a safer way to compute the floor or ceiling as an `int` value is the following:

```
double exact = 7.56;
int lowEstimate = (int)Math.round(Math.floor(exact));
int highEstimate = (int)Math.round(Math.ceil(exact));
```

This way if `Math.floor(exact)` returns slightly less than `7.0`, the final result will still be `7` not `6`, and if `Math.ceil(exact)` returns slightly less than `8.0`, the final result will still be `8` not `7`.)

`Math` constants

The class `Math` also has the two predefined constants `E` and `PI`. The constant `PI` (often written π in mathematical formulas) is used in calculations involving circles, spheres, and other geometric figures based on circles. `PI` is approximately `3.14159`. The constant `E` is the base of the natural logarithm system (often written e in mathematical formulas) and is approximately `2.72`. (We do not use the predefined constant `E` in this text.) The constants `PI` and `E` are defined constants, as described in Chapter 1. For example, the following computes the area of a circle, given its radius:

```
area = Math.PI * radius * radius;
```

Notice that because the constants `PI` and `E` are defined in the class `Math`, they must have the class name `Math` and a dot before them. For example, you could use the constant `Math.PI` in the definition of the class `RoundStuff` in Display 5.1 instead of defining a value for `PI` inside the class definition as we did in Display 5.1.

Self-Test Exercises

14. What values are returned by each of the following?

```
Math.round(3.2), Math.round(3.6),
Math.floor(3.2), Math.floor(3.6),
Math.ceil(3.2), and Math.ceil(3.6).
```

15. Suppose `answer` is a variable of type `double`. Write an assignment statement to assign `Math.round(answer)` to the `int` variable `roundedAnswer`.

16. Suppose `n` is of type `int` and `m` is of type `long`. What is the type of the value returned by `Math.min(n, m)`? Is it `int` or `long`?

WRAPPER CLASSES

Java treats the primitive types, such as `int` and `double`, differently from the class types, such as the class `String` and the programmer-defined classes. For example, later in this chapter you will see that an argument to a method is treated differently depending on whether the argument is of a primitive or class type. At times you may find yourself in a situation where you want to use a primitive type but you want or need the type to be

wrapper class

a class type. **Wrapper classes** provide a class type corresponding to each of the primitive types so that you can have class types that behave somewhat like primitive types.

To convert a value of a primitive type to an "equivalent" value of a class type, you create an object of the corresponding wrapper class using the primitive type value as an argument to the wrapper class constructor. The wrapper class for the primitive type `int` is the predefined class `Integer`. If you want to convert an `int` value, such as `42`, to an object of type `Integer`, you can do so as follows:

`Integer` class

```
Integer integerObject = new Integer(42);
```

The variable `integerObject` now names an object of the class `Integer` that corresponds to the `int` value `42`. (The object `integerObject` does in fact have the `int` value `42` stored in an instance variable of the object `integerObject`.) This process of going from a value of a primitive type to an object of its wrapper class is sometimes called **boxing**.

boxing

To go in the reverse direction, from an object of type `Integer` to the corresponding `int` value, you do the following:

```
int i = integerObject.intValue();
```

The method `intValue()` recovers the corresponding `int` value from an object of type `Integer`. This process of going from an object of a wrapper class to the corresponding value of a primitive type is sometimes called **unboxing**.

unboxing

other wrapper classes

The wrapper classes for the primitive types `byte`, `short`, `long`, `float`, `double`, and `char` are `Byte`, `Short`, `Long`, `Float`, `Double`, and `Character`, respectively. The methods for converting from the wrapper class object to the corresponding primitive type are `intValue` for the class `Integer`, as we have already seen, `byteValue` for the classes `Byte`, `shortValue` for the classes `Short`, `longValue` for the classes `Long`, `floatValue` for the class `Float`, `doubleValue` for the class `Double`, and `charValue` for the class `Character`.

Wrapper Classes

Every primitive type has a corresponding wrapper class. A wrapper class allows you to have a class object that corresponds to a value of a primitive type. Wrapper classes also contain a number of useful predefined constants and static methods.

The material on wrapper classes that we have seen thus far explains why they are called *wrapper classes.* However, a possibly even more important use of the wrapper classes is that they contain a number of useful constants and static methods. So, wrapper classes have two distinct personalities: one is their ability to produce class objects corresponding to values of primitive types, and the other is as a repository of useful constants and methods. It was not necessary to combine these two personalities into one kind of class. Java could have had two sets of classes, one for each per-

sonality, but the designers of the Java libraries chose to have only one set of classes for both personalities.

You can use the associated wrapper class to find the value of the largest and smallest values of any of the primitive number types. For example, the largest and smallest values of type `int` are

largest and smallest values

```
Integer.MAX_VALUE and Integer.MIN_VALUE
```

The largest and smallest values of type `double` are

```
Double.MAX_VALUE and Double.MIN_VALUE
```

Wrapper classes have static methods that can be used to convert back and forth between string representations of numbers and the corresponding number of type `int`, `double`, `long`, or `float`. For example, the static method `parseDouble` of the wrapper class `Double` will convert a string to a value of type `double`. So,

parse-Double

```
Double.parseDouble("199.98")
```

returns the `double` value `199.98`. If there is any possibility that the string named by `theString` has extra leading or trailing blanks, you should instead use

```
Double.parseDouble(theString.trim( ))
```

The method `trim` is a method in the class `String` that trims off leading and trailing white space, such as blanks.

If the string is not a correctly formed numeral, then the invocation of `Double.parseDouble` will cause your program to end. The use of `trim` helps somewhat in avoiding this problem.

Similarly, the static methods `Integer.parseInt`, `Long.parseLong`, and `Float.parseFloat` convert from string representations to numbers of the corresponding primitive types `int`, `long`, and `float`, respectively.

parseInt

Each of the numeric wrapper classes also has a static method called `toString` that will convert in the other direction from a numeric value to a string representation of the numeric value. For example,

```
Double.toString(123.99)
```

returns the string value `"123.99"`.

`Character`, the wrapper class for the primitive type `char`, contains a number of static methods that are useful for string processing. Some of these methods are shown in Display 5.4.

Character

There is also a wrapper class `Boolean` corresponding to the primitive type `boolean`. It has names for two constants of type `Boolean`, `Boolean.TRUE` and `Boolean.FALSE`, which are the `Boolean` objects corresponding to the values `true` and `false` of the primitive type `boolean`.

Boolean

Display 5.4 **Some Methods in the Class Character** ***(Part 1 of 2)***

The class `Character` is in the `java.lang` package, so it requires no `import` statement.

```
public static char toUpperCase(char argument)
```

Returns the uppercase version of its `argument`. If the `argument` is not a letter, it is returned unchanged.

EXAMPLE:

`Character.toUpperCase('a')` and `Character.toUpperCase('A')` both return `'A'`.

```
public static char toLowerCase(char argument)
```

Returns the lowercase version of its `argument`. If the `argument` is not a letter, it is returned unchanged.

EXAMPLE:

`Character.toLowerCase('a')` and `Character.toLowerCase('A')` both return `'a'`.

```
public static boolean isUpperCase(char argument)
```

Returns `true` if its `argument` is an uppercase letter; otherwise returns `false`.

EXAMPLE:

`Character.isUpperCase('A')` returns `true`. `Character.isUpperCase('a')` and `Charac-ter.isUpperCase('%')` both return `false`.

```
public static boolean isLowerCase(char argument)
```

Returns `true` if its `argument` is a lowercase letter; otherwise returns `false`.

EXAMPLE:

`Character.isLowerCase('a')` returns `true`. `Character.isLowerCase('A')` and `Charac-ter.isLowerCase('%')` both return `false`.

```
public static boolean isWhitespace(char argument)
```

Returns `true` if its `argument` is a whitespace character; otherwise returns `false`. Whitespace characters are those that print as white space, such as the space character (blank character), the tab character (`'\t'`), and the line break character (`'\n'`).

EXAMPLE:

`Character.isWhitespace(' ')` returns `true`. `Character.isWhitespace('A')` returns `false`.

```
public static boolean isLetter(char argument)
```

Returns `true` if its `argument` is a letter; otherwise returns `false`.

EXAMPLE:

`Character.isLetter('A')` returns `true`. `Character.isLetter('%')` and `Character.isLet-ter('5')` both return `false`.

Display 5.4 Some Methods in the Class Character *(Part 1 of 2)*

```
public static boolean isDigit(char argument)
```

Returns `true` if its `argument` is a digit; otherwise returns `false`.

EXAMPLE:

`Character.isDigit('5')` returns `true`. `Character.isDigit('A')` and `Charac-ter.isDigit('%')` both return `false`.

```
public static boolean isLetterOrDigit(char argument)
```

Returns `true` if its `argument` is a letter or a digit; otherwise returns `false`.

EXAMPLE:

`Character.isLetterOrDigit('A')` and `Character.isLetterOrDigit('5')` both return `true`. `Character.isLetterOrDigit('&')` returns `false`.

Pitfall

A Wrapper Class Does Not Have a No-Argument Constructor

Normally it is good programming practice to define a no-argument constructor for any class you define. However, on rare occasions a no-argument constructor simply does not make sense. The wrapper classes discussed in the previous subsection do not have a no-argument constructor. This makes sense if you think about it. To use the static methods in a wrapper class, you need no calling object and hence need no constructor at all. The other function of a wrapper class is to provide a class object corresponding to a value of a primitive type. For example,

```
new Integer(42)
```

creates an object of the class `Integer` that corresponds to the `int` value 42. There is no no-argument constructor for the class `Integer` because it makes no sense to have an object of the class `Integer` unless it corresponds to an `int` value, and if it does correspond to an `int` value, that `int` value is naturally an argument to the constructor.

Self-Test Exercises

17. Suppose `result` is a variable of type `double` that has a value. Write a Java expression that returns a string that is the normal way of writing the value in `result`.

18. Suppose `stringForm` is a variable of type `String` that names a `String` that is the normal way of writing some `double`, such as `"41.99"`. Write a Java expression that returns the `double` value named by `stringForm`.

19. How would you do exercise 18 if the string might contain leading and/or trailing blanks, such as " 41.99 "?

20. Write Java code to output the largest and smallest values of type `long` allowed in Java.

21. How do you create an object of the class `Character` that corresponds to the letter `'Z'`?

22. Does the class `Character` have a no-argument constructor?

23. What is the output produced by the following code?

```
Character characterObject1 = new Character('a');
Character characterObject2 = new Character('A');
if (characterObject1.equals(characterObject2))
    System.out.println("Objects are equal.");
else
    System.out.println("Objects are Not equal.");
```

5.2 References and Class Parameters

Do not mistake the pointing finger for the moon.

Zen Saying

Variables of a class type and variables of a primitive type behave quite differently in Java. Variables of a primitive type name their values in a straightforward way. For example, if `n` is an `int` variable, then `n` can contain a value of type `int`, such as `42`. If `v` is a variable of a class type, then `v` does not directly contain an object of its class. Instead, `v` names an object by containing the memory address of where the object is located in memory. In this section, we discuss how a variable of a class type names objects, and we also discuss the related topic of how method parameters of a class type behave in Java.

VARIABLES AND MEMORY

secondary and main memory

A computer has two forms of memory called *main memory* and *secondary memory.* The **secondary memory** is used to hold files for more or less permanent storage. The **main memory** is used by the computer when it is running a program. Values stored in a program's variables are kept in this main memory. It will help our understanding of class type variables to learn a few details about how program variables are represented in (main) memory. For now assume that each variable in a program is of some primitive type, such as `int`, `double`, or `char`. Once you understand how variables of a primitive type are stored in memory, it will be easier to describe how variables of a class type behave.

byte

Main memory consists of a long list of numbered locations called **bytes**, each containing eight bits; that is, eight 0/1 digits. The number that identifies a byte is called its

address. A data item, such as a number or a letter, can be stored in one of these bytes, and the address of the byte is then used to find the data item when it is needed.

address

Values of most data types have values that require more than one byte of storage. When a data type requires more than one byte of storage, several adjacent bytes are used to hold the data item. In this case the entire chunk of memory that holds the data item is still called a **memory location**. The address of the first of the bytes that make up this memory location is used as the address for this larger memory location. Thus, as a practical matter, you can think of the computer's main memory as a long list of memory locations of *varying sizes*. The size of each of these locations is expressed in bytes, and the address of the first byte is used as the address (name) of that memory location. Display 5.5 shows a picture of a hypothetical computer's main memory. Each primitive type variable in a program is assigned one of these memory locations, and the value of the variable is stored in this memory location.

