Most useful programs don’t just manipulate numbers and strings. Instead, they deal with data items that are more complex and that more closely represent entities in the real world. Examples of these data items include bank accounts, employee records, and graphical shapes.

The Java language is ideally suited for designing and manipulating such data items, or objects. In Java, you define classes that describe the behavior of these objects. In this chapter, you will learn how to define classes that describe objects with very simple behavior. As you learn more about Java programming in subsequent chapters, you will be able to implement classes whose objects carry out more sophisticated actions.
CHAPTER 2  An Introduction to Objects and Classes

CHAPTER CONTENTS

2.1 Using and Constructing Objects  34
Syntax  2.1: Object Construction  37
2.2 Object Variables  37
Syntax 2.2: Variable Definition  40
Syntax 2.3: Importing a Class from a Package  40
Common Error 2.1: Forgetting to Initialize Variables  41
Advanced Topic 2.1: Importing Classes  41
2.3 Defining a Class  42
Syntax 2.4: Method Implementation  43
Syntax 2.5: The return Statement  44
2.4 Testing a Class  44
Productivity Hint 2.1: Using the Command Line Effectively  46
2.5 Instance Fields  47
Syntax 2.6: Instance Field Declaration  49
2.6 Constructors  49
Syntax 2.7: Constructor Implementation  51
2.7 Designing the Public Interface of a Class  51
Advanced Topic 2.2: Overloading  53
2.8 Commenting the Public Interface  54
Productivity Hint 2.2: The Javadoc Utility  56
Productivity Hint 2.3: Keyboard Shortcuts for Mouse Operations  57
2.9 Specifying the Implementation of a Class  58
Common Error 2.2: Trying to Reset an Object by Calling a Constructor  62
HOWTO 2.1: Designing and Implementing a Class  62
2.10 Variable Types  66
Common Error 2.3: Forgetting to Initialize Object References in a Constructor  67
2.11 Explicit and Implicit Method Parameters  67
Common Error 2.4: Trying to Call a Method Without an Implicit Parameter  68
Advanced Topic 2.3: Calling One Constructor from Another  69
Random Fact 2.1: Mainframes—When Dinosaurs Ruled the Earth  70

2.1 Using and Constructing Objects

Objects and classes are central concepts for Java programming. It will take you some time to master these concepts fully, but since every Java program uses at least a couple of objects and classes, it is a good idea to have a basic understanding of these concepts right away.

An object is an entity that you can manipulate in your program, generally by calling methods. For example, you saw in Chapter 1 that System.out refers to an object, and you saw how to manipulate it by calling the println method. (Actually, several different methods are available, all called println: one for printing strings, one for printing integers, one for printing floating-point numbers, and so on; the reason is discussed in Section 2.8.) When you call the println method, some activities occur inside the object, and the ultimate effect is that the object causes text to appear in the console window.

You should think of the object as a “black box” with a public interface (the methods you can call) and a hidden implementation (the code and data that are necessary to make these methods work).

Different objects support different sets of methods. For example, you can apply the println method to the System.out object, but not to the string object "Hello, World!". That is, it would be an error to call "Hello, World!".println(); // This method call is an error

Objects are entities in your program that you manipulate by invoking methods.

The public interface of a class specifies what you can do with its objects. The hidden implementation describes how these actions are carried out.
The reason is simple. The System.out and "Hello, World!" objects belong to different classes. The System.out object is an object of the class PrintStream, but the "Hello, World!" object is an object of class String. You can apply the println method to any object of the PrintStream class, but the String class does not support the println method. The String class supports a good number of other methods; you will see many of them in Chapter 3. For example, the length method counts the number of characters in a string. You can apply that method to any object of type String. Thus,

"Hello, World!".length() // This method call is OK

is a correct method call—it computes the number of characters in the string object "Hello, World!" and returns the result, 13. (The quotation marks are not counted.) You can verify that the length method does return the length of a String object by writing a short test program

```java
public class LengthTest
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World!").length());
    }
}
```

Every object belongs to a class. The class defines the methods for the objects. Thus, the PrintStream class defines the print and println methods. The String class defines the length method and many other methods.

The System.out object is created automatically when a Java program loads the System class. String objects are created when you specify a string enclosed in quotation marks. However, in most Java programs, you want to create more objects.

To see how to create new objects, let us turn to another class: the Rectangle class in the Java class library. Objects of type Rectangle describe rectangular shapes—see Figure 1.

Note that a Rectangle object isn't a rectangular shape—it is a set of numbers that describe the rectangle (see Figure 2). Each rectangle is described by the $x$- and $y$-coordinates of its top left corner, its width, and its height. To make a new rectangle, you
need to specify these four values. For example, you can make a new rectangle with top left corner at (5, 10), width 20, and height 30 as follows:

    new Rectangle(5, 10, 20, 30)

The `new` operator causes the creation of an object of type `Rectangle`. The process of creating a new object is called **construction**. The four values 5, 10, 20, and 30 are called the **construction parameters**. Different classes will require different construction parameters. For example, to construct a `Rectangle` object, you supply four numbers that describe the position and size of the rectangle. To construct a `Car` object, you might supply the model name and year.

Actually, some classes let you construct objects in multiple ways. For example, you can also obtain a `Rectangle` object by supplying no construction parameters at all (but you must still supply the parentheses):

    new Rectangle()

This constructs a (rather useless) rectangle with top left corner at the origin (0, 0), width 0, and height 0.

To construct any object, you do the following:

1. Use the `new` operator.
2. Give the name of the class.
3. Supply construction parameters (if any) inside parentheses.

What can you do with a `Rectangle` object? Not much, for now. In Chapter 4, you will learn how to display rectangles and other shapes in a window. You can pass a rectangle object to the `System.out.println` or `print` method, which just prints a description of the rectangle object onto the console window:

    public class RectangleTest
    {
        public static void main(String[] args)
        {
            System.out.println(new Rectangle(5, 10, 20, 30));
        }
    }

This program prints the line

    java.awt.Rectangle[x=5,y=10,width=20,height=30]

More specifically, this program creates an object of type `Rectangle`, then passes that object to the `println` method. Afterward, that object is no longer used.
2.2 Object Variables

Of course, usually you want to do something more to an object than just create it, print it, and forget it. To remember an object, you need to hold it in an **object variable**. As was mentioned in Chapter 1, a variable is an item of information in memory whose location is identified by a symbolic name. An object variable is a container that stores the location of an object.

In Java, every variable has a particular **type** that identifies what kind of information it can contain. You create a variable by giving its type followed by a name for the variable. For example,

```java
Rectangle cerealBox;
```

This statement defines an object variable, `cerealBox`. The type of this variable is `Rectangle`. That is, after the `cerealBox` variable has been defined by the preceding statement, thereafter in the program it must always contain the location of a `Rectangle` object, never a `Car` or `String` object.

**Syntax 2.1: Object Construction**

```java
new ClassName (parameters)
```

**Example:**

```java
new Rectangle(5, 10, 20, 30)
new Car("BMW 540ti", 2004)
```

**Purpose:**

To construct a new object, initialize it with the construction parameters, and return a reference to the constructed object.

---

You store object locations in object variables.

---

**Figure 3**

An Uninitialized Object Variable

**Figure 4**

An Object Variable Containing an Object Reference
You can choose any variable names you like, provided you follow a few simple rules.

- Names can be made up of letters, digits, and the underscore (_) character. They cannot start with a digit, though.

- You cannot use other symbols such as ? or % in variable names.

- Spaces are not permitted inside names, either.

- Furthermore, you cannot use reserved words such as public as names; these words are reserved exclusively for their special Java meanings. Appendix A4 lists all reserved words.

