Chapter 3 Implementing Classes

CHAPTER GOALS

- To become familiar with the process of implementing classes
- To be able to implement simple methods
- To understand the purpose and use of constructors
- To understand how to access instance fields and local variables
- To appreciate the importance of documentation comments
- **G** To implement classes for drawing graphical shapes

In this chapter, you will learn how to implement your own classes. You will start with a given design that specifies the public interface of the class—that is, the methods through which programmers can manipulate the objects of the class. You then need to implement the methods. This step requires that you find a data representation for the objects, and supply the instructions for each method. You then provide a tester to validate that your class works correctly. You also document your efforts so that other programmers can understand and use your creation.

3.1 Levels of Abstraction

3.1.1 Black Boxes

When you lift the hood of a car, you will find a bewildering collection of mechanical components. You will probably recognize the motor and the tank for the wind-shield washer fluid. Your car mechanic will be able to identify many other components, such as the transmission and the electronic control module—the device that controls the timing of the spark plugs and the flow of gasoline into the motor. But ask your mechanic what is inside the electronic control module, and you will likely get a shrug.

It is a *black box*, something that magically does its thing. A car mechanic would never open the box—it contains electronic parts that can only be serviced at the factory. Of course, the device may have a color other than black, and it may not even be box-shaped. But engineers use the term "black box" to describe any device whose inner workings are hidden. Note that a black box is not totally mysterious. Its interaction with the outside world is well-defined. For example, the car mechanic can test that the engine control module sends the right firing signals to the spark plugs.

82

83

Why do car manufacturers put black boxes into cars? The black box greatly simplifies the work of the car mechanic, leading to lower repair costs. If the box fails, it is simply replaced with a new one. Before engine control modules were invented, gasoline flow into the engine was regulated by a mechanical device called a carburetor, a notoriously fussy mess of springs and latches that was expensive to adjust and repair.

Of course, for many drivers, the *entire car* is a "black box". Most drivers know nothing about its internal workings and never want to open the hood in the first place. The car has pedals, buttons, and a gas tank door. If you give it the right inputs, it does its thing, transporting you from here to there.

And for the engine control module manufacturer, the transistors and capacitors that go inside are black boxes, magically produced by an electronics component manufacturer.

In technical terms, a black box provides *encapsulation*, the hiding of unimportant details. Encapsulation is very important for human problem solving. A car mechanic is more efficient when the only decision is to test the electronic control module and to replace it when it fails, without having to think about the sensors and transistors inside. A driver is more efficient when the only worry is putting gas in the tank, not thinking about the motor or electronic control module inside.

However, there is another aspect to encapsulation. Somebody had to come up with the right *concept* for each particular black box. Why do the car parts manufacturers build electronic control modules and not another device? Why do the transportation device manufacturers build cars and not personal helicopters?

Concepts are discovered through the process of *abstraction*, taking away inessential features, until only the essence of the concept remains. For example, “car” is an abstraction, describing devices that transport small groups of people, traveling on the ground, and consuming gasoline. Is that the right abstraction? Or is a vehicle with an electric engine a “car”? We won't answer that question and instead move on to the significance of encapsulation and abstraction in computer science.

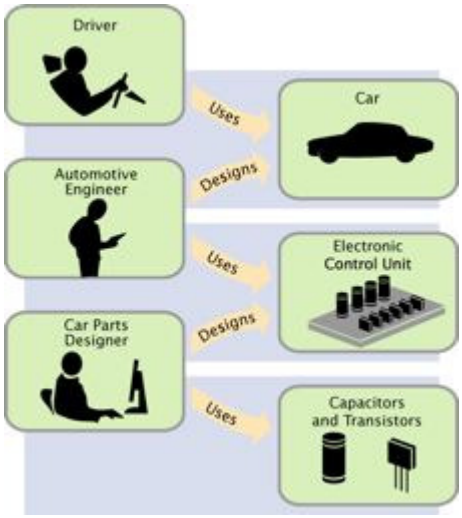
3.1.2 Object-Oriented Design

In old times, computer programs manipulated *primitive types* such as numbers and characters. As programs became more complex, they manipulated more and more of these primitive quantities, until programmers could no longer keep up. It was just too confusing to keep all that detail in one's head. As a result, programmers gave wrong instructions to their computers, and the computers faithfully executed them, yielding wrong answers.