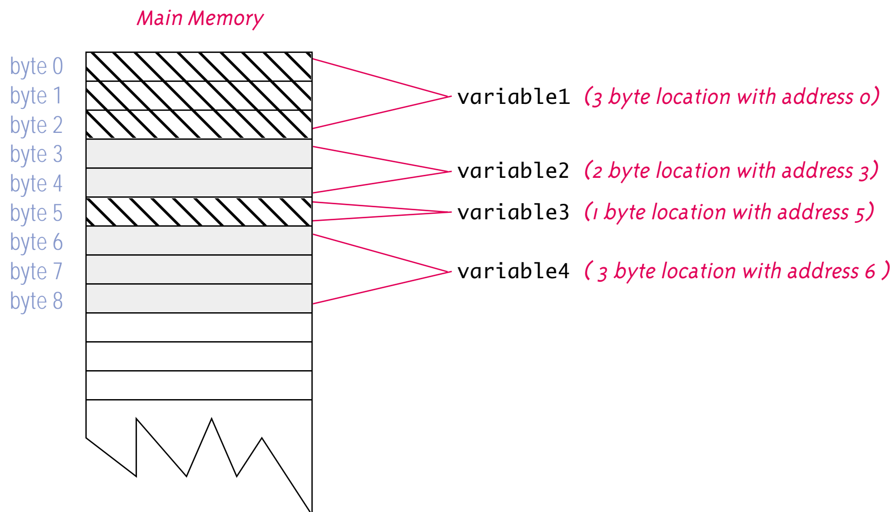
memory location

variables of a primitive type

Bytes and Addresses

Main memory is divided into numbered locations called **bytes**. The number associated with a byte is called its **address**. A group of consecutive bytes is used as the location for the value of a variable. The address of the first byte in the group is used as the address of this larger memory location.

Display 5.5 Variables in Memory



Why Eight Bits?

A **byte** is a memory location that can hold eight bits. What is so special about eight? Why not ten bits? There are two reasons why eight is special. First, eight is a power of 2. (8 is 2^3.) Since computers use bits, which have only two possible values, powers of 2 are more convenient than powers of 10. Second, it turns out that it requires seven bits to code a single character of the ASCII character set. So, eight bits (one byte) is the smallest power of 2 that will hold a single ASCII character.

References

In order to have a simple example to help explain *references*, we will use the class `ToyClass` defined in Display 5.6.

Variables of a class type name objects of its class differently from how variables of primitive types, such as `int` or `char`, store their values. Every variable, whether of a primitive type or a class type, is implemented as a location in the computer memory. For a variable of a primitive type, the value of the variable is stored in the memory location assigned to the variable. However, a variable of a class type stores only the memory address of where an object is located. The object named by the variable is stored in some other location in memory, and the variable contains only the memory address of where the object is stored. This memory address is called a **reference** (to the object).[1] This is diagrammed in Display 5.7.

references

Variables of a primitive type and variables of a class type are different for a reason. A value of a primitive type, such as the type `int`, always requires the same amount of memory to store one value. There is a maximum value of type `int`, so values of type `int` have a limit on their size. However, an object of a class type, such as an object of the class `String`, might be of any size. The memory location for a variable of type `String` is of a fixed size, so it cannot store an arbitrarily long string. It can, however, store the address of any string since there is a limit to the size of an address.

Since variables of a class type contain a reference (memory address), two variables may contain the same reference, and in such a situation, both variables name the same object. Any change to the object named by one of these variables will produce a change to the object named by the other variable, since they are the same object. For example, consider the following code. (The class `ToyClass` is defined in Display 5.6, but the meaning of the code should be obvious and you should not need to look up the definition.)

```
ToyClass variable1 = new ToyClass("Joe", 42);
ToyClass variable2;
```

[1] Readers familiar with languages that use pointers will recognize a reference as another name for a pointer. However, Java does not use the term *pointer*, but instead uses the term *reference*. Moreover, these references are handled automatically. There are no programmer-accessible pointer (reference) operations for dereferencing or other pointer operations. The details are all handled automatically in Java.

```
variable2 = variable1; //Now both variables name the same object.
variable2.set("Josephine", 1);
System.out.println(variable1);
```

Display 5.6 A Simple Class

```
1   public class ToyClass
2   {
3       private String name;
4       private int number;

5       public ToyClass(String initialName, int initialNumber)
6       {
7           name = initialName;
8           number = initialNumber;
9       }

10      public ToyClass()
11      {
12          name = "No name yet.";
13          number = 0;
14      }

15      public void set(String newName, int newNumber)
16      {
17          name = newName;
18          number = newNumber;
19      }

20      public String toString()
21      {
22          return (name + " " + number);
23      }

24      public static void changer(ToyClass aParameter)
25      {
26          aParameter.name = "Hot Shot";
27          aParameter.number = 42;
28      }

29      public boolean equals(ToyClass otherObject)
30      {
31          return ( (name.equals(otherObject.name))
32                    && (number == otherObject.number) );
33      }
34  }
```

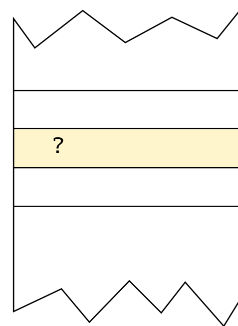
Display 5.7 Class Type Variables Store a Reference

```
public class ToyClass
{
    private String name;
    private int number;
```

The complete definition of the class ToyClass is given in Display 5.6.

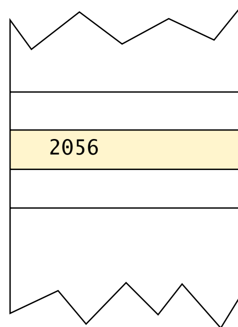
```
ToyClass sampleVariable;
```

Creates the variable sampleVariable in memory but assigns it no value.

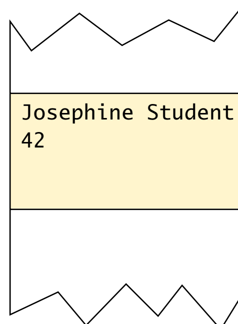



```
sampleVariable =
new ToyClass("Josephine Student", 42);
```

Creates an object and places the object someplace in memory and then places the address of the object in the variable sampleVariable. We do not know what the address of the object is, but let's assume it is 2056. The exact number does not matter.



For emphasis, we made the arrow point to the memory location referenced.



The output is

```
Josephine 1
```

The object named by `variable1` has been changed without ever using the name `variable1`. This is diagrammed in Display 5.8.

assignment with variables of a class type

Note that when you use the assignment operator with variables of a class type, you are assigning a reference (memory address), so the result of the following is to make `variable1` and `variable2` two names for the same object:

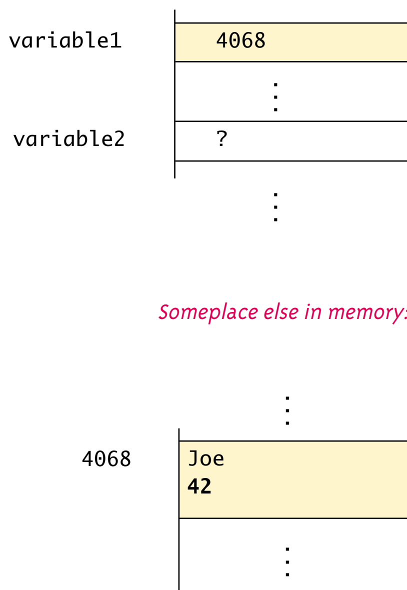
```
variable2 = variable1;.
```

Variables of a Class Type Hold References

A variable of a primitive type stores a value of that type. However, a variable of a class type does not store an object of that class. A variable of a class type stores the reference (memory address) of where the object is located in the computer's memory. This causes some operations, such as = and ==, to behave quite differently for variables of a class type than they do for variables of a primitive type.

Display 5.8 Assignment Operator with Class Type Variables *(Part 1 of 2)*

```
ToyClass variable1 = new ToyClass("Joe", 42);
ToyClass variable2;
```



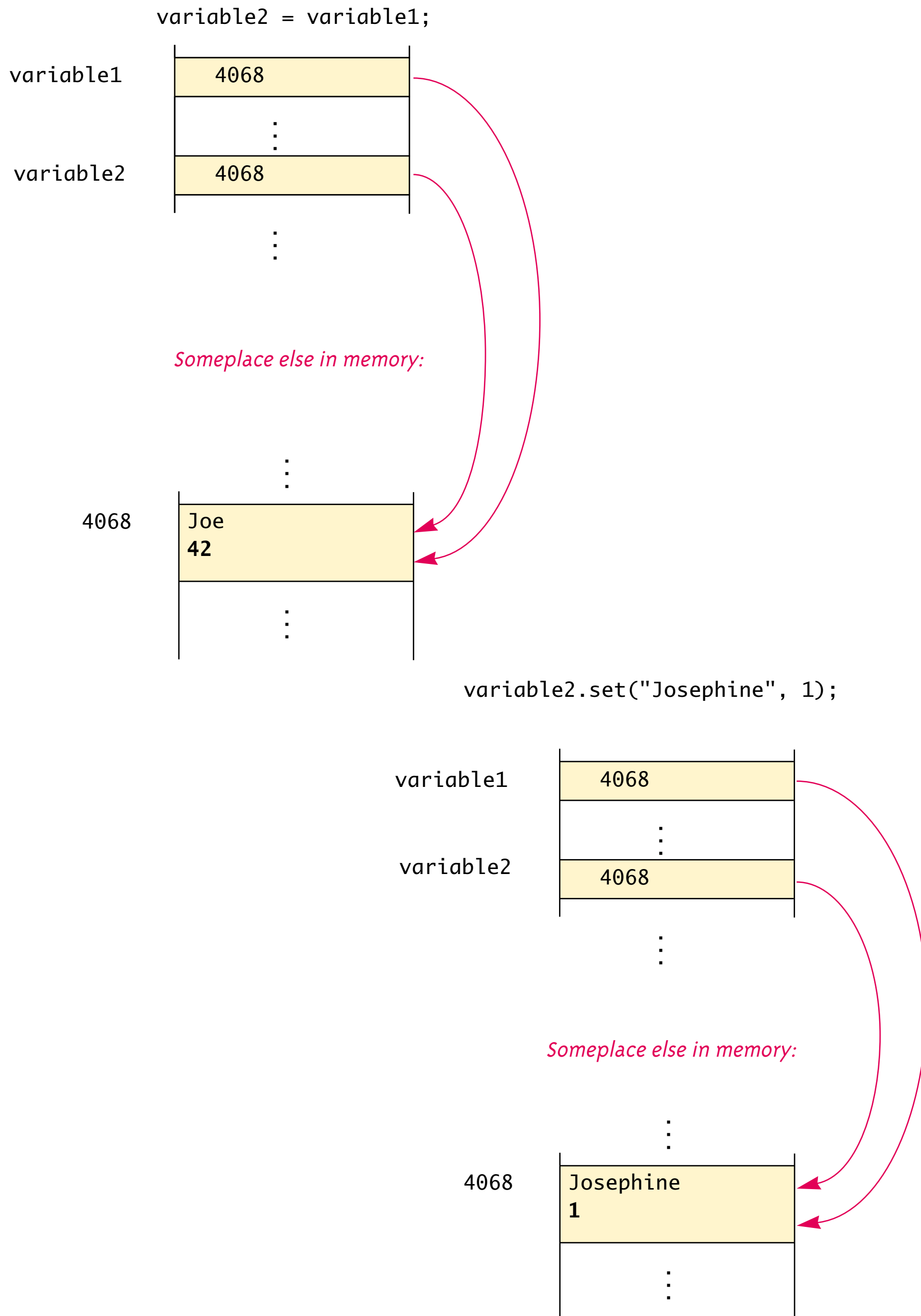
We do not know what memory address (reference) is stored in the variable `variable1`. Let's say it is 4068. The exact number does not matter.

Note that you can think of

```
new ToyClass("Joe", 42)
```

as returning a reference.

Display 5.8 Assignment Operator with Class Type Variables *(Part 2 of 2)*



Reference Types

A type whose variables contain references are called **reference types**. In Java, class types are reference types, but primitive types are not reference types.

A variable of a class type stores a memory address, and a memory address is a number. However, a variable of a class type cannot be used like a variable of a number type, such as `int` or `double`. This is intentional. The important property of a memory address is that it identifies a memory location. The fact that the implementors used numbers, rather than letters or strings or something else, to name memory locations is an accidental property. Java prevents you from using this accidental property to prevent you from doing things such as obtaining access to restricted memory or otherwise screwing up the computer.

CLASS PARAMETERS

Strictly speaking, all parameters in Java are call-by-value parameters. This means that when an argument is plugged in for a parameter (of any type), the argument is evaluated and the value obtained is used to initialize the value of the parameter. (Recall that a parameter is really a local variable.) However, in the case of a parameter of a class type, the value plugged in is a reference (memory address), and that makes class parameters behave quite differently from parameters of a primitive type.