- Variable names are also case-sensitive; that is, cerealBox and Cerealbox are different names.

Look again at the declaration of the cerealBox variable. So far, the variable is not initialized—it doesn’t yet contain any object location at all (see Figure 3). You need to set cerealBox to an object location. How do you get an object location? The new operator creates a new object and returns its location. Use that value to initialize the variable.

```java
Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
```

You may wonder what happens if you leave the cerealBox variable uninitialized. See Common Error 2.1 for an answer. Figure 4 shows the result.

An object location is also often called an object reference. When a variable contains the location of an object, we say that it refers to an object. For example, cerealBox refers to the Rectangle object that the new operator constructed.

It is very important that you remember that the cerealBox variable does not contain the object. It refers to the object. You can have two object variables refer to the same object:

```java
Rectangle r = cerealBox;
```

Now you can access the same Rectangle object both as cerealBox and as r, as shown in Figure 5.

Usually, your programs use objects in the following ways:

1. Construct an object with the new operator.
2. Store the object reference in an object variable.
3. Call methods on the object variable.

The Rectangle class has over 50 methods, some useful, some less so. To give you a flavor of manipulating Rectangle objects, let us look at a method of the Rectangle class. The translate method moves a rectangle by a certain distance in the x- and y-directions. For example,

```java
cerealBox.translate(15, 25);
```
moves the rectangle by 15 units in the $x$-direction and 25 units in the $y$-direction. Moving a rectangle doesn’t change its width or height, but it changes the top left corner. The code fragment

```java
Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
cerealBox.translate(15, 25);
System.out.println(cerealBox);
```

prints

```
java.awt.Rectangle[x=20,y=35,width=20,height=30]
```

Let’s turn this code fragment into a complete program. As with the Hello program, you need to carry out three steps:

1. Invent a new class, say `MoveTest`.
2. Supply a `main` method.
3. Place instructions inside the `main` method.

For this program, you need to carry out another step in addition to those: You need to `import` the `Rectangle` class from a package. A package is a collection of classes with a related purpose. All classes in the standard library are contained in packages. The `Rectangle` class belongs to the package `java.awt` (where `awt` is an abbreviation for “Abstract Windowing Toolkit”), which contains many classes for drawing windows and graphical shapes.

To use the `Rectangle` class from the `java.awt` package, simply place the following line at the top of your program:

```java
import java.awt.Rectangle;
```

Why didn’t you have to import the `System` and `String` classes that were used in the Hello program? The reason is that the `System` and `String` classes are in the `java.lang` package, and all classes from this package are automatically imported, so you never need to import them yourself.
Thus, the complete program is:

**File MoveTest.java**

```java
import java.awt.Rectangle;

public class MoveTest {
    public static void main(String[] args) {
        Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
        // move the rectangle
        cerealBox.translate(15, 25);
        // print the moved rectangle
        System.out.println(cerealBox);
    }
}
```

**Syntax 2.2: Variable Definition**

```
TypeName variableName;
TypeName variableName = expression;
```

Example:

```java
Rectangle cerealBox;
String name = "Dave";
```

Purpose:
To define a new variable of a particular type and optionally supply an initial value

**Syntax 2.3: Importing a Class from a Package**

```
import packageName.ClassName;
```

Example:

```java
import java.awt.Rectangle;
```

Purpose:
To import a class from a package for use in a program
2.2 Object Variables

**Forgetting to Initialize Variables**

You just learned how to store an object reference in a variable so that you can manipulate the object in your program. This is a very common step, and it can lead to one of the most common programming errors—using a variable that you forgot to initialize.

Suppose your program contains the lines

```java
Rectangle cerealBox;
cerealBox.translate(15, 25);
```

Now you have a variable `cerealBox`. An object variable is a container for an object reference. But you haven’t put anything into the variable—it is not initialized. Thus, there is no rectangle to translate.

The compiler spots these problems. If you make this mistake, the compiler will complain that you are trying to use an uninitialized variable.

The remedy is to initialize the variable. You can initialize a variable with any object reference, either a reference to a new object or an existing object.

```java
// initialize with new object reference
Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
// initialize with existing object reference
Rectangle cerealBox = anotherRectangle;
```

**Importing Classes**

You have seen the simplest and clearest method for importing classes from packages. Simply use an `import` statement that names the package and class for each class that you want to import. For example,

```java
import java.awt.Rectangle;
import java.awt.Point;
```

There is a shortcut that many programmers find convenient. You can import all classes from a package name with the construct

```java
import packagename.*;
```

For example, the statement

```java
import java.awt.*;
```

imports all classes from the `java.awt` package. This is less trouble to type, but we won’t use this style in this book, for a simple reason. If a program imports multiple packages
and you encounter an unfamiliar class name, then you have to look up all of those packages to find the class. For example, suppose you see a program that imports

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

Furthermore, suppose you see a class name `Image`. You would not know whether the `Image` class is in the `java.awt` package or the `java.io` package. Why do you care in which package it is? You need to know if you want to use the class in your own programs.

Note that you cannot import multiple packages with a single `import` statement. For example,

```java
import java.*.*;  // Error
```

is a syntax error.

You can avoid all `import` statements by using the full name (both package name and class name) whenever you use a class. For example,

```java
java.awt.Rectangle cerealBox =
    new java.awt.Rectangle(5, 10, 20, 30);
```

That is pretty tedious, and you won’t find many programmers doing it.

### 2.3 Defining a Class

In this section, you will learn how to define your own classes. Recall that a class defines the methods that you can apply to its objects. We will start with a very simple class that contains a single method.

```java
public class Greeter {
    public String sayHello() {
        String message = "Hello, World!";
        return message;
    }
}
```

A method definition contains the following parts:

- An access specifier (such as `public`)
- The return type of the method (such as `String`)
- The name of the method (such as `sayHello`)
- A list of the parameters of the method, enclosed in parentheses (the `sayHello` method has no parameters)
- The body of the method: a sequence of statements enclosed in braces

The access specifier controls which other methods can call this method. Most methods should be declared as `public`. That way, all other methods in your program can call them.
2.3 Defining a Class

(Occasionally, it can be useful to have methods that are not so widely callable—turn to Chapter 11 for more information on this issue.)

The return type is the type of the value that the method returns to its caller. The `sayHello` method returns an object of type `String` (namely, the string "Hello, World!").

Some methods just execute some statements without returning a value. Those methods are tagged with a return type of `void`.

Many methods depend on other information. For example, the `translate` method of the `Rectangle` class needs to know how far you want to move the rectangle horizontally and vertically. These items are called the parameters of the method. Each parameter is a variable, with a type and a name. Parameter variables are separated by commas. For example, the implementors of the Java library defined the `translate` method like this:

```java
public class Rectangle
{
    public void translate(int x, int y)
    {
        method body
    }
    . . .
}
```

**Syntax 2.4: Method Implementation**

```java
public class ClassName
{
    . . .
    accessSpecifier returnType methodName(parameterType parameterName, ...) 
    {
        method body
    }
    . . .
}
```

**Example:**

```java
public class Greeter
{
    public String sayHello()
    {
        String message = "Hello, World!";
        return message;
    }
}
```

**Purpose:**

To define the behavior of a method
The method body contains the statements that the method executes. The `sayHello` method body, for example, contains two statements. The first statement initializes a `String` variable with a `String` object:

```
String message = "Hello, World!";
```

The second statement is a special statement that terminates the method. When the `return` statement is executed, the method exits. If the method has a return type other than `void`, then the `return` statement must contain a return value, namely the value that the method sends back to its caller. The `sayHello` method returns the object reference stored in `message`—that is, a reference to the "Hello, World!" string object.

Now you have seen how to define a class that contains a method. In the next section, you will see what you can do with the class.