Of course, the answer to this problem was obvious. Software developers soon learned to manage complexity. They encapsulated routine computations, forming software “black boxes” that can be put to work without worrying about the internals. They used the process of abstraction to invent data types that are at a higher level than numbers and characters.

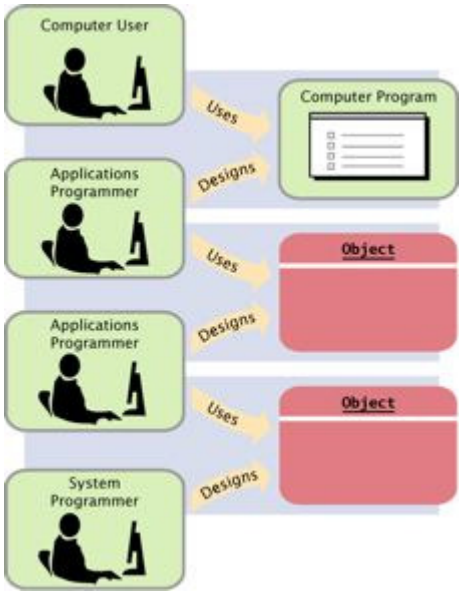
At the time that this book is written, the most common approach for structuring computer programming is the *object-oriented* approach. The black boxes from which a program is manufactured are called objects. An object has an internal structure—perhaps just some numbers, perhaps other objects—and a well-defined behavior. Of course, the internal structure is hidden from the programmer who uses it. That programmer only learns about the object's behavior and then puts it to work in order to achieve a higher-level goal.

Figure 1



Levels of Abstraction in Automotive Design

Figure 2



Levels of Abstraction in Software Design

Who designs these objects? Other programmers! What do they contain? Other objects! This is where things get confusing for beginning students. In real life, the users of black boxes are quite different from their designers, and it is easy to understand the levels of abstraction (see Figure 1). With computer programs, there are also levels of abstraction (see Figure 2), but they are not as intuitive to the uninitiated. To make matters potentially more confusing, you will often need to switch roles, being the designer of objects in the morning and the user of the same objects in the afternoon. In that regard, you will be like the builders of the first automobiles, who singlehandedly produced steering wheels and axles and then assembled their own creations into a car.

There is another challenging aspect of designing objects. Software is infinitely more flexible than hardware because it is unconstrained from physical limitations. Designers of electronic parts can exploit a limited number of physical effects to create transistors, capacitors, and the like. Transportation device manufacturers can't easily produce personal helicopters because of a whole host of physical limitations, such as fuel consumption and safety. But in software, anything goes. With few constraints from the outside world, you can design good and bad abstractions with equal facility. Understanding what makes good design is an important part of the education of a software engineer.

84

85

3.1.3 Crawl, Walk, Run

In Chapter 2, you learned to be an object user. You saw how to obtain objects, how to manipulate them, and how to assemble them into a program. In that chapter, your role was analogous to the automotive engineer who learns how to use an engine control module, and how to take advantage of its behavior in order to build a car.

In this chapter, you will move on to implementing classes. A design will be handed to you that describes the behavior of the objects of a class. You will learn the necessary Java programming techniques that enable your objects to carry out the desired behavior. In these sections, your role is analogous to the car parts manufacturer who puts together an engine control module from transistors, capacitors, and other electronic parts.

In Chapters 8 and 12, you will learn more about designing your own classes. You will learn rules of good design, and how to discover the appropriate behavior of

objects. In those chapters, your job is analogous to the car parts engineer who specifies how an engine control module should function.

SELF CHECK

1. In Chapters 1 and 2, you used `System.out` as a black box to cause output to appear on the screen. Who designed and implemented `System.out`?
2. Suppose you are working in a company that produces personal finance software. You are asked to design and implement a class for representing bank accounts. Who will be the users of your class?

3.2 Specifying the Public Interface of a Class

In this section, we will discuss the process of specifying the behavior of a class. Imagine that you are a member of a team that works on banking software. A fundamental concept in banking is a *bank account*. Your task is to understand the design of a `BankAccount` class so that you can implement it, which in turn allows other programmers on the team to use it.