Recall that the following makes `variable1` and `variable2` two names for the same object:

```
ToyClass variable1 = new ToyClass("Joe", 42);
ToyClass variable2;
variable2 = variable1;
```

So, any change made to `variable2` is in fact made to `variable1`. The same thing happens with parameters of a class type. The parameter is a local variable that is set equal to the value of its argument. But if its argument is a variable of a class type, this copies a reference into the parameter. So, the parameter becomes another name for the argument, and any change made to the object named by the parameter will be made to the object named by the argument, because they are the same object. Thus, a method can change the instance variables of an object given as an argument. A simple program to illustrate this is given in Display 5.9. Display 5.10 contains a diagram of the computer's memory as the program in Display 5.9 is executed.

Many programming languages have a parameter passing mechanism known as *call-by-reference.* If you are familiar with call-by-reference parameters, we should note that the Java parameter passing mechanism is similar to, but is not exactly the same as, call-by-reference.

Display 5.9 Parameters of a Class Type

ToyClass is defined in Display 5.6

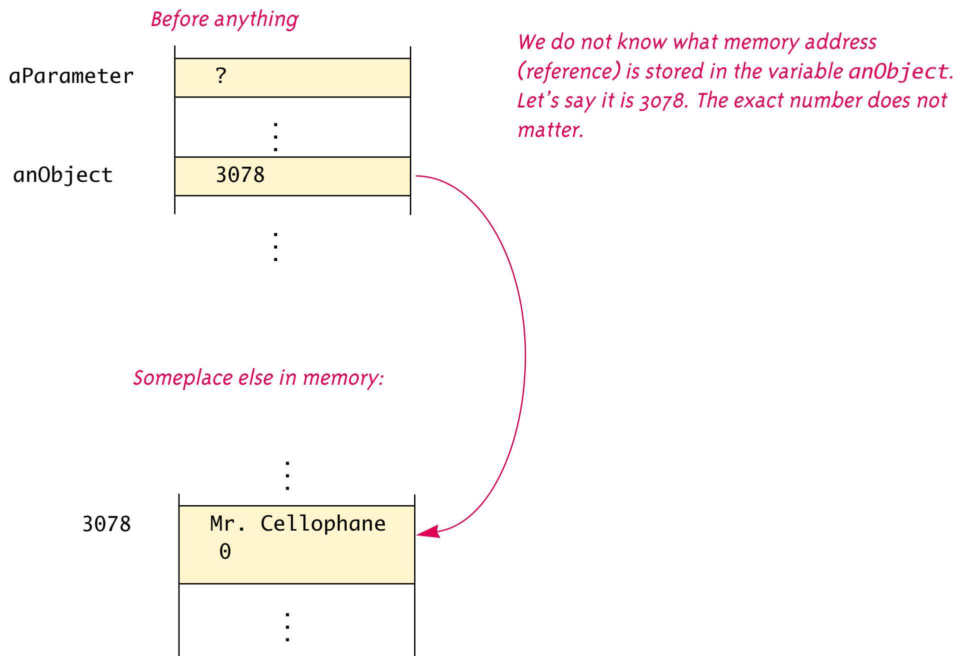
```
1   public class ClassParameterDemo
2   {
3       public static void main(String[] args)
4       {
5           ToyClass anObject = new ToyClass("Mr. Cellophane", 0);
6           System.out.println(anObject);
7           System.out.println(
8                   "Now we call changer with anObject as argument.");
9           ToyClass.changer(anObject);
10          System.out.println(anObject);
11      }
12  }
```

Notice that the method `changer` changed the instance variables in the object `anObject`.

Sample Dialogue

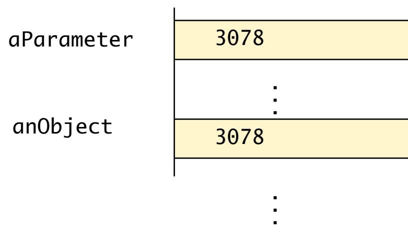
```
Mr. Cellophane 0
Now we call changer with anObject as argument.
Hot Shot 42
```

Display 5.10 Memory Picture for Display 5.9 *(Part 1 of 2)*

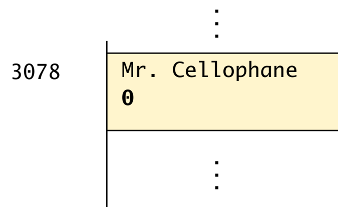


Display 5.10 Memory Picture for Display 5.9 *(Part 2 of 2)*

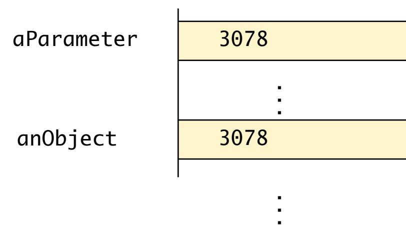
anObject is plugged in for aParamter.
anObject and aParameter become two names for the same object.



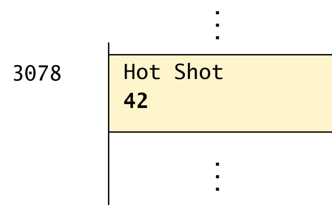
Someplace else in memory:



ToyClass.changer(anObject); is executed
and so the following are executed:
aParameter.name = "Hot Shot";
aParameter.number = 42;
As a result anObject is changed.



Someplace else in memory:



Differences Between Primitive and Class-Type Parameters

A method cannot change the value of a variable of a primitive type that is an argument to the method. On the other hand, a method can change the values of the instance variables of an argument of a class type.

Pitfall

Use of = and == with Variables of a Class Type

You have already seen that the assignment operator used with variables of a class type produces two variables that name the same object, which is very different from how assignment behaves with variables of a primitive type.

== with variables of a class type

The test for equality using == with variables of a class type also behaves in what may seem like a peculiar way. The operator == does not check that the objects have the same values for their instance variables. It merely checks for equality of memory address, so two objects in two different locations in memory would test as being "not equal" when compared using ==, even if their instance variables contain equivalent data. For example, consider the following code. (The class ToyClass is defined in Display 5.6.)

```
ToyClass variable1 = new ToyClass("Joe", 42),
         variable2 = new ToyClass("Joe", 42);
if (variable1 == variable2)
    System.out.println("Equal using ==");
else
    System.out.println("Not equal using ==");
```

This code will produce the output

```
Not equal using ==
```

Even though these two variables name objects that are intuitively equal, they are stored in two different locations in the computer's memory. This is why you usually use an `equals` method to compare objects of a class type. The variables `variable1` and `variable2` would be considered "equal" if compared using the `equals` method as defined for the class `ToyClass` (Display 5.6).

Self-Test Exercises

24. What is a reference type? Are class types reference types? Are primitive types (like `int`) reference types?

25. When comparing two objects of a class type to see if they are "equal" or not, should you use == or the method `equals`?

26. When comparing two objects of a primitive type (like `int`) to see if they are "equal" or not, should you use == or the method `equals`?
27. Can a method with an argument of a class type change the values of the instance variables in the object named by the argument? For example, if the argument is of type `ToyClass` defined in Display 5.6, can the method change the `name` of its argument?
28. Suppose a method has a parameter of type `int` and the method is given an `int` variable as an argument. Could the method have been defined so that it changes the value of the variable given as an argument?

THE CONSTANT null

null

The constant `null` is a special constant that may be assigned to a variable of any class type. It is used to indicate that the variable has no "real value." If the compiler insists that you initialize a variable of a class type and there is no suitable object to initialize it with, you can use the value `null`, as in the following example:

```
YourClass yourObject = null;
```

It is also common to use `null` in constructors to initialize instance variables of a class type when there is no obvious object to use. We will eventually see other uses for the constant `null`.

Note that `null` is not an object. It is like a reference (memory address) that does not refer to any object (does not name any memory location). So, if you want to test whether a class variable contains `null`, you use == or `!=`; you do not use an `equals` method. For example, the following correctly tests for `null`:

```
if (yourObject == null)
    System.out.println("No real object here.");
```

null

`null` is a special constant that can be used to give a value to any variable of any class type. The constant `null` is not an object but a sort of placeholder for a reference to an object. Because it is like a reference (memory address), you use == and != rather than the method `equals` when you test to see whether a variable contains `null`.

JOptionPane REVISITED

One example of the use of `null` involves `JOptionPane`. In Chapter 2 we noted that `ShowInputDialogJOptionPane` produced a window with a text field and two buttons labeled `OK` and `Cancel`, but we did not fully describe what happens if the user clicks the `Cancel` button. For example, consider execution of the following:

```
String podString =
   JOptionPane.showInputDialog("Enter number of pods:");
```

If the user enters the string `"100"` in the text window and clicks the `OK` button, then the value of `podString` is set to `"100"`. If the user clicks the `Cancel` button, then `showInputDialog` returns `null`, so the value of `podString` is set to `null`. This works out well because `null` is different from any real `String` and yet can be stored in a variable of type `String` (or any other class type). So, in this case `null` means something like "No string was entered." You can use a test for this `null` value to program an action for the `Cancel` button, as illustrated by the following very simple example (which is only part of a program):

```
String podString =
   JOptionPane.showInputDialog("Enter number of pods:");
int numberOfPods;
if (podString == null) //if the user clicks Cancel
{
    String message = "I guess you don't know the number of pods.\n"
                    + "So, I'll have to end this program.";
    JOptionPane.showMessageDialog(null, message );
    System.exit(0);
}
else
    numberOfPods = Integer.parseInt(podString);
```

The following use of `null` (taken from the piece of code we just saw) is another example of how `null` is used:

```
JOptionPane.showMessageDialog(null, message );
```

In this case `null` is just a placeholder. In some situations (which we have not yet discussed), the first argument would be a real object. In this case we need no real object for the first argument, so we use `null` to fill that argument position.

Pitfall

NULL POINTER EXCEPTION

If the compiler asks you to initialize a class variable, you can always initialize the variable to `null`. However, `null` is not an object, so you cannot invoke a method using a variable that is initialized to `null`. If you try, you will get an error message that says "Null Pointer Exception." For example, the following code would produce a "Null Pointer Exception" if it were included in a program:

```
ToyClass aVariable = null;
String representation = aVariable.toString();
```

The problem is that you are trying to invoke the method `toString()` using `null` as a calling object. But, `null` is not an object; it is just a placeholder. So, `null` has no methods. Since you are using `null` incorrectly, the error message reads "Null Pointer Exception." You will get this error message any time a class variable has not be assigned a (reference to an) object, even if you have

not assigned `null` to the variable. Any time you get a "Null Pointer Exception," look for an uninitialized class variable.

The way to correct the problem is to use new to create a class object, as follows:

```
ToyClass aVariable = new ToyClass("Joe", 42);
String representation = aVariable.toString( );
```

THE new OPERATOR AND ANONYMOUS OBJECTS

Consider an expression such as the following, where `ToyClass` is defined in Display 5.6:

```
ToyClass variable1 = new ToyClass("Joe", 42);
```

As illustrated in Display 5.8, the portion `new ToyClass("Joe", 42)` is an invocation of a constructor and you can think of the constructor as returning a reference to the location in memory of the object created by the constructor. If you take this view, then the equal sign in this line of code is just an ordinary assignment operator.

There are times when you create an object using `new` and use the object as an argument to a method, but then never again use the object. In such cases, you need not give the object a variable name. You can instead use the expression with the `new` operator and the constructor directly as the argument. For example, suppose you wanted to test to see if the object in `variable1` (in the earlier line of code) is equal to an object with the same number and with the name spelled in all uppercase letters. You can do so as follows:

```
if (variable1.equals(new ToyClass("JOE", 42)))
    System.out.println("Equal");
else
    System.out.println("Not equal");
```

This is equivalent to the following:

```
ToyClass temp = new ToyClass("JOE", 42);
if (variable1.equals(temp))
    System.out.println("Equal");
else
    System.out.println("Not equal");
```

In the second version the object is created and its reference is placed in the variable `temp`. Then `temp` is plugged in for the parameter in the `equals` method. But all the parameter passing mechanism does is to take the reference stored in `temp` and plug it in to the parameter for `equals`. The first version simplifies the process. It creates the reference to the object and directly plugs it in to the parameter in `equals`. It bypasses the variable `temp` but ends up plugging in the same reference as the argument to `equals`.

An expression such as

```
new ToyClass("JOE", 42)
```

anonymous object

when not assigned to a variable is known as an **anonymous object**. It evaluates to a reference to an object of the class. It is called *anonymous* because the object is not assigned a variable to serve as its name. We will eventually encounter situations where the use of such anonymous objects is common.

Anonymous Objects

An expression with a new operator and a constructor creates a new object and returns a reference to the object. If this reference is not assigned to a variable, but instead the expression with new and the constructor is used as an argument to some method, then the object produced is called an **anonymous object**.

Example:

```
if (variable1.equals(new ToyClass("JOE", 42)))
    System.out.println("Equal");
else
    System.out.println("Not equal");
```

The expression `new ToyClass("JOE", 42)` (or more exactly the object it creates) is an example of an anonymous object.