---

**Testing a Class**

In the preceding section, you saw the definition of a simple `Greeter` class. What can you do with it? Of course, you can compile the file `Greeter.java`. However, you can't execute the resulting `Greeter.class` file. It doesn't contain a `main` method. That is normal—most classes don't contain a `main` method.

To do something with your class, you have two choices. Some development environments, such as the excellent BlueJ program, let you create objects of a class and call methods on those objects. Figure 6 shows the result of creating an object of the `Greeter` class and invoking the `sayHello` method. The dialog box contains the return value of the method.

Alternatively, if you don't have a development environment that lets you test a class interactively, you can write a `test class`. A test class is a class with a `main` method that contains statements to test another class. A test class typically carries out the following steps:

1. Construct one or more objects of the class that is being tested.
2. Invoke one or more methods.
3. Print out one or more results.
2.4 Testing a Class

The RectangleTest class in Section 2.1 is a good example of a test class. That class tests the Rectangle class—a class in the Java library. Here is a class to test the Greeter class. The main method constructs an object of type Greeter, invokes the sayHello method, and displays the result on the console.

```java
public class GreeterTest {
    public static void main(String[] args) {
        Greeter worldGreeter = new Greeter();
        System.out.println(worldGreeter.sayHello());
    }
}
```

To produce a program, you need to combine these two classes. The details for building the program depend on your compiler and development environment. In most environments, you need to carry out these steps:

1. Make a new subfolder for your program.
2. Make two files, one for each class.
3. Compile both files.
4. Run the test program.
For example, if you use the Java SDK command line tools, the steps are like this:

```
mkdir greeter
  cd greeter
  edit Greeter.java
  edit GreeterTest.java
  javac Greeter.java
  javac GreeterTest.java
  java GreeterTest
```

Many students are surprised that such a simple program contains two classes. However, this is normal. The two classes have entirely different purposes. The `Greeter` class (which we will make more interesting in the next section) describes objects that can utter greetings. The `GreeterTest` class runs a test that puts a `Greeter` object through its paces. The `GreeterTest` program is necessary only if your development environment does not have a facility for interactive testing.

---

### Productivity Hint 2.1

**Using the Command Line Effectively**

- If your programming environment lets you accomplish all routine tasks with menus and dialog boxes, you can skip this note. However, if you need to invoke the editor, the compiler, the linker, and the program to test manually, then it is well worth learning about command line editing.

- Most operating systems (UNIX, DOS, OS/2) have a **command line interface** to interact with the computer. (In Windows, you can use the DOS command line interface by double-clicking the “MS-DOS Prompt” icon, or, if that icon doesn’t appear on your “Programs” menu, clicking “Run...” and typing `command.com`.) You launch commands at a **prompt**. The command is executed, and on completion you get another prompt.

- When you develop a program, you find yourself executing the same commands over and over. Wouldn’t it be nice if you didn’t have to type beastly commands like `javac MyProg.java` more than once? Or if you could fix a mistake rather than having to retype the command in its entirety? Many command line interfaces have an option to do just that, but they don’t always make it obvious. If you use Windows, you need to install a program called doskey. If you use UNIX, some shells let you cycle through your old commands. If your default configuration does not have that feature, ask how you can change to a better shell, such as `bash` or `tcsh`.

- Once you have your shell configured properly, you can use the up and down arrow keys to recall old commands and the left and right arrow keys to edit lines. You can also perform **command completion**. For example, to reissue the same `javac` command, type `javac` and press F8 (Windows) or type `!javac` (UNIX).
2.5 Instance Fields

Right now, our `Greeter` class isn’t very interesting, because all objects act in the same way. Suppose you construct two objects:

```
Greeter greeter1 = new Greeter();
Greeter greeter2 = new Greeter();
```

Then both `greeter1` and `greeter2` return exactly the same result when you invoke the `sayHello` method. Let’s modify the `Greeter` class so that one object can return the message "Hello, World!" and another can return "Hello, Dave!".

To achieve this purpose, each `Greeter` object must store `state`. The state of an object is the set of values that determine how an object reacts to method calls. In the case of our improved `Greeter` object, the state is the name that we want to use in the greeting, such as "World" or "Dave".

An object stores its state in one or more variables called `instance fields`. You declare the instance fields for an object in the class.

```
public class Greeter
{
    ...
    private String name;
}
```

An instance field declaration consists of the following parts:

- An `access specifier` (usually `private`)
- The `type` of the variable (such as `String`)
- The name of the variable (such as `name`)

Each object of a class has its own set of instance fields. For example, if `worldGreeter` and `daveGreeter` are two objects of the `Greeter` class, then each object has its own `name` field, called `worldGreeter.name` and `daveGreeter.name` (see Figure 7).

Instance fields are generally declared with the access specifier `private`. That specifier means that they can be accessed only by the methods of the `same class`, not by any other method. In particular, the `name` variable can be accessed only by the `sayHello` method.

In other words, if the instance fields are declared private, then all data access must occur through the public methods. Thus, the instance fields of an object are effectively hidden from the programmer who uses a class. They are of concern only to the programmer who implements the class. The process of hiding the data and providing methods for data access is called `encapsulation`. Although it is theoretically possible in Java to leave instance fields public, that is very uncommon in practice. We will always make all instance fields private in this book.
For example, because the name instance field is private, you cannot access the instance field in methods of another class:

```java
public class GreeterTest {
    public static void main(String[] args) {
        . . .
        System.out.println(daveGreeter.name); // ERROR
    }
}
```

Only the sayHello method can access the private name variable. If we later add other methods to the Greeter class, such as a sayGoodbye method, then those methods can access the private instance field as well.

Here is the implementation of the sayHello method of the improved Greeter class.

```java
public String sayHello() {
    String message = "Hello, " + name + "!";
    return message;
}
```

The + symbol denotes string concatenation, an operation that forms a new string by putting shorter strings together one after another.

This method computes a string message by combining three strings: "Hello, ", the string stored in the name instance field, and the string consisting of an exclamation
point "!". If the name variable refers to the string "Dave", then the resulting string is "Hello, Dave!".

Note that this method uses two separate object variables: the local variable message and the instance field name. A local variable belongs to an individual method, and you can use it only in the method in which you declare it. An instance field belongs to an object, and you can use it in all methods of its class.

To complete the improved Greeter class, we need to be able to construct objects with different values for the name instance field. We want to specify the name when constructing the object:

```java
Greeter worldGreeter = new Greeter("World");
Greeter daveGreeter = new Greeter("Dave");
```

To accomplish this, we need to supply a constructor in the class definition. A constructor specifies how an object should be initialized. In our example, we have one construction parameter—a string describing the name. Here is the code for the constructor.

```java
public Greeter(String aName) {
    name = aName;
}
```
A constructor always has the same name as the class of the objects it constructs. Similar to methods, constructors are generally declared as `public` to enable any code in a program to construct new objects of the class. Unlike methods, though, constructors do not have return types.

The `new` operator invokes the constructor:

```
new Greeter("Dave")
```

This expression constructs a new object whose `name` instance field is set to the string "Dave".

Constructors are not methods. You cannot invoke a constructor on an existing object. For example, the call

```
worldGreeter.Greeter("Harry"); // Error
```

is illegal. You can use a constructor only in combination with the `new` operator.

Here is the complete code for the enhanced `Greeter` class.

**File Greeter.java**

```java
public class Greeter {
    public Greeter(String aName) {
        name = aName;
    }

    public String sayHello() {
        String message = "Hello, " + name + "!";
        return message;
    }

    private String name;
}
```

Here is a test class that you can use to confirm that the `Greeter` class works correctly.