You need to know exactly what features of a bank account need to be implemented. Some features are essential (such as deposits), whereas others are not important (such as the gift that a customer may receive for opening a bank account). Deciding which features are essential is not always an easy task. We will revisit that issue in Chapters 8 and 12. For now, we will assume that a competent designer has decided that the following are considered the essential operations of a bank account:

In order to implement a class, you first need to know which methods are required.

- Deposit money
- Withdraw money
- Get the current balance

In Java, operations are expressed as method calls. To figure out the exact specification of the method calls, imagine how a programmer would carry out the

bank account operations. We'll assume that the variable `harrysChecking` contains a reference to an object of type `BankAccount`. We want to support method calls such as the following:

```
harrysChecking.deposit(2000);
harrysChecking.withdraw(500);
System.out.println(harrysChecking.getBalance());
```

Note that the first two methods are mutators. They modify the balance of the bank account and don't return a value. The third method is an accessor. It returns a value that you can print or store in a variable.

As you can see from the sample calls, the `BankAccount` class should define three methods:

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

Recall from Chapter 2 that `double` denotes the double-precision floating-point type, and `void` indicates that a method does not return a value.

When you define a method, you also need to provide the method *body*, consisting of statements that are executed when the method is called.

```
public void deposit(double amount)
{
   body—filled in later
}
```

You will see in Section 3.5 how to fill in the method body.

Every method definition contains the following parts:

- An *access specifier* (usually `public`)
- The *return type* (such as `void` or `double`)
- The name of the method (such as `deposit`)

- A list of the *parameters* of the method (if any), enclosed in parentheses (such as `double amount`)
- The *body* of the method: 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 a program can call them. (Occasionally, it can be useful to have `private` methods. They can only be called from other methods of the same class.)

A method definition contains an access specifier (usually `public`), a return type, a method name, parameters, and the method body.

The return type is the type of the output value. The `deposit` method does not return a value, whereas the `getBalance` method returns a value of type `double`.

86

87

SYNTAX 3.1 Method Definition

```
accessSpecifier returnType
methodName(parameterType parameterName, . . . )
{
   method body
}
```

Example:

```
public void deposit(double amount)
{
   . . .
}
```

Purpose:

To define the behavior of a method

Each parameter (or input) to the method has both a type and a name. For example, the `deposit` method has a single parameter named `amount` of type `double`. For each parameter, choose a name that is both a legal variable name and a good description of the purpose of the input.

Next, you need to supply constructors. We will want to construct bank accounts that initially have a zero balance, by using the default constructor:

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

What if a programmer who uses our class wants to start out with another balance? A second constructor that sets the balance to an initial value will be useful:

```
BankAccount momsSavings = new BankAccount(5000);
```

To summarize, it is specified that two constructors will be provided:

- `public BankAccount()`
- `public BankAccount(double initialBalance)`

A constructor is very similar to a method, with two important differences.

- The name of the constructor is always the same as the name of the class (e.g., `BankAccount`)
- Constructors have no return type (not even `void`)

Just like a method, a constructor also has a body—a sequence of statements that is executed when a new object is constructed.

Constructors contain instructions to initialize objects. The constructor name is always the same as the class name.

```
public BankAccount()
{
   body—filled in later
}
```

87

88

The statements in the constructor body will set the internal data of the object that is being constructed—see Section 3.5.

Don't worry about the fact that there are two constructors with the same name—*all* constructors of a class have the same name, that is, the name of the class. The compiler can tell them apart because they take different parameters.

When defining a class, you place all constructor and method definitions inside, like this:

```
public class BankAccount
{
      // Constructors
      public BankAccount()
      {
          body—filled in later
      }
      public BankAccount(double initialBalance)
      {
          body—filled in later
      }
      // Methods
      public void deposit(double amount)
      {
          body—filled in later
      }
      public void withdraw(double amount)
      {
          body—filled in later
      }
      public double getBalance()
      {
          body—filled in later
      }
      private fields—filled in later
}
```

You will see how to supply the missing pieces in the following sections.