Self-Test Exercises

29. What is wrong with a program that starts as follows? The class `ToyClass` is defined in Display 5.6.

```
ToyClass anObject = null;
anObject.set("Chiana", 42);
```

30. What is the type of the constant `null`?

31. Suppose `aVariable` is a variable of a class type. Which of the following correctly tests to see if `aVariable` contains `null`?

```
aVariable.equals(null)
aVariable == null
```

32. Is the following legal in Java? The class `ToyClass` is defined in Display 5.6.

```
System.out.println(new ToyClass("Mr. Cellophane", 0));
```

5.3 Using and Misusing References

Loose lips sink ships.

Military slogan

Just as a military campaign requires constant vigilance to ensure that its plans are kept secret, so your programming requires constant vigilance to ensure that private instance variables remain truly private. As we will see, just adding the `private` modifier before instance variable declarations is not always all that you need to do. There can be privacy leaks in a poorly designed class just as there can be privacy leaks in a military campaign.

The material in this section is important but more subtle and harder to digest than the material we have seen before now. If you want, you may postpone reading this section until you have had more practice defining and using classes. You do not need the material in this section before doing Section 5.4.

Example

A Person Class

It is common to have instance variables of a class type. The class `Person` defined in Display 5.11 has two instance variables of type `Date`. So, the class `Person` has instance variables of a class type. (The class `Date` was defined in Chapter 4, Display 4.11. We have reproduced the relevant portions of the date class definition in Display 5.12.) In fact, all the instance variables for the class `Person` are of class types. An object of the class `Person` has the basic data about people that is found in such places as on tomb stones and in author listings in library catalogues. It describes a person by giving the person's name, date of birth, and date of death. If the person is still alive, then the value `null` is used as the date of death. (So, `null` is good.) A simple program illustrating the class `Person` is given in Display 5.13. We will here discuss a few details about the class `Person`, but will discuss most of the various methods in the class `Person` as we cover the corresponding topic in the following subsections.

Normally a class definition should include a no-argument constructor. However, there are cases where a no-argument constructor makes little sense. For example, the wrapper classes such as `Integer` and `Double` have no no-argument constructors, as we explained in the Pitfalls subsection "A Wrapper Class Does Not Have a No-Argument Constructor," which appeared earlier in this chapter. The class `Person` also has no no-argument constructor for a reason. A person may have no date of death, but a person always has a date of birth. A no-argument constructor should initialize all instance variables, but there is no suitable value to initialize the instance variable `born` unless it is provided as an argument to the constructor. In particular, it makes no sense to initialize the instance variable `born` to `null`; that would indicate that the person was never born. It makes little sense to have a person who was never born, so it makes little sense to have a no-argument constructor for the class `Person`. Note that because we defined some constructors for the class `Person` but did not define a no-argument constructor, it follows that the class `Person` has no no-argument constructor.

Display 5.11 A Person Class *(Part 1 of 5)*

```
1   /**
2    Class for a person with a name and dates for birth and death.
3    Class invariant: A Person always has a date of birth, and if the Person has a
4    date of death, then the date of death is equal to or later than the date of birth.
5   */
6   public class Person
7   {
8       private String name;
9       private Date born;
10      private Date died;//null indicates still alive.

11      public Person(String initialName, Date birthDate, Date deathDate)
12      {
13          if (consistent(birthDate, deathDate))
14          {
15              name = initialName;
16              born = new Date(birthDate);
17              if (deathDate == null)
18                  died = null;
19              else
20                  died = new Date(deathDate);
21           }
22           else
23           {
24               System.out.println("Inconsistent dates. Aborting.");
25               System.exit(0);
26           }
27      }

28      public Person(Person original)
29      {
30          if (original == null)
31          {
32              System.out.println("Fatal error.");
33              System.exit(0);
34          }

35          name = original.name;
36          born = new Date(original.born);

37          if (original.died == null)
38              died = null;
39          else
40              died = new Date(original.died);
41      }
```

The class Date was defined in Display 4.11 and many of the details are repeated in Display 5.12.

We will discuss `Date` and the significance of these constructor invocations in the subsection entitled "Copy Constructors."

Copy constructor

Display 5.11 A Person Class *(Part 2 of 5)*

```
42      public void set(String newName, Date birthDate, Date deathDate)
          <Definition of this method is Self Test Exercise 38.>

43      public String toString()
44      {
45          String diedString;
46          if (died == null)
47              diedString = ""; //Empty string
48          else
49              diedString = died.toString();

50          return (name + ", " + born + "-" + diedString);
51      }

52      public boolean equals(Person otherPerson)
53      {
54          if (otherPerson == null)
55              return false;
56          else
57              return (name.equals(otherPerson.name)
58                      && born.equals(otherPerson.born)
59                      && datesMatch(died, otherPerson.died) );
60      }

61      /**
62       To match date1 and date2 must either be the same date or both be null.
63      */
64      private static boolean datesMatch(Date date1, Date date2)
65      {
66          if (date1 == null)
67              return (date2 == null);
68          else if (date2 == null) //&& date1 != null
69              return false;
70          else // both dates are not null.
71              return(date1.equals(date2));
72      }

73      /**
74       Precondition: newDate is a consistent date of birth.
75       Postcondition: Date of birth of the calling object is newDate.
76      */
77      public void setBirthDate(Date newDate)
78      {
79          if (consistent(newDate, died))
80              born = new Date(newDate);
```

This is the `toString` method of the class `Date`. (line 49)

This is equivalent to `born.toString()`. (line 50)

This is the `equals` method for the class `String`. (line 57)

This is the `equals` method for the class `Date`. (line 58)

Display 5.11 A Person Class *(Part 3 of 5)*

```
 81           else
 82           {
 83               System.out.println("Inconsistent dates. Aborting.");
 84               System.exit(0);
 85           }
 86       }

 87       /**
 88        Precondition: newDate is a consistent date of death.
 89        Postcondition: Date of death of the calling object is newDate.
 90       */
 91       public void setDeathDate(Date newDate)
 92       {
 93
 94           if (!consistent(born, newDate))
 95           {
 96               System.out.println("Inconsistent dates. Aborting.");
 97               System.exit(0);
 98           }
 99
100           if (newDate == null)
101               died = null;
102           else
103               died = new Date(newDate);
104       }

105       public void setName(String newName)
106       {
107           name = newName;
108       }

109        /**
110        Precondition: The date of birth has been set, and changing the year
111        part of the date of birth will give a consistent date of birth.
112        Postcondition: The year of birth is (changed to) newYear.
113       */
114       public void setBirthYear(int newYear)
115       {
116           if (born == null) //Precondition is violated
117           {
118               System.out.println("Fata; Error. Aborting.");
119               System.exit(0);
120           }
121           born.setYear(newYear);
122           if (!consistent(born, died))
123           {
```

The date of death can be `null`. However, there is no corresponding code in `setBirthDate` because the method `consistent` ensures that the date of birth is never `null`.

Display 5.11 A Person Class *(Part 4 of 5)*

```
124                 System.out.println("Inconsistent dates. Aborting.");
125                 System.exit(0);
126             }
127         }

128         /**
129          Precondition: The date of death has been set, and changing the year
130          part of the date of death will give a consistent date of death.
131          Postcondition: The year of death is (changed to) newYear.
132         */
133         public void setDeathYear(int newYear)
134         {
135             if (died == null) //Precondition is violated
136             {
137                 System.out.println("Fata; Error. Aborting.");
138                 System.exit(0);
139             }
140             died.setYear(newYear);
141             if (!consistent(born, died))
142             {
143                 System.out.println("Inconsistent dates. Aborting.");
144                 System.exit(0);
145             }
146         }

147         public String getName()
148         {
149             return name;
150         }

151         public Date getBirthDate()
152         {
153             return new Date(born);
154         }

155         public Date getDeathDate()
156         {
157             if (died == null)
158                 return null;
159             else
160                 return new Date(died);
161          }

162         /**
163          To be consistent, birthDate must not be null. If there is no date of
164          death (deathDate == null), that is consistent with any birthDate.
165          Otherwise, the birthDate must come before or be equal to the deathDate.
166         */
```

Display 5.11 A Person Class *(Part 5 of 5)*

```
167     private static boolean consistent(Date birthDate, Date deathDate)
168     {
169         if (birthDate == null)
170             return false;
171         else if (deathDate == null)
172             return true;
173         else
174             return (birthDate.precedes(deathDate)
175                       || birthDate.equals(deathDate));
176     }
177 }
```

Since we are assuming that an object of the class `Person` always has a birth date (which is not `null`), the following should always be true of an object of the class `Person`:

> An object of the class `Person` has a date of birth (which is not `null`), and if the object has a date of death, then the date of death is equal to or later than the date of birth.

class invariant

If you check the definition of the class `Person`, you will see that this statement is always true. It is true of every object created by a constructor, and all the other methods preserve the truth of this statement. In fact, the private method `consistent` was designed to provide a check for this property. A statement, such as the above, which is always true for every object of the class is called a **class invariant**.

Class Invariant

A statement that is always true for every object of the class is called a **class invariant**. A class invariant can help to define a class in a consistent and organized way.

Note that the definition of `equals` for the class `Person` includes an invocation of `equals` for the class `String` and an invocation of the method `equals` for the class `Date`. Java determines which `equals` method is being invoked from the type of its calling object. Since the instance variable `name` is of type `String`, the invocation `name.equals(...)` is an invocation of the method `equals` for the class `String`. Since the instance variable `born` is of type `Date`, the invocation `born.equals(...)` is an invocation of the method `equals` for the class `Date`.

Similarly, the definition of the method `toString` for the class `Person` includes invocations of the method `toString` for the class `Date`.

Display 5.12 **The Class Date (Partial Definition)** ***(Part 1 of 2)***

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

6       public Date(String monthString, int day, int year)
7       {
8           setDate(monthString, day, year);
9       }

10      public Date(Date aDate)
11      {
12          if (aDate == null)//Not a real date.
13          {
14               System.out.println("Fatal Error.");
15               System.exit(0);
16          }

17          month = aDate.month;
18          day = aDate.day;
19          year = aDate.year;
20      }

21      public void setDate(String monthString, int day, int year)
22      {
23          if (dateOK(monthString, day, year))
24          {
25              this.month = monthString;
26              this.day = day;
27              this.year = year;
28          }
29          else
30          {
31              System.out.println("Fatal Error");
32              System.exit(0);
33          }
34      }
35      public String toString()
36              ...
37      public boolean equals(Date otherDate)
```

This is not a complete definition of the class `Date`*. The complete definition of the class* `Date` *is in Display 4.11, but this has the details that are important to what we are discussing in this chapter.*

Copy constructor

The method `dateOK` checks that the date is a legitimate date, such as not having more than 31 days.

<The complete definition of `equals` is given in the answer to Self-Test Exercise 34, and is a better version than the one given in Chapter 4.>

```
38      /**
39       Returns true if the calling object date is before otherDate (in time).
40      */
```

Display 5.12 The Class Date (Partial Definition) *(Part 2 of 2)*

```
41      public boolean precedes(Date otherDate)
42              ...
43      private boolean dateOK(int monthInt, int dayInt, int yearInt)
44              ...
45  }
```

These methods have the obvious meanings. If you need to see a full definition, see Display 4.11 in Chapter 4 and Self-Test Exercise 34 in this chapter.

Display 5.13 Demonstrating the Class Person *(Part 1 of 2)*

```
1   public class PersonDemo
2   {
3       public static void main(String[] args)
4       {
5           Person bach =
6               new Person("Johann Sebastian Bach",
7                   new Date("Mar", 21, 1685), new Date("Jul", 28, 1750));
8           Person stravinsky =
9               new Person("Igor Stravinsky",
10                  new Date("Jun", 17, 1882), new Date("Apr", 6, 1971));
11          Person adams =
12              new Person("John Adams",
13                  new Date("Feb", 15, 1947), null);

14          System.out.println("A Short List of Composers:");
15          System.out.println(bach);
16          System.out.println(stravinsky);
17          System.out.println(adams);

18          Person bachTwin = new Person(bach);
19          System.out.println("Comparing bach and bachTwin:");
20          if (bachTwin == bach)
21              System.out.println("Same reference for both.");
22          else
23              System.out.println("Distinct copies.");

24          if (bachTwin.equals(bach))
25              System.out.println("Same data.");
26          else
27              System.out.println("Not same data.");
28      }
29  }
```

Display 5.13 **Demonstrating the Class Person *(Part 2 of 2)***

Sample Dialogue

```
A Short List of Composers:
Johann Sebastian Bach, Mar 21, 1685–Jul 28, 1750
Igor Stravinsky, Jun 17, 1882–Apr 6, 1971
John Adams, Feb 15, 1947–
Comparing bach and bachTwin:
Distinct copies.
Same data.
```

Pitfall

null Can Be an Argument to a Method

If a method has a parameter of a class type, then null may be used as the corresponding argument when the method is invoked. Sometimes, using null as an argument can be the result of an error, but it can sometimes be an intentional argument. For example, the class Person (Display 5.11) uses null for a date of death to indicate that the person is still alive. So, null is sometimes a perfectly normal argument for methods such as consistent. Method definitions should account for null as a possible argument and not assume the method always receives a true object to plug in for a class parameter.

Notice the definition of the method equals for the class Person. A test for equality has the form

```
object1.equals(object2)
```

The calling object object1 must be a true object of the class Person; a calling object cannot be null. However, the argument object2 can be either a true object or null. If the argument is null, then equals should return false, since a true object cannot reasonably be considered to be equal to null. In fact, the Java documentation specifies that when the argument to an equals method is null, the equals method should return false. Notice that our definition does return false when the argument is null.

Self-Test Exercises

33. What is the difference between the following two pieces of code? The first piece appears in Display 5.13.