**File GreeterTest.java**

```java
public class GreeterTest {
    public static void main(String[] args) {
        Greeter worldGreeter = new Greeter("World");
        System.out.println(worldGreeter.sayHello());

        Greeter daveGreeter = new Greeter("Dave");
        System.out.println(daveGreeter.sayHello());
    }
}
```
The purpose of the `Greeter` class was to show you the mechanics of defining classes, methods, instance fields, and constructors. Frankly, that class was not very useful. In this section we will create a more interesting class that describes the behavior of a `BankAccount`. More importantly, we will go through the thought process that is required when you design a new class.

Before you start programming, you need to understand how the objects of your class behave. Consider what kind of operations you can carry out with a bank account. You can

- Deposit money
- Withdraw money
- Get the current balance

In Java, these operations are expressed as method calls. Let's suppose the variable `harrysChecking` contains a reference to an object of type `BankAccount`. You'll want to be able to call methods such as the following:

```java
harrysChecking.deposit(2000);
harrysChecking.withdraw(500);
System.out.println(harrysChecking.getBalance());
```
That is, the `BankAccount` class should define three methods:

- `deposit`
- `withdraw`
- `getBalance`

Next, you need to determine the parameters and return types of these methods. As you can see from the code samples, the `deposit` and `withdraw` methods receive a number (the dollar amount) and return no values. The `getBalance` method has no parameter and returns a number.

Java has several number types—you will learn about them in the next chapter. The most flexible number type is called `double`, which stands for “double precision floating-point number”. Think of a number in `double` format as any number that can appear in the display panel of a calculator, such as 250, 6.75, or -0.333333333.

Now that you know that you can use the `double` type for numbers, you can write down the methods of the `BankAccount` class:

```java
public void deposit(double amount)
public void withdraw(double amount)
public double getBalance()
```

Now let’s do the same for the constructors of the class. How do we want to construct a bank account? It seems reasonable that a call

```java
BankAccount harrysChecking = new BankAccount();
```

should construct a new bank account with a zero balance. What if we want to start out with another balance? A second constructor would be useful that sets the balance to an initial value:

```java
BankAccount harrysChecking = new BankAccount(5000);
```

That gives us two constructors:

```java
public BankAccount()
public BankAccount(double initialBalance)
```

The compiler figures out which constructor to call by looking at the parameters. For example, if you call

```java
new BankAccount()
```

then the compiler picks the first constructor. If you call

```java
new BankAccount(5000)
```

then the compiler picks the second constructor. But if you call

```java
new BankAccount("lotsa moolah")
```

then the compiler generates an error message—for this class there is no constructor that takes a parameter of type `String`.

You may think that it is strange to have two constructors that have the same name and that differ only in the parameter type. (The first constructor has no parameters; the second one has one parameter, a
number.) If a name is used to denote more than one constructor or method, that name is overloaded. See Advanced Topic 2.2 for more information on name overloading. Name overloading is common in Java, especially for constructors. After all, we have no choice what to call the constructor. The name of a constructor must be identical to the name of the class.

The constructors and methods of a class form the public interface of the class. These are the operations that any code in your program can access to create and manipulate BankAccount objects. Here is a complete listing of the public interface of the BankAccount class:

```java
public BankAccount()
public BankAccount(double initialBalance)
public void deposit(double amount)
public void withdraw(double amount)
public double getBalance()
```

The behavior of our BankAccount class is simple, but it lets you carry out all of the important operations that commonly occur with bank accounts. For example, here is how you can transfer an amount from one bank account to another:

```java
// transfer from one account to another
double transferAmount = 500;
momsSavings.withdraw(transferAmount);
harrysChecking.deposit(transferAmount);
```

And here is how you can add interest to a savings account:

```java
double interestRate = 5; // 5% interest
double interestAmount = momsSavings.getBalance() * interestRate / 100;
momsSavings.deposit(interestAmount);
```

As you can see, you can use objects of the BankAccount class to carry out meaningful tasks, without knowing how the BankAccount objects store their data or how the BankAccount methods do their work. This is an important aspect of object-oriented programming. The process of determining the feature set for a class is called abstraction. Think about how an abstract painting strips away extraneous details and tries to represent only the essential features of an object. When you design the public interface of a class, you also need to find what operations are essential to manipulate objects in your program.

---

**Advanced Topic**

**Overloading**

When the same name is used for more than one method or constructor, the name is overloaded. This is particularly common for constructors, because all constructors must have the same name—the name of the class. In Java you can overload methods and constructors, provided the parameter types are different. For example, the PrintStream class defines many methods, all called println, to print various number types and to print objects:

```java
class PrintStream
{
```
public void println(String s) {}
public void println(double a) {
    ...}

When the println method is called,
    system.out.println(x);
the compiler looks at the type of x. If x is a String value, the first method is called. If x
is a double value, the second method is called. If x does not match the parameter type of
any of the methods, the compiler generates an error.

For overloading purposes, the type of the return value does not matter. You cannot
have two methods with identical names and parameter types but different return values.

### Commenting the Public Interface

When you define classes and methods, you should get into the habit of thoroughly commenting their behavior. In Java there is a very useful standard form for documentation comments. If you use this form in your classes, a program called javadoc can automatically generate a neat set of HTML pages that describe them. (See Productivity Hint 2.2 for a description of this utility.)

A documentation comment starts with a /**, a special comment delimiter used by the javadoc utility. Then you describe the method’s purpose. Then, for each method parameter, you supply a line that starts with @param, followed by the parameter name and a short explanation. Finally, you supply a line that starts with @return, describing the return value. You omit the @param tag for methods that have no parameters, and you omit the @return tag for methods whose return type is void.

The javadoc utility copies the first sentence of each comment to a summary table. Therefore, it is best to write that first sentence with some care. It should start with an uppercase letter and end with a period. It does not have to be a grammatically complete sentence, but it should be meaningful when it is pulled out of the comment and displayed in a summary.

Here are two typical examples.

```java
/**
 * Withdraws money from the bank account.
 * @param amount the amount to withdraw
 */
public void withdraw(double amount) {
    implementation—filled in later
}
/**
 * Gets the current balance of the bank account.
 * @return the current balance
 */
public double getBalance() {
    implementation—filled in later
}
```
The comments you have just seen explain individual methods. You should also supply a brief comment for each class, explaining its purpose. The comment syntax for class comments is very simple: Just place the documentation comment above the class.

```java
/**
 * A bank account has a balance that can be changed by deposits and withdrawals.
 */
public class BankAccount
{
    . . .
}
```

Your first reaction may well be “Whoa! Am I supposed to write all this stuff?” These comments do seem pretty repetitive. But you should still take the time to write them, even if it feels silly at times. There are three reasons.

First, the javadoc utility will format your comments into a neat set of documents that you can view in a web browser. It makes good use of the seemingly repetitive phrases. The first sentence of the comment is used for a summary table of all methods of your class (see Figure 8). The @param and @return comments are neatly formatted in the detail description of each method (see Figure 9). If you omit any of the comments, then javadoc generates documents that look strangely empty.

Next, it is actually easy to spend more time pondering whether a comment is too trivial to write than it takes just to write it. In practical programming, very simple methods are rare. It is harmless to have a trivial method overcommented, whereas a complicated method without

---

**Figure 8**

A Method Summary Generated by javadoc
any comment can cause real grief to future maintenance programmers. According to the standard Java documentation style, every class, every method, every parameter, and every return value should have a comment.

Finally, it is always a good idea to write the method comment first, before writing the method code. This is an excellent test to see that you firmly understand what you need to program. If you can't explain what a class or method does, you aren't ready to implement it.

**Productivity Hint**

**The javadoc Utility**

You should always insert documentation comments in your code, whether or not you use javadoc to produce HTML documentation. But most people find the HTML documentation convenient, so it is worth learning how to run javadoc.