The public constructors and methods of a class form the *public interface* of the class. These are the operations that any programmer can use to create and manipulate `BankAccount` objects. Our `BankAccount` class is simple, but it allows programmers to carry out all of the important operations that commonly occur with bank accounts. For example, consider this program segment, authored by a programmer who uses the `BankAccount` class. These statements transfer an amount of money from one bank account to another:

```
// Transfer from one account to another
double transferAmount = 500;
momsSavings.withdraw(transferAmount);
```

```
harrysChecking.deposit(transferAmount);
```

88
89

SYNTAX 3.2 Constructor Definition

```
accessSpecifier ClassName(parameterType
parameterName, . . . )
{
    constructor body
}
```

Example:

```
public BankAccount(double initialBalance)
{
    . . .
}
```

Purpose:

To define the behavior of a constructor

SYNTAX 3.3 Class Definition

```
accessSpecifier class ClassName
{
    constructors
    methods
    fields
}
```

Example:

```
public class BankAccount
{
    public BankAccount(double initialBalance) {. .
.}
    public void deposit(double amount) {. . .}
    . . .
}
```

Purpose:

To define a class, its public interface, and its implementation details

And here is a program segment that adds interest to a savings account:

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

89
90

As you can see, programmers 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.

Of course, as implementors of the `BankAccount` class, we will need to supply the internal details. We will do so in Section 3.5. First, however, an important step remains: *documenting* the public interface. That is the topic of the next section.

SELF CHECK

3. How can you use the methods of the public interface to *empty* the `harrys-Checking` bank account?
4. Suppose you want a more powerful bank account abstraction that keeps track of an *account number* in addition to the balance. How would you change the public interface to accommodate this enhancement?

3.3 Commenting the Public Interface

When you implement classes and methods, you should get into the habit of thoroughly *commenting* their behaviors. 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 3.1 for a description of this utility.)

A documentation comment is placed before the class or method definition that is being documented. It 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`.

> Use documentation comments to describe the classes and public methods of your programs.

The `javadoc` utility copies the *first* sentence of each comment to a summary table in the HTML documentation. 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.

```
/**
   Withdraws money from the bank account.
   @param amount the amount to withdraw
*/
public void withdraw(double amount)
{
    implementation—filled in later                    90
}                                                     91
/**
    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*. 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.

```
/**
    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 take the time to write them, even if it feels silly.

It is always a good idea to write the method comment *first*, before writing the code in the method body. 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.

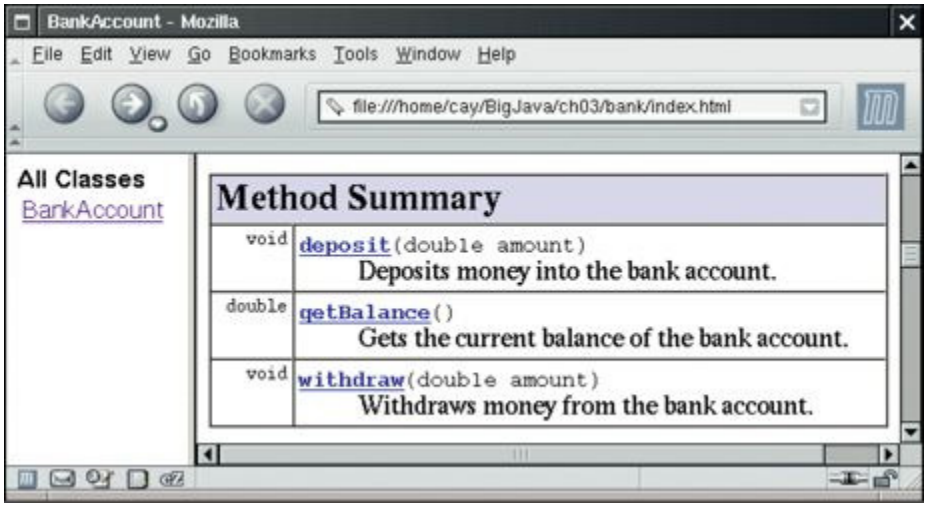
What about very simple methods? You can easily spend more time pondering whether a comment is too trivial to write than it takes to write it. In practical programming, very simple methods are rare. It is harmless to have a trivial method overcommented, whereas a complicated method without any comment 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.

> Provide documentation comments for every class, every method, every parameter, and every return value.