```
Person adams =
            new Person("John Adams",
                 new Date("Feb", 15, 1947), null);
```

```
//Second piece is below:

Date theDate = new Date("Feb", 15, 1947);
Person adams = new Person("John Adams", theDate, null);
```

34. When we defined the class `Date` in Chapter 4 (Display 4.11), we had not yet discussed `null`. So, the definition of `equals` given there did not account for the possibility that the argument could be `null`. Rewrite the definition of `equals` for the class `Date` to account for the possibility that the argument might be `null`.

COPY CONSTRUCTORS

copy constructor

A **copy constructor** is a constructor with a single argument of the same type as the class. The copy constructor should create an object that is a separate, independent object but with the instance variables set so that it is an exact copy of the argument object.

For example, Display 5.12 reproduces the copy constructor for the class `Date` defined in Display 4.11. The copy constructor, or any other constructor, creates a new object of the class `Date`. That part happens automatically and is not shown in the code for the copy constructor. The code for the copy constructor then goes on to set the instance variables to the values equal to those of its one parameter, `aDate`. But, the new date created is a separate object even though it represents the same date. Consider the following code:

```
Date date1 = new Date("Jan", 1, 2006);
Date date2 = new Date(date1);
```

After this code is executed, both `date1` and `date2` represent the date January 1, 2006, but they are two different objects. So, if we change one of these objects, it will not change the other. For example, consider

```
date2.setDate("Jul", 4, 1776);
System.out.println(date1);
```

The output produced is

```
Jan 1, 2006
```

When we changed `date2`, we did not change `date1`. This may not be a difficult or even subtle point, but it is critically important to much of what we discuss in this section of the chapter. (See Self-Test Exercise 36 in this chapter to see the copy constructor contrasted with the assignment operator.)

Now let's consider the copy constructor for the class `Person` (Display 5.11), which is a bit more complicated. It is reproduced in what follows:

```
public Person(Person original)
{
```

```
    if (original == null)
    {
        System.out.println("Fatal error.");
        System.exit(0);
    }
    name = original.name;
    born = new Date(original.born);
    if (original.died == null)
        died = null;
    else
        died = new Date(original.died);
}
```

We want the object created to be an independent copy of `original`. That would not happen if we had used the following instead:

```
public Person(Person original) //Unsafe
{
    if (original == null)
    {
        System.out.println("Fatal error.");
        System.exit(0);
    }
    name = original.name;
    born = original.born; //Not good.
    died = original.died; //Not good.
}
```

Although this alternate definition looks innocent enough and may work fine in many situations, it does have serious problems.

Assume we had used the unsafe version of the copy constructor instead of the one in Display 5.11. The "Not good." code simply copies references from `original.died` to the corresponding arguments of the calling object. So, the object created is not an independent copy of the `original` object. For example, consider the code

```
Person original =
       new Person("Natalie Dressed", new Date("Apr", 1, 1984), null);
Person copy = new Person(original);
copy.setBirthYear(1800);
System.out.println(original);
```

The output would be

```
Natalie Dressed, Apr 1, 1800-
```

When we changed the birth year in the object `copy` we also changed the birth year in the object `original`; because we are using our unsafe version of the copy constructor, both `original.born` and `copy.born` contain the same reference to the same `Date` object.

All this assumed that, contrary to fact, we have used the unsafe version of the copy constructor; fortunately, we used a safer version of the copy constructor that sets the `born` instance variables as follows:

```
born = new Date(original.born);
```

which is equivalent to

```
this.born = new Date(original.born);
```

This version, which we did use, makes the instance variable `this.born` an independent `Date` object that represents the same date as `original.born`. So if you change a date in the `Person` object created by the copy constructor, you will not change that date in the `original Person` object.

Note that if a class, such as `Person`, has instance variables of a class type, such as the instance variables `born` and `died`, then to define a correct copy constructor for the class `Person`, you must already have copy constructors for the class `Date` of the instance variables. The easiest way to ensure this for all your classes is to always include a copy constructor in every class you define.

Copy Constructor

A **copy constructor** is a constructor with one parameter of the same type as the class. A copy constructor should be designed so the object it creates is intuitively an exact copy of its parameter, but a completely independent copy. See Displays 5.11 and 5.12 for examples of copy constructors.

The Java documentation says to use a method named `clone` instead of a copy constructor, and, as you will see later in this book, there are situations where the copy constructor will not work as desired and you need the `clone` method. However, we do not yet have enough background to discuss the `clone` method. The `clone` method is discussed later in this book (Chapters 8 and 13). We will use both copy constructors and the `clone` method.

Self-Test Exercises

35. What is a copy constructor?

36. What output is produced by the following code?

```
Date date1 = new Date("Jan", 1, 2006);
Date date2;
date2 = date1;
date2.setDate("Jul", 4, 1776);
System.out.println(date1);
```

What output is produced by the following code? Only the third line is different from the previous case.

```
Date date1 = new Date("Jan", 1, 2006);
Date date2;
date2 = new Date(date1);
date2.setDate("Jul", 4, 1776);
System.out.println(date1);
```

37. What output is produced by the following code?

```
Person original =
          new Person("Natalie Dressed",
                        new Date("Apr", 1, 1984), null);
Person copy = new Person(original);
copy.setBirthDate( new Date("Apr", 1, 1800));
System.out.println(original)
```

Pitfall

Privacy Leaks

Consider the accessor method getBirthDate for the class Person (Display 5.11), which we reproduce in what follows:

leaking accessor methods

```
public Date getBirthDate()
{
    return new Date(born);
}
```

Do not make the mistake of defining the accessor method as follows:

```
public Date getBirthDate() //Unsafe
{
    return born; //Not good
}
```

Assume we had used the unsafe version of getBirthDate instead of the one in Display 5.11. It would then be possible for a program that uses the class Person to change the private instance variable born to any date whatsoever and bypass the checks in constructor and mutator methods of the class Person. For example, consider the following code, which might appear in some program that uses the class Person:

```
Person citizen = new Person(
  "Joe Citizen", new Date("Jan", 1, 1900), new Date("Jan", 1, 1990));
Date dateName = citizen.getBirthDate();
dateName.setDate("Apr", 1, 3000);
```

privacy leak

This code changes the date of birth so it is after the date of death (an impossibility in the universe as we know it). This citizen was not born until after he or she died! This sort of situation is known as a **privacy leak**, because it allows a programmer to circumvent the `private` modifier before an instance variable such as `born` and change the private instance variable to anything whatsoever.

The following code would be illegal in our program:

```
citizen.born.setDate("Apr", 1, 3000); //Illegal
```

This is illegal because `born` is a private instance variable. However, with the unsafe version of `getBirthDate` (and we are now assuming that we did use the unsafe version), the variable `dateName` contains the same reference as `citizen.born` and so the following is legal and equivalent to the illegal statement:

```
dateName.setDate("Apr", 1, 3000);//Legal and equivalent to illegal one.
```

It is as if you have a friend named Robert who is also known as Bob. So he is called both Robert and Bob. Some bully wants to beat up Robert, so you say "You cannot beat up Robert." The bully says "OK, I will not beat up Robert, but I will beat up Bob." Bob and Robert are two names for the same person. So, if you protect Robert but do not protect Bob, you have really accomplished nothing.

All this assumed that, contrary to fact, we have used the unsafe version of `getBirthDate`, which simply returns the reference in the private instance variable `born`. Fortunately, we used a safer version of `getBirthDate`, which has the following `return` statement:

```
return new Date(born);
```

This `return` statement does not return the reference in the private instance variable `born`. Instead it uses the copy constructor to return a reference to a new object that is an exact copy of the object named by `born`. If the copy is changed, that has no effect on the date whose reference is in the instance variable `born`. Thus, a privacy leak is avoided.

leaking mutator methods

Note that returning a reference is not the only possible source of privacy leaks. A privacy leak can also arise from an incorrectly defined constructor or mutator method. Notice the definition for the method `setBirthDate` in Display 5.11 and reproduced below:

```
public void setBirthDate(Date newDate)
{
    if (consistent(newDate, died))
        born = new Date(newDate);
    else
    {
        System.out.println("Inconsistent dates. Aborting.");
        System.exit(0);
    }
}
```

Note that the instance variable born is set to a copy of the parameter newDate. Suppose that instead of

```
born = new Date(newDate);
```

we had simply used

```
born = newDate;
```

And suppose we use the following code in some program:

```
Person personObject = new Person(
  "Josephine", new Date("Jan", 1, 2000), null);
Date dateName = new Date("Feb", 2, 2002);
personObject.setBirthDate(dateName);
```

where personObject names an object of the class Person. The following will change the year part of the Date object named by the born instance variable of the object personObject and will do so without going through the checks in the mutator methods for Person:

```
dateName.setYear(1000);
```

Since dateName contains the same reference as the private instance variable born of the object personObject, changing the year part of dateName changes the year part of the private instance variable born of personObject. Not only does this bypass the consistency checks in the mutator method setBirthDate, but it is also a likely source of an inadvertent change to the born instance variable.

If we define setBirthDate as we did in Display 5.11 and as shown below, this problem does not happen. (If you do not see this, go through the code step-by-step and trace what happens.)

```
public void setBirthDate(Date newDate)
{
    if (consistent(newDate, died))
        born = new Date(newDate);
    . . .
```

One final word of warning: Using copy constructors as we have been doing is not the officially sanctioned way to make copies of an object in Java. The officially sanctioned way to create copies of an object is to define a method named clone. We will discuss clone methods in Chapters 8 and 13. In Chapter 8 we show you that there are advantages to using a clone method instead of a copy constructor. In Chapter 13 we describe the officially sanctioned way to define the clone method. For what we will be doing until then, a copy constructor will be a very adequate way of creating copies of an object.

clone

MUTABLE AND IMMUTABLE CLASSES

Contrast the accessor methods `getName` and `getBirthDate` of the class `Person` (Display 5.11). We reproduce the two methods in what follows:

```
public String getName()
{
    return name;
}
```

```
public Date getBirthDate()
{
    return new Date(born);
}
```

Notice that the method `getBirthDate` does not simply return the reference in the instance variable `born`, but instead uses the copy constructor to return a reference to a copy of the birthdate object. We have already explained why we do this. If we return the reference in the instance variable `born`, then we can place this reference in a variable of type `Date`, and that variable could serve as another name for the private instance variable `born`, and that would allow us to violate the privacy of the instance variable `born` by changing it using a mutator method of the class `Date`. This is exactly what we discussed in the previous subsection. So, why didn't we do something similar in the method `getName`?

The method `getName` simply returns the reference in the private instance variable `name`. So, if we do the following in a program, then the variable `nameAlias` will be another name for the `String` object of the private instance variable `name`:

```
Person citizen = new Person(
  "Joe Citizen", new Date("Jan", 1, 1900), new Date("Jan", 1, 1990));
String nameAlias = citizen.getName();
```

It looks as though we could use a mutator method from the class `String` to change the name referenced by `nameAlias` and so violate the privacy of the instance variable `name`. Is something wrong? Do we have to rewrite the method `getName` to use the copy constructor for the class `String`? No, everything is fine. We cannot use a mutator method with `nameAlias` because the class `String` has no mutator methods! The class `String` contains no methods that change any of the data in a `String` object.

At first it may seem as though you can change the data in an object of the class `String`. What about the string processing we have seen, such as the following?