From a command shell, you invoke the javadoc utility with the command

```
javadoc MyClass.java
```

or

```
javadoc *.java
```
Commenting the Public Interface

The javadoc utility then produces files MyClass.html in HTML format, which you can inspect in a browser. If you know HTML (see Chapter 4), you can embed HTML tags into the comments to specify fonts or add images. Perhaps most importantly, javadoc automatically provides hyperlinks to other classes and methods.

You can actually run javadoc before implementing any methods. Just leave all the method bodies empty. Don’t run the compiler—it would complain about missing return values. Simply run javadoc on your file to generate the documentation for the public interface that you are about to implement.

The javadoc tool is wonderful because it does one thing right: It lets you put the documentation together with your code. That way, when you update your programs, you can see right away which documentation needs to be updated. Hopefully, you will update it right then and there. Afterward, run javadoc again and get updated information that is both timely and nicely formatted.

Productivity Hint

Keyboard Shortcuts for Mouse Operations

Programmers spend a lot of time with the keyboard and the mouse. Programs and documentation are many pages long and require a lot of typing. The constant switching among the editor, compiler, and debugger takes up quite a few mouse clicks. The designers of programs such as a Java integrated development environment have added some features to make your work easier, but it is up to you to discover them.

Just about every program has a user interface with menus and dialog boxes. Click on a menu and click on a submenu to select a task. Click on each field in a dialog box, fill in the requested answer, and click on the OK button. These are great user interfaces for the beginner, because they are easy to master, but they are terrible user interfaces for the regular user. The constant switching between the keyboard and the mouse slows you down. You need to move a hand off the keyboard, locate the mouse, move the mouse, click the mouse, and move the hand back onto the keyboard. For that reason, most user interfaces have keyboard shortcuts: combinations of keystrokes that allow you to achieve the same tasks without having to switch to the mouse at all.

All Microsoft Windows applications use the following conventions:

- The Alt key plus the underlined letter in a menu name (such as the F in “File”) pulls down that menu. Inside a menu, just type the underlined character in the name of a submenu to activate it. For example, Alt+F followed by O selects “File” “Open”. Once your fingers know about this combination, you can open files faster than the fastest mouse artist.

- Inside dialog boxes, the Tab key is important; it moves from one option to the next. The arrow keys move within an option. The Enter key accepts the entire dialog, and Esc cancels it.

- In a program with multiple windows, Ctrl+Tab usually toggles through the windows managed by that program, for example between the source and error window.
Alt+Tab toggles between applications, letting you toggle quickly between, for example, the text editor and a command shell window.

Hold down the Shift key and press the arrow keys to highlight text. Then use Ctrl+X to cut the text, Ctrl+C to copy it, and Ctrl+V to paste it. These keys are easy to remember. The V looks like an insertion mark that an editor would use to insert text. The X should remind you of crossing out text. The C is just the first letter in “Copy.” (OK, so it is also the first letter in “Cut”—no mnemonic rule is perfect.) You find these reminders in the Edit menu of most text editors.

Of course, the mouse has its use in text processing: to locate or select text that is on the same screen but far away from the cursor.

Take a little bit of time to learn about the keyboard shortcuts that the designers of your programs provided for you, and the time investment will be repaid many times during your programming career. When you blaze through your work in the computer lab with keyboard shortcuts, you may find yourself surrounded by amazed onlookers who whisper, “I didn’t know you could do that.”

Now that you understand the public interface of the BankAccount class, let’s provide the implementation. As you already know, you need to supply a class with these ingredients:

```
public class BankAccount {
    constructors
    methods
    fields
}
```

We have seen which constructors and methods we need. Let us turn to the instance fields. The instance fields are used to store the object state. In the case of our simple bank account objects, the state is the current balance of the bank account. (A more complex bank account might have a richer state—perhaps the current balance together with the interest rate paid, the date for mailing out the next statement, and so on.) For now, a single instance field suffices:

```
public class BankAccount {
    . . .
    private double balance;
}
```

Note that the instance field is declared with the access specifier private. That means, the bank balance can be accessed only by the constructors and methods of the same class—namely, deposit, withdraw, and getBalance—and not by any constructor or method of another class. How the state of a bank account is maintained is a private
implementation detail of the class. Recall that the practice of hiding the implementation details and providing methods for data access is called encapsulation.

The BankAccount class is so simple that it is not obvious what benefit you gain from the encapsulation. After all, you can always find out the current balance by calling the getBalance method. You can set the balance to any value by calling deposit with an appropriate amount.

The primary benefit of the encapsulation mechanism is the guarantee that an object cannot accidentally be put into an incorrect state. For example, suppose you want to make sure that a bank account is never overdrawn. You can simply implement the withdraw method so that it refuses to carry out a withdrawal that would result in a negative balance. (You will need to wait until Chapter 5 to see how to implement that protection.) On the other hand, if any code could freely modify the balance instance field of a BankAccount object, then it would be an easy matter to store a negative number in the variable.

Now that you know what methods you need, and how the object state is represented, it is an easy matter to implement each of the methods. For example, here is the deposit method:

```java
public void deposit(double amount)
{
    double newBalance = balance + amount;
    balance = newBalance;
}
```

Here is the constructor with no parameters.

```java
public BankAccount()
{
    balance = 0;
}
```

You will find the complete BankAccount class, with the implementations of all methods, at the end of this section.

If your development environment lets you construct objects interactively, then you can test this class immediately. Figures 10 and 11 show how to test the class in BlueJ. Otherwise, you need to supply a test class. The BankAccountTest class at the end of this section constructs a bank account, deposits and withdraws some money, and prints the remaining balance.

File BankAccount.java

```java
/**
   * A bank account has a balance that can be changed by
   * deposits and withdrawals.
   */
public class BankAccount
{
    /**
     * Constructs a bank account with a zero balance.
     */
    public BankAccount()
    {
        balance = 0;
    }
```
An Introduction to Objects and Classes

```java
/**
 * Constructs a bank account with a given balance.
 * @param initialBalance the initial balance
 */
public BankAccount(double initialBalance)
{
    balance = initialBalance;
}

/**
 * Deposits money into the bank account.
 * @param amount the amount to deposit
 */
public void deposit(double amount)
{
    double newBalance = balance + amount;
    balance = newBalance;
}
```

Figure 10
Calling the withdraw Method in BlueJ
2.9 Specifying the Implementation of a Class

Withdrawing money from the bank account.

@<br />
**
public void withdraw(double amount)
{
    double newBalance = balance - amount;
    balance = newBalance;
}

/**
 * Gets the current balance of the bank account.
 * @return the current balance
 */
public double getBalance()
{
    return balance;
}
File BankAccountTest.java

```java
1 /**
2 * A class to test the BankAccount class.
3 */
4 public class BankAccountTest
5 {
6     /**
7         Tests the methods of the BankAccount class.
8         @param args not used
9     */
10    public static void main(String[] args)
11    {
12        BankAccount harrysChecking = new BankAccount();
13        harrysChecking.deposit(2000);
14        harrysChecking.withdraw(500);
15        System.out.println(harrysChecking.getBalance());
16    }
17 }
```

Common Error 2.2

Trying to Reset an Object by Calling a Constructor

The constructor is invoked only when an object is first created. You cannot call the constructor to reset an object:

```java
BankAccount harrysChecking = new BankAccount();
harrysChecking.withdraw(500);
harrysChecking.BankAccount(); // Error—can't reconstruct object
```

The constructor sets a new account object to a zero balance, but you cannot invoke a constructor on an existing object. The remedy is simple: Make a new object and overwrite the current one.

```java
harrysChecking = new BankAccount(); // OK
```

Designing and Implementing a Class

This is the first of several “HOWTO” sections in this book. Users of the Linux operating system have HOWTO guides that give answers to the common questions “How do I get started?” and “What do I do next?” in solving a variety of problems. Similarly, the HOWTO sections in this book give you step-by-step procedures for carrying out specific tasks.