The `javadoc` utility formats 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 3). The `@param` and `@return` comments are neatly formatted in the detail description of each method (see Figure 4). If you omit any of the comments, then `javadoc` generates documents that look strangely empty.

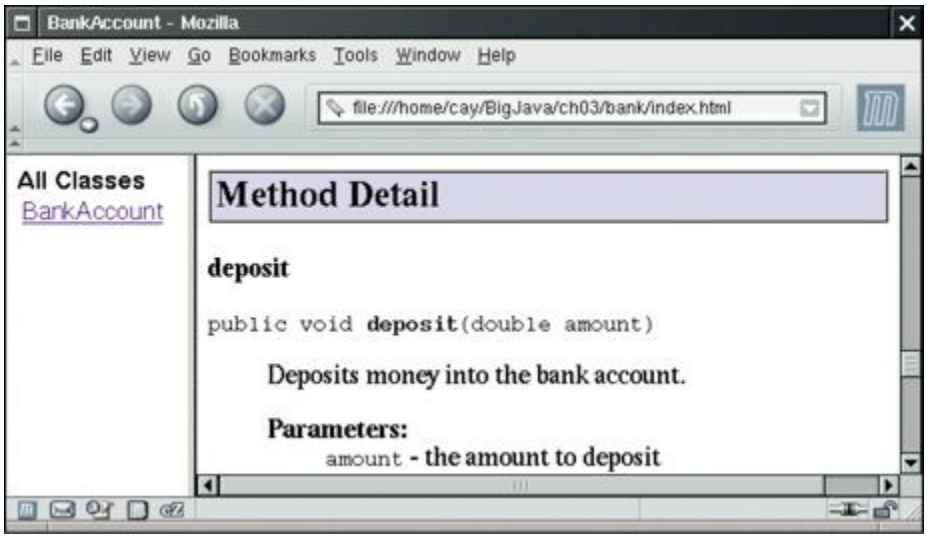
This documentation format should look familiar. The programmers who implement the Java library use `javadoc` themselves. They too document every class, every method, every parameter, and every return value, and then use `javadoc` to extract the documentation in HTML format.

Figure 3



A Method Summary Generated by `javadoc`

Figure 4



Method Detail Generated by `javadoc`

SELF CHECK

5. Suppose we enhance the `BankAccount` class so that each account has an account number. Supply a documentation comment for the constructor

```
public BankAccount(int accountNumber, double
initialBalance)
```

6. Why is the following documentation comment questionable?

```
/**
   Each account has an account number.
   @return the account number of this account
*/
public int getAccountNumber()
```

92

93

PRODUCTIVITY HINT 3.1: The `javadoc` Utility

Always insert documentation comments in your code, whether or not you use `javadoc` to produce HTML documentation. Most people find the HTML documentation convenient, so it is worth learning how to run `javadoc`. Some programming environments (such as BlueJ) can execute `javadoc` for you. Alternatively, you can invoke the `javadoc` utility from a command shell, by issuing the command

```
javadoc MyClass.java
```

or, if you want to document multiple Java files,

```
javadoc *.java
```

The `javadoc` utility produces files such as `MyClass.html` in HTML format, which you can inspect in a browser. If you know HTML (see Appendix H), 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 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 allows you to 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 timely and nicely formatted.

3.4 Instance Fields

Now that you understand the specification of the public interface of the `BankAccount` class, let's provide the implementation.

First, we need to determine the data that each bank account object contains. In the case of our simple bank account class, each object needs to store a single value, the current balance. (A more complex bank account class might store additional data—perhaps an account number, the interest rate paid, the date for mailing out the next statement, and so on.)

An object stores its data in *instance fields*. A *field* is a technical term for a storage location inside a block of memory. An *instance* of a class is an object of the class. Thus, an instance field is a storage location that is present in each object of the class.