```
String greeting = "Hello";
greeting = greeting + " friend.";
```

Have we not changed the data in the `String` object from `"Hello"` to `"Hello friend."`? No, we have not. The expression `greeting + " friend."` does not change the object `"Hello"`; it creates a new object, so the assignment statement

```
greeting = greeting + " friend.";
```

replaces the reference to `"Hello"` with a reference to the different `String` object `"Hello friend."`. The object `"Hello"` is unchanged. To see that this is true, consider the following code:

```
String greeting = "Hello";
String helloVariable = greeting;
greeting = greeting + " friend.";
System.out.println(helloVariable);
```

This produces the output `"Hello"`. If the object `"Hello"` had been changed, the output would have been `"Hello friend."`.

A class that contains no methods (other than constructors) that change any of the data in an object of the class is called an **immutable class**, and objects of the class are called **immutable objects**. The class `String` is an immutable class. It is perfectly safe to return a reference to an immutable object, because the object cannot be changed in any undesirable way; in fact, it cannot be changed in any way whatsoever.

immutable class

A class that contains public mutator methods or other public methods, such as input methods, that can change the data in an object of the class is called a **mutable class**, and objects of the class are called **mutable objects**. The class `Date` is an example of a mutable class; many, perhaps most, of the classes you define will be mutable classes. As we noted in the subsection entitled "Privacy Leaks" (but using other words): You should never write a method that returns a mutable object, but should instead use a copy constructor (or other means) to return a reference to a (completely independent) copy of the mutable object.

mutable class

Tip

Deep Copy versus Shallow Copy

In the previous two subsections we contrasted the following two ways of defining the method `getBirthDate` (Display 5.11):

```
public Date getBirthDate()
{
    return new Date(born);
}

public Date getBirthDate() //Unsafe
{
    return born; //Not good
}
```

As we noted, the first definition is the better one (and the one used in Display 5.11). The first definition returns what is known as a *deep copy* of the object `born`. The second definition returns what is known as a *shallow copy* of the object `born`.

A **deep copy** of an object is a copy that, with one exception, has no references in common with the original. The one exception is that references to immutable objects are allowed to be shared (since immutable objects cannot change in any way and so cannot be changed in any undesirable way). For example, the first definition of `getBirthDate` returns a deep copy of the date stored by the instance variable `born`. So, if you change the object returned by `getBirthDate`, that will not change the `Date` object named by the instance variable `born`. The reason for this is that we defined the copy constructor for the class `Date` to create a deep copy (Display 5.12). Normally, copy constructors and accessor methods should return a deep copy.

Any copy that is not a deep copy is called a **shallow copy**. For example, the second definition of `getBirthDate` returns a shallow copy of the date stored by the instance variable `born`.

We will have more to say about deep and shallow copies in later chapters.

Never Return a Reference to a Mutable Private Object

A class that contains mutator methods or other methods, such as input methods, that can change the data in an object of the class is called a **mutable class**, and objects of the class are called **mutable objects**. When defining accessor methods (or almost any methods), your method should not return a reference to a mutable object. Instead, use a copy constructor (or other means) to return a reference to a (completely independent) copy of the mutable object.

Tip

Assume Your Coworkers Are Malicious

Our discussion of privacy leaks in the previous subsections was concerned about the effect of somebody trying to defeat the privacy of an instance variable. You might object that your coworkers are nice people and would not knowingly sabotage your software. That is probably true, and we do not mean to accuse your coworkers of malicious intent. However, the same action that can be performed intentionally by a malicious enemy can also be performed inadvertently by your friends or even by you yourself. The best way to guard against such honest mistakes is to pretend that you are defending against a malicious enemy.

Self-Test Exercises

38. Complete the definition of the method `set` for the class `Person` (Display 5.11).
39. Classify each of the following classes as either mutable or immutable: `Date` (Display 4.11), `Person` (Display 5.11), and `String`.
40. Normally it is dangerous to return a reference to a private instance variable of a class type, but it is okay if the class type is `String`. What's special about the class `String` that makes this okay?

5.4 Packages and javadoc

...he furnished me,
From mine own library with volumes that
I prize above my dukedom.

William Shakespeare, *The Tempest*

In this section we cover packages, which are Java libraries, and then cover the `javadoc` program, which automatically extracts documentation from packages and classes. Although these are important topics, they are not used in the rest of this book. So, you can cover this section at any time you wish; you need not cover this section before going on in this book.

This section does not use any of the material in Section 5.3 and so can be covered before Section 5.3.

This section assumes that you know about directories (which are called folders in some operating systems), that you know about path names for directories (folders), and that you know about PATH (environment) variables. These are not Java topics. They are part of your operating system, and the details depend on your particular operating system. If you can find out how to set the PATH variable on your operating system, you will know enough about these topics to understand this section.

PACKAGES AND `import` STATEMENTS

package

A **package** is Java's way of forming a library of classes. You can make a package from a group of classes and then use the package of classes in any other class or program you write without the need to move the classes to the directory (folder) in which you are working. All you need do is include an **`import` statement** that names the package. We have already used `import` statements with some predefined standard Java packages. For

`import` statement

example, the following, which we have used before, makes available the three classes `BufferedReader`, `InputStream`, and `IOException` of the package `java.io`:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
```

You can make all the classes in the package available by using the following instead:

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

There is no overhead cost for importing the entire package as opposed to just a few classes.

The `import` statements should be at the beginning of the file. Only blank lines, comments, and `package` statements may precede the list of `import` statements. We discuss `package` statements next.

`import` STATEMENT

You can use a class from a package in any program or class definition by placing an `import` statement that names the package and class at the start of the file containing the program or class definition. The program or class need not be in the same directory as the classes in the package.

SYNTAX:

```
import Package_Name.Class_Name;
```

EXAMPLES:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
```

You can import all the classes in a package by using an asterisk in place of the class's name.

SYNTAX:

```
import Package_Name.*;
```

EXAMPLE:

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

To make a package, you group all the classes together into a single directory (folder) and add the following `package` statement to beginning of each class file:

```
package Package_Name;
```

This `package` statement should be at the beginning of the file. Only blank lines and comments may precede the `package` statement. If there are both `import` statements and `package` statements, any `package` statements come before the `import` statements. Aside from the addition of the `package` statement, class files are just as we have already described them. (It is technically only the `.class` files that must be in the package directory.)

PACKAGE

A **package** is a collection of classes that have been grouped together into a directory and given a package name. The classes in the package are each placed in a separate file, and the file is given the same name as the class, just as we have been doing all along. Each file in the package must have the following at the beginning of the file. Only comments and blank lines may precede this package statement.

SYNTAX:

```
package Package_Name;
```

EXAMPLES:

```
package utilities.numericstuff;
package java.io;
```

THE PACKAGE java.lang

The package `java.lang` contains classes that are fundamental to Java programming. These classes are so basic that the package is always imported automatically. Any class in `java.lang` does not need an `import` statement to make it available to your code. For example, the classes `Math` and `String` and the wrapper classes introduced earlier in this chapter are all in the package `java.lang`.

PACKAGE NAMES AND DIRECTORIES

A package name is not an arbitrary identifier. It is a form of path name to the directory containing the classes in the package.

In order to find the directory for a package, Java needs two things: the name of the package and the value of your *CLASSPATH variable.*

You should already be familiar with the environment variable of your operating system that is known as the *PATH variable.* The **CLASSPATH variable** is a similar environment variable used to help locate Java packages. The value of your CLASSPATH variable tells Java where to begin its search for a package. The CLASSPATH variable is not a Java variable. It is an environment variable that is part of your operating system. The value of your CLASSPATH variable is a list of directories. The exact syntax for this

CLASSPATH variable

list varies from one operating system to another, but it should be the same syntax as that used for the (ordinary) PATH variable. When Java is looking for a package, it begins its search in the directories listed in the CLASSPATH variable.

The name of a package specifies the relative path name for the directory that contains the package classes. It is a relative path name because it assumes that you start in one of the directories listed in the value of your CLASSPATH variable. For example, suppose the following is a directory listed in your CLASSPATH variable (your operating system may use `/` instead of `\`):

```
\libraries\newlibraries
```

And suppose your package classes are in the directory

```
\libraries\newlibraries\utilities\numericstuff
```

In this case, the package *should* be named

```
utilities.numericstuff
```

and all the classes in the file must start with the package statement

```
package utilities.numericstuff;
```

The dot in the package name means essentially the same thing as the `\` or `/`, whichever symbol your operating system uses for directory paths. The package name tells you (and Java) what subdirectories to go through to find the package classes, starting from a directory on the class path. This is depicted in Display 5.14. (If there happen to be two directories in the class path variable that can be used, then of all the ones that can be used, Java always uses the first one listed in the CLASSPATH variable.)

Any class that uses the class in this `utilities.numericstuff` package must contain either the `import` statement

```
import utilities.numericstuff.*;
```

or an `import` statement for each class in the package that is used.

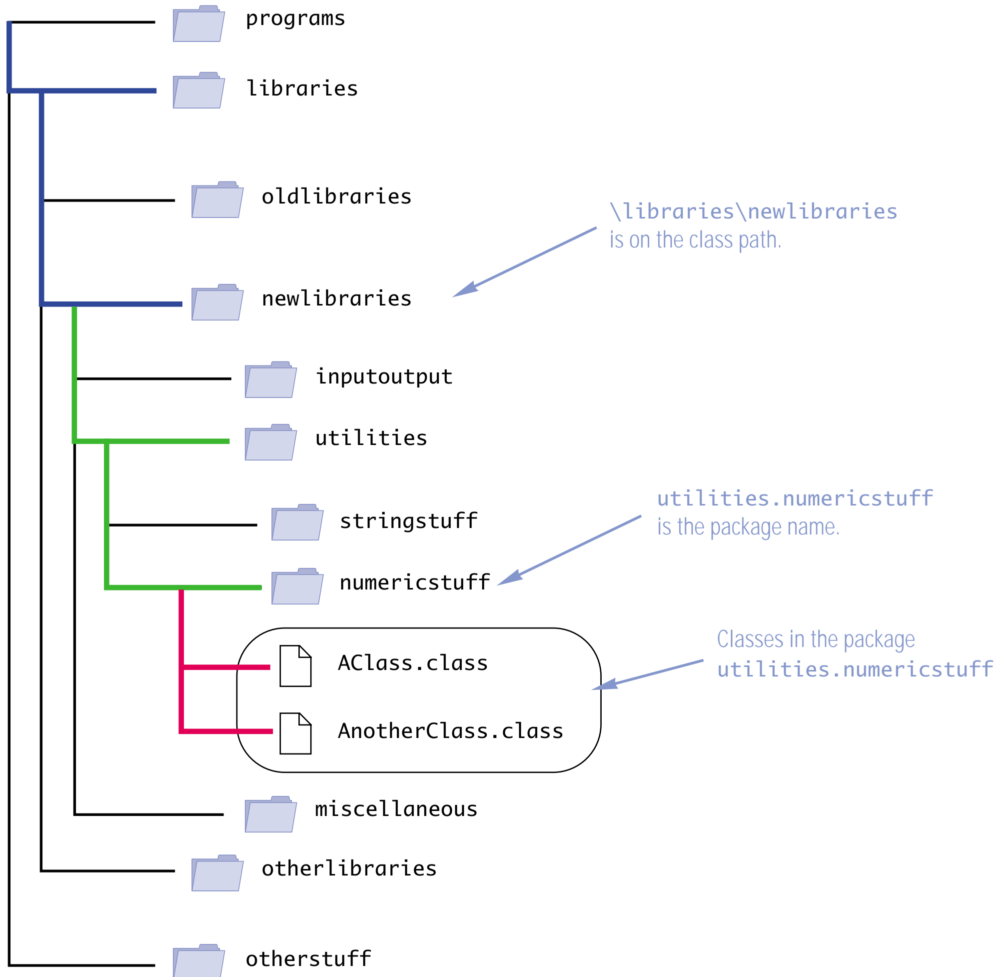
CLASSPATH

The way you set the value of your CLASSPATH variable depends on your operating system, but we can give you some suggestions that may work. The CLASSPATH variable is usually spelled as one word with all uppercase letters, as in `CLASSPATH`. You will probably have a plain old "PATH variable" that tells the operating system where to find the code for commands like `javac` and other commands that you can give as single-line commands. If you can find out how to set the PATH variable, you should be able to set the CLASSPATH variable in the same way.

If you are on a UNIX system, you are likely to be able to set the CLASSPATH with some command similar to the following :

```
set CLASSPATH=/libraries/newlibraries;/otherstuff/specialjava;.
export CLASSPATH
```

Display 5.14 **A Package Name**



If this does not work, you might try omitting the word `set` or replacing `set` with `setenv`. You might also try placing the list of directories in quotes. There are many versions of UNIX, all with their own minor variations. You may need to consult a local expert or check the documentation for your operating system.

If you are using a Windows machine, you can set the CLASSPATH variable by setting or creating an environment variable named `CLASSPATH` using the Control Panel.

Package Names and the CLASSPATH Variable

A package name must be a path name for the directory that contains the classes in the package, but the package name uses dots in place of \ or / (whichever your operating system uses). When naming the package, you use a relative path name that starts from any directory listed in the value of the CLASSPATH (environment) variable.

Examples:

```
utilities.numericstuff
java.io
```

In this book we are assuming that class path directories are separated by a semicolon. That is common, but some operating systems use some other separator, such as a colon. Check your documentation if a semicolon does not work as a separator.

Pitfall

Subdirectories Are Not Automatically Imported

Suppose you have two packages `utilities.numericstuff` and `utilities.numericstuff.statistical`. In this case, you know that `utilities.numericstuff.statistical` is in a subdirectory (subfolder) of the directory (folder) containing `utilities.numericstuff`. This leads some programmers to assume that the following `import` statement imports both packages:

```
import utilities.numericstuff.*;
```

This is not true. When you import an entire package, you do not import subdirectory packages.

To import all classes in both of these packages, you need

```
import utilities.numericstuff.*;
import utilities.numericstuff.statistical.*;
```

The Default Package

default package

All the classes in your current directory (that do not belong to some other package) belong to an unnamed package called the **default package**. As long as the current directory is on your CLASSPATH, all the classes in the default package are automatically available to your code. This is why we have always assumed that all the classes we defined are in the same directory. That way we need not clutter our discussion with concern about `import` statements.

Pitfall

NOT INCLUDING THE CURRENT DIRECTORY IN YOUR CLASS PATH

Your CLASSPATH variable will allow you to list more than one directory. Most operating systems use the dot to indicate the current directory. The **current directory** is not any one specific directory; it is the directory in which you are currently "located." If you do not know what your current directory is, then it is probably the directory that contains the class you are writing. For example, the following value for a CLASSPATH variable lists two ordinary directories and the current directory:

current directory

```
\libraries\newlibraries;\otherstuff\specialjava;.
```

Whenever you set or change the CLASSPATH variable, be sure to include the current directory as one of the alternatives. With the above displayed CLASSPATH value, if the package is not found by starting in either of the previous two directories, Java will look in the subdirectories of the current directory. If you want Java to check the current directory before the other directories on the CLASSPATH variable, list the current directory (the dot) first, as follows:

```
.;\libraries\newlibraries;\otherstuff\specialjava
```

When looking for a package, Java tries the directories in the class path in order and uses the first one that works.

Omitting the current directory from the CLASSPATH variable can interfere with running Java programs, regardless of whether or not the programs use packages. If the current directory is omitted, then Java may not even be able to find the `.class` file for the program itself, so you may not be able to run any programs at all. Thus, if you do set the CLASSPATH variable, it is critical that you include the current directory in the CLASSPATH. No such problems will occur if you have not set the CLASSPATH variable at all; it arises only if you decide to set the CLASSPATH variable.

If you are having problems setting the CLASSPATH variable, one interim solution is to delete the CLASSPATH variable completely and to keep all the class files for one program in the same directory. This will allow you to still do some work while you seek advice on setting the CLASSPATH variable.

SPECIFYING A CLASS PATH WHEN YOU COMPILE ✥

You can specify a class path when you compile a class. To do so, add `-classpath` followed by the class path as illustrated below:

```
javac -classpath .;C:\libraries\nemeric;C:\otherstuff YourClass.java
```

This will compile `YourClass.java`, overriding any CLASSPATH setting, and use the class path given after `-classpath`. Note that the directories are separated by semicolons. If you want classes in the current directory to be available to your class, then be sure the class path includes the current directory, which is indicated by a dot.

When you run the class compiled as just shown, you should again use the `-classpath` option as indicted below:

```
java -classpath .;C:\libraries\nemeric;C:\otherstuff YourClass
```

It is important to include the current directory on the class path when you run the program. If your program is in the default package, it will not be found unless you include the current directory. It is best to just get in the habit of always including the current directory in all class paths.

Since the class path specified in compiling and running your classes is input to a program (`javac` or `java`) that is part of the Java environment and is not a command to the operating system, you can use either / or \ in the class path, no matter which of these two your operating system uses.

NAME CLASHES ✜

name clash

In addition to being a way of organizing libraries, packages also provide a way to deal with *name clashes.* A **name clash** is a situation in which two classes have the same name. If different programmers writing different packages have used the same name for a class, the ambiguity can be resolved by using the package name.

Suppose a package named `sallyspack` contains a class called `HighClass` and that another package named `joespack` also contains a class named `HighClass`. You can use both classes named `HighClass` in the same program by using the more complete names `sallyspack.HighClass` and `joespack.HighClass`. For example:

```
sallyspack.HighClass object1 = new sallyspack.HighClass( );
joespack.HighClass object2 = new joespack.HighClass( );
```

fully qualified class name

These names that include the package name, like `sallyspack.HighClass` and `joespack.HighClass`, are called **fully qualified** class names.

If you use fully qualified class names, you do not need to import the class, since this longer class name includes the package name.

Self-Test Exercises

41. Suppose you want to use the class `CoolClass` in the package `mypackages.library1` in a program you write. What do you need to do to make this class available to your program? What do you need to do to make all the classes in the package available to your program?

42. What do you need to do to make a class a member of the package named `mypackages.library1`?

43. Can a package have any name you want, or are there restrictions on what you can use for a package name? Explain any restrictions.

INTRODUCTION TO javadoc ✜

The principles of encapsulation using information hiding say that you should separate the interface of a class (the instructions on how to use the class) from the implementation (the detailed code that tells the computer how the class does its work). In some other programming languages, such as C++, you normally define a class in two files. One file contains something like the interface or API that tells a programmer all that he or she needs to know to use the class. The other file contains the implementation details that are needed for the class code to run. This system is an obvious way to separate interface from implementation, but it is not what Java does.

Java does not divide a class definition into two files. Instead, Java has the interface and implementation of a class mixed together into a single file. If this were the end of the story, Java would not do a good job of encapsulation using information hiding. However, Java has a very good way of separating the interface from the implementation of a class. Java comes with a program named `javadoc` that will automatically extract the interface from a class definition. If your class definition is correctly commented, a programmer using your class need only look at this API (documentation) produced by `javadoc`. The documentation produced by `javadoc` is identical in format to the documentation for the standard Java library classes.

javadoc

The result of running `javadoc` on a class is to produce an HTML file with the API (interface) documentation for the class. HTML is the basic language used to produce documents to view with a Web browser, so the documentation produced by `javadoc` is viewed on a Web browser. A brief introduction to HTML is given in Chapter 17. However, you need not know any HTML to run `javadoc` or to read the documentation it produces.

`javadoc` can be used to obtain documentation for a single class. However, it is primarily intended to obtain documentation for an entire package.

We will first discuss how you should comment your classes so that you can get the most value out of `javadoc`. We will then describe how you run the `javadoc` program.

COMMENTING CLASSES FOR javadoc ✜

To get a more useful `javadoc` document, you must write your comments in a particular way. All the classes in this book have been commented for use with `javadoc`. However, to save space, the comments in this book are briefer than what would be ideal for `javadoc`.

The `javadoc` program will extract the heading for your class (or classes) as well as the headings for all public methods, public instance variables, public static variables, and certain comments. No method bodies and no private items are extracted when `javadoc` is run in the normal default mode.

For `javadoc` (in default mode) to extract a comment, the comment must satisfy two conditions:

1. The comment must *immediately precede* a public class definition, a public method definition, or other public item.
2. The comment must be a block comment (that is, the `/*` and `*/` style of comment), and the opening `/*` must contain an extra `*`. So the comment must be marked by `/**` at the beginning and `*/` at the end.

Unless you explicitly set an extra option to `javadoc`, line comments (that is, `//` style comments) will not be extracted, and comments preceding any private items also will not be extracted.

The comment that precedes a public method definition can include any general information you wish it to. There is also special syntax for inserting descriptions of parameters, any value returned, and any exceptions that might be thrown. We have not yet discussed exceptions. That is done in Chapter 9, but we include mention of them here so this section will serve as a more complete reference on `javadoc`. You need not worry about exceptions or the details about "throws" discussed here until you reach Chapter 9.

@ tag

The special information about parameters and so forth are preceded by the `@` symbol and are called **@ tags**. Below is an example of a method comment for use with `javadoc`:

```
/**
 Tests for equality of two objects of type Person. To be equal the two
 objects must have the same name, same birth date, and same death date.

 @param otherPerson The person being compared to the calling object.
 @return            Returns true if the calling object equals otherPerson.
*/
public boolean equals(Person otherPerson)
```

(The method `equals` is from the class `Person` defined in Display 5.11. If you need more context, look at that display.)

Note that the `@` tags all come after any general comment and that each `@` tag is on a line by itself. The following are some of the `@` tags allowed:

`@param` *Parameter_Name Parameter_Description*

`@return` *Description_Of_Value_Returned*

`@throws` *Exception_Type Explanation*

`@deprecated`

`@see` *Package_Name.Class_Name*

`@author` *Author*

`@version` *Version_Information*

The `@` tags should appear in the above order, first `@param`, then `@return`, then `@throws`, and so forth. If there are multiple parameters, they should each have their own `@param` and appear on a separate line. The parameters and their `@param` description should be

listed in their left-to-right order in the parameter list. If there are multiple authors, they should each have their own `@author` and appear on a separate line. The author and version information is not extracted unless suitable option flags have been set, as described in the next subsection.

If `@deprecated` is included in a method comment, then the documentation will note that the method is *deprecated.* A **deprecated** method is one that is being phased out. To allow for backward compatibility the method still works, but it should not be used in new code.

deprecated

If an `@` tag is included for an item, `javadoc` will extract the explanation for that item and include it in the documentation. You should always include a more or less complete set of `@` tags in the comment for each of your methods. In this book, we omit the `@` tags to save space, but we encourage you to always include them. The comments that are not part of an `@` tag will appear as a general comment for the method, along with the method heading.

You can also insert HTML commands in your comments so that you gain more control over `javadoc`, but that is not necessary and may not even be desirable. HTML commands can clutter the comments, making them harder to read when you look at the source code.

RUNNING javadoc ✜

To run `javadoc` on a package, all you need to do is give the following command:

```
javadoc -d Documentation_Directory Package_Name
```

It would be normal to run this command from the directory containing the classes in the package, but it can be run from any directory, provided you have correctly set the CLASSPATH environment variable. The *Documentation_Directory* is the name of the directory in which you want `javadoc` to place the HTML documents it produces. For example, the following might be used to obtain documentation on the package `mylibraries.numericstuff`:

```
javadoc -d documentation/mypackages mylibraries.numericstuff
```

The HTML documents produced will be in the subdirectory `documentation/mypackages` of where this command is run. If you prefer, you may use a complete path name in place of the relative path name `documentation/mypackages`. If you omit the `-d` and *Documentation_Directory*, `javadoc` will create suitable directories for the documentation.

You can link to standard Java documents so that your HTML documents include live links to standard classes and methods. The syntax is as follows:

```
javadoc -link Link_To_Standard_Docs -d Documentation_Directory Package_Name
```

Link_To_Standard_Docs is either a path to a local version of the Java documentation or the URL of the Sun Website with standard Java documentation. As of this writing, that URL is

```
http://java.sun.com/j2se/1.4/docs/api/
```

You need not run javadoc on an entire package. You can run javadoc on a single class file. For example, the following should be run from the directory containing Date.java and will produce documentation for the class Date:

```
javadoc Date.java
```

You can run javadoc on all classes in a directory with

```
javadoc *.java
```

You can add the -d and/or -link options to any of these commands. For example:

```
javadoc -link http://java.sun.com/j2se/1.4/docs/api/ -d mydocs *.java
```

These and other options for javadoc are summarized in Display 5.15.

extra code on CD

The result of running javadoc on ConsoleIn.java is given in the file ConsoleIn.html on the accompanying CD. Typically you get directories and many more HTML files than just the basic file like ConsoleIn.html. To get a better understanding of javadoc, you should try running javadoc in various settings and observe the files it produces.

Display 5.15 Options for javadoc

Option	Description
-link *Link_To_Other_Docs*	Provides a link to another set of documentation. Normally, this is used with either a path name to a local version of the Java documentation or the URL of the Sun Website with standard Java documentation.
-d *Documentation_Directory*	Specifies a directory to hold the documentation generated. *Documentation_Directory* may be a relative or absolute path name.
-author	Includes author information (from @author tags). This information is omitted unless this option is set.
-version	Includes version information (from @version tags). This information is omitted unless this option is set.
-classpath *List_of_Directories*	Overrides the CLASSPATH environment variable and makes *List_of_Directories* the CLASSPATH for the execution of this invocation of javadoc. Does not permanently change the CLASSPATH variable.
-private	Includes private members as well as public members in the documentation.

Self-Test Exercises

44. When run in default mode, does `javadoc` ever extract the body of a method definition? When run in default mode, does `javadoc` ever extract anything marked `private` in a class definition?

45. When run in default mode, what comments does `javadoc` extract and what comments does it not extract?

Chapter Summary

- A static method is one that does not require a calling object, but can use the class name in place of the calling object.
- A static variable is similar to an instance variable except that there is only one copy of the static variable that is used by all objects of the class.
- A wrapper class allows you to have a class object that corresponds to a value of a primitive type. Wrapper classes also contain a number of useful predefined constants and static methods.
- A variable of a class type stores the reference (memory address) of where the object is located in the computer's memory. This causes some operations, such as `=` and `==`, to behave quite differently for variables of a class type than they do for variables of a primitive type.
- When you use the assignment operator with two variables of a class type, the two variables become two names for the same object.
- A method cannot change the value of a variable of a primitive type that is an argument to the method. On the other hand, a method can change the values of the instance variables of an argument of a class type. This is because with class parameters it is a reference that is plugged in to the parameter.
- `null` is a special constant that can be used to give a value to any variable of any class type.
- An expression with a `new` operator and a constructor can be used as an argument to a method. Such an argument is called an *anonymous object*.
- A copy constructor is a constructor with one parameter of the same type as the class. A copy constructor should be designed so the object it creates is intuitively an exact copy of its parameter, but a completely independent copy, that is a deep copy.
- A class that contains mutator methods or any methods that can change the data in an object of the class is called a *mutable class*, and objects of the class are called *mutable objects*. When defining accessor methods (or almost any methods), your method should not return a reference to a mutable object. Instead, use a copy constructor (or other means) to return a reference to a deep copy of the mutable object.

- Packages are Java's version of class libraries.
- Java comes with a program named `javadoc` that will automatically extract the interface from all the classes in a package or from a single class definition.

Answers to Self-Test Exercises

1. Yes, it is legal, although it would be preferable style to use the class name `RoundStuff` in place of `roundObject`.
2. No, all methods in the class are static and so there is no need to create objects. If we follow our style rules, no constructors would ever be used and so there is no need for constructors.
3. Yes, a class can contain both static and nonstatic (that is, regular) methods.
4. You cannot invoke a nonstatic method within a static method (unless you create an object to serve as the calling object of the nonstatic method).
5. You can invoke a static method within a nonstatic method.
6. You cannot reference an instance variable within a static method, because a static method can be used without a calling object and hence without any instance variables.
7. Each object of a class has its own copy of each instance variable, but a single copy of each static variable is shared by all objects.
8. No, you cannot use an *instance variable* (without a class name and dot) in the definition of a *static method* of the same class. Yes, you can use an *instance variable* (without an object name and dot) in the definition of a *nonstatic method* of the same class.
9. Yes, you can use a *static variable* in the definition of a *static method* of the same class. Yes, you can use a *static variable* in the definition of a *nonstatic method* of the same class. So, you can use a static variable in any method.
10. No, you cannot use either an explicit or an implicit occurrence of the `this` parameter in the definition of a static method.
11. The following methods could and should be marked `static`: `dateOK`, `monthOK`, and `monthString`.
12. All static variables should be marked `private` with the exception of one case: If the static variable is used as a named constant (that is, if it is marked `final`), then it can be marked either `public` or `private` depending on the particular situation.
13. They can both be named by using the class name and a dot, rather than an object name and a dot.
14. `3, 4,`
    `3.0, 3.0,`
    `4.0, and 4.0`
15. `roundedAnswer = (int)Math.round(answer);`

16. `long`. Since one argument is of type `long`, the `int` argument is automatically type cast to `long`.

17. `Double.toString(result)`

18. `Double.parseDouble(stringForm)`

19. `Double.parseDouble(stringForm.trim())`

20.
```
System.out.println("Largest long is " + Long.MAX_VALUE +
                     " Smallest long is " + Long.MIN_VALUE);
```

21. `Character zeeObject = new Character('Z');`

22. No, none of the wrapper classes discussed in this chapter have no-argument constructors.

23. `Objects are Not equal.`

24. A reference type is a type whose variables contain references, that is, memory addresses. Class types are reference types. Primitive types are not reference types.

25. When comparing two objects of a class type, you should use the method `equals`.

26. When comparing two objects of a primitive type, you should you use ==.

27. Yes, a method with an argument of a class type can change the values of the instance variables in the object named by the argument.

28. No, a method cannot change the value of an `int` variable given as an argument to the method.

29. The variable `anObject` names no object and so the invocation of the `set` method is an error. One way to fix things is as follows:

```
ToyClass anObject = new ToyClass();
anObject.set("Chiana", 42);
```

30. The constant `null` can be assigned to a variable of any class type. It does not really have a type, but you can think of its type as being the type of a memory address. You can also think of `null` as being of every class type.

31. `aVariable == null`

32. It is unlikely but it is legal. This is an example of an anonymous object, as described in the text.

33. The only difference is that the object of type `Date` is given the name `theDate` in the second version. It makes no difference to the object `adams`.

34. This definition of `equals` is used in the file `Date.java` in the Chapter 5 directory of the source code on the CD:

extra code on CD

```
public boolean equals(Date otherDate)
{
    if (otherDate == null)
        return false;
    else
```

```
        return ( (month.equals(otherDate.month)) &&
            (day == otherDate.day) && (year == otherDate.year) );
}
```

35. A copy constructor is a constructor with one parameter of the same type as the class. A copy constructor should be designed so that the object it creates is intuitively an exact copy of its parameter, but a completely independent copy, that is a deep copy.

36. The first piece of code produces the output:

```
Jul 4, 1776
```

The second piece of code produces the output:

```
Jan 1, 2006
```

37. `Natalie Dressed, Apr 1, 1984-`

38.
```
public void set(String newName, Date birthDate, Date deathDate)
{
    if (consistent(birthDate, deathDate))
    {
            name = newName;
            born = new Date(birthDate);
            if (deathDate == null)
                died = null;
            else
                died = new Date(deathDate);
     }
     else
     {
         System.out.println("Inconsistent dates. Aborting.");
         System.exit(0);
     }
}
```

Note that the following is not a good definition since it could lead to a privacy leak:

```
public void set(String newName, Date birthDate, Date deathDate)
{//Not good
    name = newName;
    born = birthDate;
    died = deathDate;
}
```

39. The class `String` is an immutable class. The classes `Date` and `Person` are mutable classes.

40. The class `String` is an immutable class.

41. To make the class available to your program, you need to insert the following at the start of the file containing your class:

```
import mypackages.library1.CoolClass;
```

To make all the classes in the package available to your program, insert the following instead:

```
import mypackages.library1.*;
```

42. To make a class a member of the package named `mypackages.library1` you must insert the following at the start of the file with the class definition and place the file in the directory corresponding to the package (as described in the text):

```
package mypackages.library1;
```

(Only the `.class` file is required to be in the directory corresponding to the package, but it may be cleaner and easier to place both the `.java` file and the `.class` file there.)

43. A package name must be a path name for the directory that contains the classes in the package, but the package name uses dots in place of \ or / (whichever your operating system uses). When naming the package, you use a relative path name that starts from any directory named in the value of the CLASSPATH (environment) variable.

44. `javadoc` never extracts the body of a method definition, nor (when run in default mode) does `javadoc` ever extract anything marked `private` in a class definition.

45. When run in default mode, `javadoc` only extracts comments that satisfy the following two conditions:

    1. The comment must immediately precede a public class definition, a public method definition, or other public item.

    2. The comment must use the `/*` and `*/` style, and the opening `/*` must contain an extra `*`. So the comment must be marked by `/**` at the beginning and `*/` at the end. In particular, `javadoc` does not extract any `//` style comments.

Programming Projects

1. Define a class named `Money` whose objects represent amounts of U.S. money. The class will have two instance variables of type `int` for the dollars and cents in the amount of money. Include a constructor with two parameters of type `int` for the dollars and cents, one with one constructor of type `int` for an amount of dollars with zero cents, and a no-argument constructor. Include the methods `add` for addition and `minus` for subtraction of amounts of money. These methods will be static methods and will each have two parameters of type

`Money` and return a value of type `Money`. Include a reasonable set of accessor and mutator methods as well as the methods `equals` and `toString`. Write a test program for your class.

Part Two: Add a second version of the methods for addition and subtraction. These methods will have the same names as the static version but will use a calling object and a single argument. For example, this version of the `add` method (for addition) has a calling object and one argument. So, `m1.add(m2)` returns the result of adding the `Money` objects `m1` and `m2`. Note that your class will have all these methods; for example, there will be two methods named `add`.

Alternate Part Two: (If you want to do both Part Two and Alternate Part Two, they must be two classes. You cannot include the methods from both Part Two and Alternate Part Two in a single class. Do you know why?) Add a second version of the methods for addition and subtraction. These methods will have the same names as the static version but will use a calling object and a single argument. The methods will be `void` methods. The result is given as the changed value of the calling object. For example, this version of the `add` method (for addition) has a calling object and one argument. So,

```
m1.add(m2);
```

changes the values of the instance variables of `m1` so they represent the result of adding `m2` to the original version of `m1`. Note that your class will have all these methods; for example, there will be two methods named `add`.

2. Define a class for rational numbers. A rational number is a number that can be represented as the quotient of two integers. For example, 1/2, 3/4, 64/2, and so forth are all rational numbers. (By 1/2, etc., we mean the everyday meaning of the fraction, not the integer division this expression would produce in a Java program.) Represent rational numbers as two values of type `int`, one for the numerator and one for the denominator. So your class will have two instance variables of type `int`. Call the class `Rational`. Include a constructor with two arguments that can be used to set the instance variables of an object to any values. Also, include a constructor that has only a single parameter of type `int`; call this single parameter `wholeNumber` and define the constructor so that the object will be initialized to the rational number `wholeNumber`/1. Also, include a no-argument constructor that initializes an object to 0 (that is, to 0/1). Note that the numerator, the denominator, or both may contain a minus sign. Define methods for addition, subtraction, multiplication, and division of objects of your class `Rational`. These methods will be static methods that each have two parameters of type `Rational` and return a value of type `Rational`. For example, `Rational.add(r1, r2)` will return the result of adding the two rational numbers (two objects of the class `Rational`, `r1` and `r2`. Define accessor and mutator methods as well as the methods `equals` and `toString`. You should include a method to normalize the sign of the rational number so that the denominator is positive and the numerator is either positive or negative. For example, after normalization, 4/–8 would be represented the same as –4/8. Also write a test program to test your class.

   Hints: Two rational numbers a/b and c/d are equal if $a*d$ equals $c*b$.

Part Two: Add a second version of the methods for addition, subtraction, multiplication, and division. These methods will have the same names as the static version but will use a calling object and a single argument. For example, this version of the `add` method (for addition) has a calling object and one argument. So, `r1.add(r2)` returns the result of adding the rationals `r1` and `r2`. Note that your class will have all these methods; for example, there will be two methods named `add`.

Alternate Part Two: (If you want to do both Part Two and Alternate Part Two, they must be two classes. You cannot include the methods from both Part Two and Alternate Part Two in a single class. Do you know why?) Add a second version of the methods for addition, subtraction, multiplication, and division. These methods will have the same names as the static version but will use a calling object and a single argument. The methods will be `void` methods. The result is given as the changed value of the calling object. For example, this version of the `add` method (for addition) has a calling object and one argument. So,

```
r1.add(r2);
```

changes the values of the instance variables of `r1` so they represent the result of adding `r2` to the original version of `r1`. Note that your class will have all these methods; for example, there will be two methods named `add`.

3. Define a class for complex numbers. A complex number is a number of the form

```
a + b*i
```

where, for our purposes, *a* and *b* are numbers of type `double`, and *i* is a number that represents the quantity $\sqrt{-1}$. Represent a complex number as two values of type `double`. Name the instance variables `real` and `imaginary`. (The instance variable for the number that is multiplied by *i* is the one called `imaginary`.) Call the class `Complex`. Include a constructor with two parameters of type `double` that can be used to set the instance variables of an object to any values. Also, include a constructor that has only a single parameter of type `double`; call this parameter `realPart` and define the constructor so that the object will be initialized to `realPart + 0*i`. Also, include a no-argument constructor that initializes an object to `0` (that is, to `0 + 0*i`). Define accessor and mutator methods as well as the methods `equals` and `toString`. Define static methods for addition, subtraction, and multiplication of objects of your class `Complex`. These methods will be static and will each have two parameters of type `Complex` and return a value of type `Complex`. For example, `Complex.add(c1, c2)` will return the result of adding the two complex numbers (two objects of the class `Complex`) `c1` and `c2`. Also write a test program to test your class.

Hints: To add or subtract two complex numbers, you add or subtract the two instance variables of type `double`. The product of two complex numbers is given by the following formula:

```
(a + b*i)*(c + d*i) = (a*c - b*d) + (a*d + b*c)*i
```

Part Two: Add a second version of the methods for addition, subtraction, and multiplication. These methods will have the same names as the static version but will use a calling

object and a single argument. For example, this version of the `add` method (for addition) has a calling object and one argument. So, `c1.add(c2)` returns the result of adding the complex numbers `c1` and `c2`. Note that your class will have all these methods; for example, there will be two methods named `add`.

Alternate Part Two: (If you want to do both Part Two and Alternate Part Two, they must be two classes. You cannot include the methods from both Part Two and Alternate Part Two in a single class. Do you know why?) Add a second version of the methods for addition, subtraction, and multiplication. These methods will have the same names as the static version but will use a calling object and a single argument. The methods will be `void` methods. The result is given as the changed value of the calling object. For example, this version of the `add` method (for addition) has a calling object and one argument. So,

```
c1.add(c2);
```

changes the values of the instance variables of `c1` so they represent the result of adding `c2` to the original version of `c1`. Note that your class will have all these methods; for example, there will be two methods named `add`.