You will often be asked to design and implement a class. For example, a homework assignment might ask you to design a Car class.
Specifying the Implementation of a Class

1. **Find out what you are asked to do with an object of the class**

For example, suppose you are asked to implement a `Car` class. You won’t have to model every feature of a real car—there are too many. The assignment should tell you *which aspects* of a car your class should simulate. Make a list, in plain English, of the operations that an object of your class should carry out, such as this one:

- Add gas to gas tank.
- Drive a certain distance.
- Check the amount of gas left in the tank.

2. **Find names for the methods**

Come up with method names and apply them to a sample object, like this:

```
Car myBeemer = new Car(. . .);
myBeemer.addGas(20);
myBeemer.drive(100);
myBeemer.getGas();
```

3. **Document the public interface**

Here is the documentation, with comments that describe the class and its methods:

```
/**
 * A car can drive and consume fuel.
 */
public class Car
{
    /**
     * Adds gas to the tank.
     * @param amount the amount of fuel to add
     */
    public void addGas(double amount)
    {
    }

    /**
     * Drives a certain amount, consuming gas.
     * @param distance the distance driven
     */
    public void drive(double distance)
    {
    }

    /**
     * Gets the amount of gas left in the tank.
     * @return the amount of gas
     */
    public double getGas()
    {
    }
}
```
Step 4  Determine instance variables

Ask yourself what information an object needs to store to do its job. Remember, the methods can be called in any order! The object needs to have enough internal memory to be able to process every method, using just its instance fields and the method parameters. Go through each method, perhaps starting with a simple one or an interesting one, and ask yourself what you need to carry out the method's task. Make instance fields to store the information that the method needs.

In the car example, we need to know (or compute) the amount of gas in the tank—the getGas method asks for it. It makes sense for a Car object to store it:

```java
public class Car {
    private double gas;
}
```

Then the addGas method simply adds to that value. The drive method must reduce the gas in the tank. By how much? That depends on the fuel efficiency of the car. If you drive 100 miles, and the car can drive 20 miles per gallon, then 5 gallons are consumed. We don't get the efficiency as part of the drive method, so the car must store it:

```java
public class Car {
    private double gas;
    private double efficiency;
}
```

Step 5  Determine constructors

Ask yourself what you need to construct an object. Often, you can just set all fields to 0 or a constant value. Sometimes, you need some essential information. Then you need to set that information in a constructor. Sometimes you will want two constructors: one that sets all fields to a default and one that sets them to user-supplied values. Design constructors as needed.

In the case of the car example, we can start out with an empty gas tank, but we need the efficiency of the car. There is no good default for it, so it should be a construction parameter. It is common to prefix construction parameter names with "a" or "an" so that you don't conflict with instance variable names.

```java
/**
   Constructs a car with a given fuel efficiency.
   @param anEfficiency the fuel efficiency of the car
*/
public Car(double anEfficiency)
{
}
```

Step 6  Implement methods

Implement the methods and constructors in your class, one at a time, starting with the easiest ones. If you find that you have trouble with the implementation, you may need to
go back to a previous step. Maybe your set of methods from steps 1 through 3 wasn’t
good? Maybe you didn’t have the right instance fields? It is common for a beginner to
run into a couple of problems that require backtracking.

Compile your class and fix any compiler errors.

Step 7 Test your class

Write a short test program and run it. The test program can just carry out the method
calls that you found in step 2.

```
public class CarTest
{
    public static void main(String[] args)
    {
        Car myBeemer = new Car(20); // 20 miles per gallon
        myBeemer.addGas(20);
        myBeemer.drive(100);
        double gasLeft = myBeemer.getGas();
        System.out.println(gasLeft);
    }
}
```

Alternatively, if you use a program that lets you test objects interactively, such as BlueJ,
construct an object and apply the method calls (see Figure 12).

![Testing a Class with BlueJ](image)
You have seen three different types of variables in this chapter:

1. Instance fields, such as the `balance` variable of the `BankAccount` class
2. Local variables, such as the `newBalance` variable of the `deposit` method
3. Parameter variables, such as the `amount` variable of the `deposit` method

These variables are similar—they all hold a value of a particular type. But they have a couple of important differences. The first difference is their *lifetime*.

An instance field belongs to an object. Each object has its own copy of each instance field. For example, if you have two `BankAccount` objects (say, `harrysChecking` and `momsSavings`), then each of them has its own `balance` field. When an object is constructed, its instance fields are created. They stay alive until no method uses the object any longer.

Local and parameter variables belong to a method. When the method starts, these variables come to life. When the method exits, they die. For example, if you call

```java
harrysChecking.deposit(500);
```

then a parameter variable called `amount` is created and initialized with the parameter value, 500. When the method returns, that variable dies. When you make another method call,

```java
momsSavings.deposit(1000);
```

a different parameter variable, also called `amount`, is created. It too dies at the end of the method. The same holds for the local variable `newBalance`. When the `deposit` method reaches the line

```java
    double newBalance = balance + amount;
```

the variable comes to life and is initialized with the sum of the object's balance and the deposit amount. The lifetime of that variable extends to the end of the method. However, the `deposit` method has a lasting effect. Its next line,

```java
    balance = newBalance;
```

sets the `balance` instance field, and that variable lives beyond the end of the `deposit` method, as long as the `BankAccount` object is in use.

The second major difference between instance and local variables is *initialization*.

You must initialize all local variables. If you don't initialize a local variable, the compiler complains when you try to use it.

Parameter variables are initialized with the values that are supplied in the method call.

Instance fields are initialized with a default value if you don't explicitly set them in a constructor. Instance fields that are numbers are initialized to 0. Object references are set to a special value called `null`. If an object reference is `null`, then it refers to no object at all. We will discuss the `null` value in greater detail in Section 5.2. Inadvertent initialization with 0 or `null` is a common cause of errors. Therefore, it is a matter of good style to initialize every instance field explicitly in every constructor.
Forgetting to Initialize Object References in a Constructor

Just as it is a common error to forget to initialize a local variable, it is easy to forget about instance fields. Every constructor needs to ensure that all instance fields are set to appropriate values.

If you do not initialize an instance field, the Java compiler will initialize it for you. Numbers are initialized with 0, but object references—such as string variables—are set to the `null` reference.

Of course, 0 is often a convenient default for numbers. However, `null` is hardly ever a convenient default for objects. Consider this “lazy” constructor for the `Greeter` class:

```java
public class Greeter {
    public Greeter() {} // do nothing
    private String name;
}
```

The `name` field is set to a `null` reference. When you call `sayHello`, it will return "Hello, null!".

If you forget to initialize a local variable in a method, the compiler flags this as an error, and you must fix it before the program runs. If you make the same mistake with an instance field, the compiler provides a default initialization, and the error becomes apparent only when the program runs.

To avoid this problem, make it a habit to initialize every instance field in every constructor.

Explicit and Implicit Method Parameters

Have a look at a particular invocation of the `deposit` method:

```java
momsSavings.deposit(500);
```

Now look again at the code of the `deposit` method:

```java
public void deposit(double amount) {
    double newBalance = balance + amount;
    balance = newBalance;
}
```

The parameter variable `amount` is set to 500 when the `deposit` method starts. But what does `balance` mean exactly? After all, our program may have multiple `BankAccount` objects, and `each of them` has its own balance.
Of course, since we deposit the money into `momsSavings`, `balance` must mean `momsSavings.balance`. In general, when you refer to an instance field inside a method, it means the instance field of the object on which the method was called.

Thus, the call to the `deposit` method depends on two values: the object to which `momsSavings` refers, and the value 500. The `amount` parameter inside the parentheses is called an *explicit* parameter, because it is explicitly named in the method definition. However, the reference to the bank account object is not explicit in the method definition—it is called the *implicit parameter* of the method.

If you need to, you can access the implicit parameter—the object on which the method is called—with the keyword `this`. For example, in the preceding method invocation, `this` was set to `momsSavings` and `amount` to 500.

Every method has one implicit parameter. You don’t give the implicit parameter a name. It is always called `this`. (There is one exception to the rule that every method has an implicit parameter: static methods do not. We will discuss them in Chapter 7.) In contrast, methods can have any number of explicit parameters, which you can name any way you like, or no explicit parameter at all.

Next, look again closely at the implementation of the `deposit` method. The statement

```java
double newBalance = balance + amount;
```

actually means

```java
double newBalance = this.balance + amount;
```

When you refer to an instance field in a method, the compiler automatically applies it to the `this` parameter. Some programmers actually prefer to manually insert the `this` parameter before every instance field because they find it makes the code clearer. Here is an example:

```java
public void deposit(double amount)
{
    double newBalance = this.balance + amount;
    this.balance = newBalance;
}
```

You may want to try it out and see if you like that style.

---

**Common Error**

**2.4**

**Trying to Call a Method Without an Implicit Parameter**

Suppose your main method contains the instruction

```java
withdraw(30); // Error
```

The compiler will not know which account to access to withdraw the money. You need to supply an object reference of type `BankAccount`:
2.11 Explicit and Implicit Method Parameters

```java
BankAccount harrysChecking = new BankAccount();
harrysChecking.withdraw(30);
```

However, there is one situation in which it is legitimate to invoke a method without,
seemingly, an implicit parameter. Consider the following modification to the `BankAccount`
class. Add a method to apply the monthly account fee:

```java
class BankAccount {
    . . .
    public void monthlyFee() {
        withdraw(10); // withdraw $10 from this account
    }
}
```

That means to withdraw from the `same` account object that is carrying out the `monthlyFee`
operation. In other words, the implicit parameter of the `withdraw` method is the (invis-
ible) implicit parameter of the `monthlyFee` method.

If you find it confusing to have an invisible parameter, you can always use the `this`
parameter to make the method easier to read:

```java
class BankAccount {
    . . .
    public void monthlyFee() {
        this.withdraw(10); // withdraw $10 from this account
    }
}
```

Advanced Topic 2.3

**Calling One Constructor from Another**

Consider the `BankAccount` class. It has two constructors: a constructor without param-
eters to initialize the balance with zero, and another constructor to supply an initial bal-
ance. Rather than explicitly setting the balance to zero, one constructor can call another
constructor of the same class instead. There is a shorthand notation to achieve this result:

```java
class BankAccount {
    public BankAccount (double initialBalance) {
        balance = initialBalance;
    }
    public BankAccount () {
        this(0);
    }
    . . .
}
The command `this(0);` means “Call another constructor of this class and supply the value 0.” Such a constructor call can occur only as the first line in another constructor. This syntax is a minor convenience. We will not use it in this book. Actually, the use of the keyword `this` is a little confusing. Normally, `this` denotes a reference to the implicit parameter, but if `this` is followed by parentheses, it denotes a call to another constructor of this class.

**Random Fact**

**Mainframes—When Dinosaurs Ruled the Earth**

When the International Business Machines Corporation, a successful manufacturer of punched-card equipment for tabulating data, first turned its attention to designing computers in the early 1950s, its planners assumed that there was a market for perhaps 50 such devices, for installation by the government, the military, and a few of the country’s largest corporations. Instead, they sold about 1,500 machines of their System 650 model and went on to build and sell more powerful computers.

The so-called mainframe computers of the 1950s, 1960s, and 1970s were huge. They filled up whole rooms, which had to be climate-controlled to protect the delicate equipment (see Figure 13). Today, because of miniaturization technology, even mainframes are getting smaller, but they are still very expensive. (At the time of this writing, the cost for a midrange IBM 3090 is approximately 4 million dollars.)

**Figure 13**

A Mainframe Computer
These huge and expensive systems were an immediate success when they first appeared, because they replaced many roomfuls of even more expensive employees, who had previously performed the tasks by hand. Few of these computers do any exciting computations. They keep mundane information, such as billing records or airline reservations; they just keep lots of them.

IBM was not the first company to build mainframe computers; that honor belongs to the Univac Corporation. However, IBM soon became the major player, partially because of technical excellence and attention to customer needs and partially because it exploited its strengths and structured its products and services in a way that made it difficult for customers to mix them with those of other vendors. In the 1960s, IBM’s competitors, the so-called “Seven Dwarfs”—GE, RCA, Univac, Honeywell, Burroughs, Control Data, and NCR—fell on hard times. Some went out of the computer business altogether, while others tried unsuccessfully to combine their strengths by merging their computer operations. It was generally predicted that they would eventually all fail. It was in this atmosphere that the U.S. government brought an antitrust suit against IBM in 1969. The suit went to trial in 1975 and dragged on until 1982, when the Reagan Administration abandoned it, declaring it “without merit”.

Of course, by then the computing landscape had changed completely. Just as the dinosaurs gave way to smaller, nimbler creatures, three new waves of computers had appeared: the minicomputers, workstations, and microcomputers, all engineered by new companies, not the Seven Dwarfs. Today, the importance of mainframes in the marketplace has diminished, and IBM, while still a large and resourceful company, no longer dominates the computer market.

Mainframes are still in use today for two reasons. They still excel at handling large data volumes. More importantly, the programs that control the business data have been refined over the last 20 or more years, fixing one problem at a time. Moving these programs to less expensive computers, with different languages and operating systems, is difficult and error-prone. Sun Microsystems, a leading manufacturer of workstations, was eager to prove that its mainframe system could be “downsized” and replaced by its own equipment. Sun eventually succeeded, but it took over five years—far longer than it expected.

**Chapter Summary**

1. Objects are entities in your program that you manipulate by invoking methods.
2. The public interface of a class specifies what you can do with its objects. The hidden implementation describes how these actions are carried out.
3. Classes are factories for objects. You construct a new object of a class with the `new` operator.
4. You store object locations in object variables.
5. All object variables must be initialized before you access them.
6. An object reference describes the location of an object.
7. Multiple object variables can contain references to the same object.
8. Java classes are grouped into packages. If you use a class from another package (other
than the java.lang package), you must import the class.
9. A method definition specifies the method name, parameters, and the statements for
  carrying out the method’s actions.
10. Use the return statement to specify the value that a method returns to its caller.
11. To test a class, use an environment for interactive testing, or write a second class to
  execute test instructions.
12. An object uses instance fields to store its state—the data that it needs to execute its
  methods.
13. Each object of a class has its own set of instance fields.
14. Encapsulation is the process of hiding object data and providing methods for data access.
15. You should declare all instance fields as private.
16. Constructors contain instructions to initialize objects. The constructor name is always
  the same as the class name.
17. The new operator invokes the constructor.
18. Overloaded methods are methods with the same name but different parameter types.
19. Abstraction is the process of finding the essential feature set for a class.
20. Use documentation comments to describe the classes and public methods of your
  programs.
21. Provide documentation comments for every class, every method, every parameter,
  and every return value.
22. Instance fields belong to an object. Parameter variables and local variables belong to
  a method—they die when the method exits.
23. Instance fields are initialized to a default value, but you must initialize local
  variables.
24. The implicit parameter of a method is the object on which the method is invoked. The
  this reference denotes the implicit parameter.
25. Use of an instance field name in a method denotes the instance field of the implicit
  parameter.

**Review Exercises**

Exercise R2.1. Explain the difference between an object and an object reference.
Exercise R2.2. Explain the difference between an object and an object variable.
Exercise R2.3. Explain the difference between an object and a class.

Exercise R2.4. Explain the difference between a constructor and a method.

Exercise R2.5. Give the Java code for an object of class `BankAccount` and for an object variable of class `BankAccount`.

Exercise R2.6. Explain the difference between an instance field and a local variable.

Exercise R2.7. Explain the difference between a local variable and a parameter variable.

Exercise R2.8. Explain the difference between

```java
new BankAccount(5000);
```
and

```java
BankAccount b = new BankAccount(5000);
```

Exercise R2.9. Explain the difference between

```java
BankAccount b;
```
and

```java
BankAccount b = new BankAccount(5000);
```

Exercise R2.10. What are the construction parameters for a `BankAccount` object?

Exercise R2.11. Give Java code to construct the following objects:

- A rectangle with center (100, 100) and all side lengths equal to 50
- A greeter who will say "Hello, Mars!"
- A bank account with a balance of $5000

Create just objects, not object variables.

Exercise R2.12. Repeat the preceding exercise, but now define object variables that are initialized with the required objects.

Exercise R2.13. Find the errors in the following statements:

```java
Rectangle r = (5, 10, 15, 20);

double x = BankAccount(10000).getBalance();

BankAccount b;
b.deposit(10000);

b = new BankAccount(10000);
b.add("one million bucks");
```

Exercise R2.14. Describe all constructors of the `BankAccount` class. List all methods that can be used to change a `BankAccount` object. List all methods that don’t change the `BankAccount` object.

Exercise R2.15. What is the value of `b` after the following operations?

```java
BankAccount b = new BankAccount(10);
b.deposit(5000);
b.withdraw(b.getBalance() / 2);
```
Exercise R2.16. If $b_1$ and $b_2$ store objects of class $\text{BankAccount}$, consider the following instructions.

$$\begin{align*}
&b_1.\text{deposit}(b_2.\text{getBalance}()); \\
&b_2.\text{deposit}(b_1.\text{getBalance}());
\end{align*}$$

Are the balances of $b_1$ and $b_2$ now identical? Explain.

Exercise R2.17. What is the $\text{this}$ reference?

Exercise R2.18. What does the following method do? Give an example of how you can call the method.

```java
public class BankAccount
{
    public void mystery(BankAccount that, double amount)
    {
        this.balance = this.balance - amount;
        that.balance = that.balance + amount;
    }
    . . . // other bank account methods
}
```

Programming Exercises

Exercise P2.1. Write a program that constructs a $\text{Rectangle}$ object, prints it, and then translates and prints it three more times, so that, if the rectangles were drawn, they would form one large rectangle:

```
+-----+
|     |
+-----+
    |
+-----+
|     |
+-----+
    |
+-----+
```

Exercise P2.2. The $\text{intersection}$ method computes the $\text{intersection}$ of two rectangles—that is, the rectangle that is formed by two overlapping rectangles:
You call this method as follows:

```java
Rectangle r3 = r1.intersection(r2);
```

Write a program that constructs two rectangle objects, prints them, and then prints the rectangle object that describes the intersection. What happens when the rectangles do not overlap?

**Exercise P2.3.** Add a method `sayGoodbye` to the `Greeter` class.

**Exercise P2.4.** Add a method `refuseHelp` to the `Greeter` class. It should return a string such as "I am sorry, Dave. I am afraid I can't do that."

**Exercise P2.5.** Write a program that constructs a bank account, deposits $1000, withdraws $500, withdraws another $400, and then prints the remaining balance.

**Exercise P2.6.** Add a method

```java
void addInterest(double rate)
```

to the `BankAccount` class that adds interest at the given rate. For example, after the statements

```java
BankAccount momsSavings = new BankAccount(1000);
momsSavings.addInterest(10); // 10% interest
```

the balance in `momsSavings` is $1,100.

**Exercise P2.7.** Write a class `SavingsAccount` that is similar to the `BankAccount` class, except that it has an added instance variable `interest`. Supply a constructor that sets both the initial balance and the interest rate. Supply a method `addInterest` (with no explicit parameter) that adds interest to the account. Write a program that constructs a savings account with an initial balance of $1,000 and interest rate 10%. Then apply the `addInterest` method five times and print the resulting balance.

**Exercise P2.8.** Implement a class `Employee`. An employee has a name (a string) and a salary (a `double`). Write a default constructor, a constructor with two parameters (name and salary), and methods to return the name and salary. Write a small program that tests your class.

**Exercise P2.9.** Enhance the class in the preceding exercise by adding a method `raiseSalary(double byPercent)` that raises the employee's salary by a certain percentage. Sample usage:

```java
Employee harry = new Employee("Hacker, Harry", 55000);
harry.raiseSalary(10); // Harry gets a 10% raise
```

**Exercise P2.10.** Implement a class `Car` with the following properties. A car has a certain fuel efficiency (measured in miles/gallon or liters/km—pick one) and a certain amount of fuel in the gas tank. The efficiency is specified in the constructor, and the initial fuel level is 0. Supply a method `drive` that simulates driving the car for a certain distance, reducing the fuel level in the gas tank, and methods `getGas`, returning the current fuel level, and `addGas`, to tank up. Sample usage:

```java
Car myBeemer = new Car(29); // 29 miles per gallon
myBeemer.addGas(20); // tank 20 gallons
myBeemer.drive(100); // drive 100 miles
System.out.println(myBeemer.getGas()); // print fuel remaining
```
Exercise P2.11. Implement a class `Student`. For the purpose of this exercise, a student has a name and a total quiz score. Supply an appropriate constructor and methods `getName()`, `addQuiz(int score)`, `getTotalScore()`, and `getAverageScore()`. To compute the latter, you also need to store the number of quizzes that the student took.

Exercise P2.12. Implement a class `Product`. A product has a name and a price, for example `new Product("Toaster", 29.95)`. Supply methods `getName()`, `getPrice()`, and `setPrice()`. Write in a program that makes two products, prints the name and price, reduces their prices by $5.00, and then prints them again.

Exercise P2.13. Implement a class `Circle` that has methods `getArea()` and `getPerimeter()`. In the constructor, supply the radius of the circle.

Exercise P2.14. Implement a class `Square` that has methods `getArea()` and `getPerimeter()`. In the constructor, supply the width of the square.

Exercise P2.15. Implement a class `SodaCan` with methods `getSurfaceArea()` and `getVolume()`. In the constructor, supply the height and radius of the can.

Exercise P2.16. Implement a class `RoachPopulation` that simulates the growth of a roach population. The constructor takes the size of the initial roach population. The `waitForDoubling` method simulates a period in which the population doubles. The `spray` method simulates spraying with insecticide, which reduces the population by 10%. The `getRoaches` method returns the current number of roaches. Implement the class and a test program that simulates a kitchen that starts out with 10 roaches. Wait, spray, print the roach count. Repeat three times.

Exercise P2.17. Implement a class `RabbitPopulation` that simulates the growth of a rabbit population. The rules are as follows: Start with one pair of rabbits. Rabbits are able to mate at the age of one month. A month later, each female produces another pair of rabbits. Assume that rabbits never die and that the female always produces one new pair (one male, one female) every month from the second month on. Implement a method `waitAMonth` that waits for one month, and a method `getPairs` that prints the current number of rabbit pairs. Write a test program that shows the growth of the rabbit population for ten months. Hint: Keep one instance field for the newborn rabbit pairs and another one for the rabbit pairs that are at least one month old.