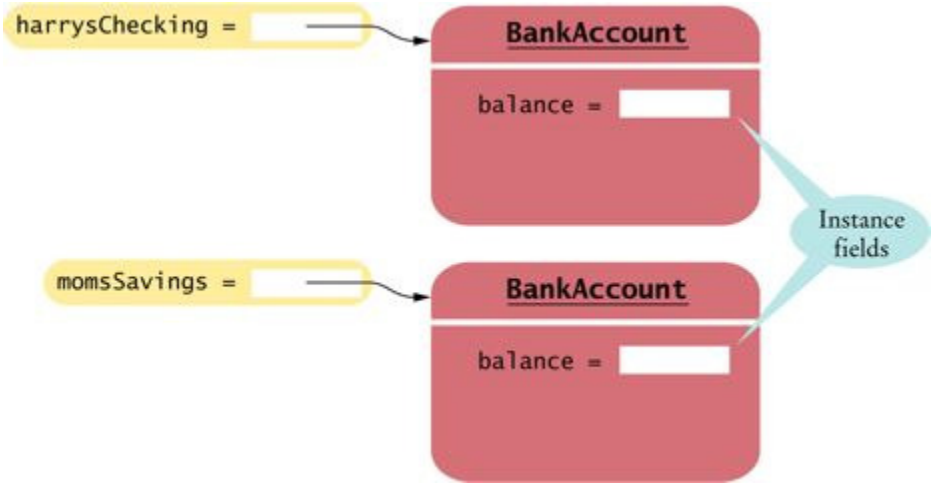
An object uses instance fields to store its state—the data that it needs to execute its methods.

The class declaration specifies the instance fields:

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

93

Figure 5



Instance Fields

An instance field declaration consists of the following parts:

- An *access specifier* (usually `private`)
- The *type* of the instance field (such as `double`)
- The name of the instance field (such as `balance`)

Each object of a class has its own set of instance fields. For example, if `harrysChecking` and `momsSavings` are two objects of the `BankAccount` class, then each object has its own `balance` field, called `harrysChecking.balance` and `momsSavings.balance` (see Figure 5).

> Each object of a class has its own set of instance fields.

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. For example, the `balance` variable can be accessed by the `deposit` method of the `BankAccount` class but not the `main` method of another class.

You should declare all instance fields as private.

```
public class BankRobber
{
      public static void main(String[] args)
      {
          BankAccount momsSavings = new
BankAccount(1000);
          . . .
          momsSavings.balance = -1000; // Error
      }
}
```

Encapsulation is the process of hiding object data and providing methods for data access.

In other words, if the instance fields are declared as 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 a very uncommon practice. We will always make instance fields private in this book.

94 95

SYNTAX 3.4 Instance Field Declaration

```
accessSpecifier class ClassName
{
      . . .
      accessSpecifier fieldType fieldName;
      . . .
}
```

Example:

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

```
    }
```

Purpose:

To define a field that is present in every object of a class

SELF CHECK

7. Suppose we modify the `BankAccount` class so that each bank account has an account number. How does this change affect the instance fields?
8. What are the instance fields of the `Rectangle` class?

3.5 Implementing Constructors and Methods

Now that we have determined the instance fields, let us complete the `BankAccount` class by supplying the bodies of the constructors and methods. Each body contains a sequence of statements. We'll start with the constructors because they are very straightforward. A constructor has a simple job: to initialize the instance fields of an object.

Constructors contain instructions to initialize the instance fields of an object.

Recall that we designed the `BankAccount` class to have two constructors. The first constructor simply sets the balance to zero:

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


The second constructor sets the balance to the value supplied as the construction parameter:

```
public BankAccount(double initialBalance)
{
    balance = initialBalance;
}
```

To see how these constructors work, let us trace the statement

```
BankAccount harrysChecking = new BankAccount(1000);
```

one step at a time. Here are the steps that are carried out when the statement executes.

- Create a new object of type `BankAccount`.
- Call the second constructor (since a construction parameter is supplied).
- Set the parameter variable `initialBalance` to 1000.
- Set the `balance` instance field of the newly created object to `initialBalance`.
- Return an object reference, that is, the memory location of the object, as the value of the `new` expression.
- Store that object reference in the `harrysChecking` variable.

Let's move on to implementing the `BankAccount` methods. Here is the `deposit` method:

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

To understand exactly what the method does, consider this statement:

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

This statement carries out the following steps:

- Set the parameter variable `amount` to 500.
- Fetch the `balance` field of the object whose location is stored in `harrysChecking`.
- Add the value of `amount` to `balance` and store the result in the variable `newBalance`.
- Store the value of `newBalance` in the `balance` instance field, overwriting the old value.

The `withdraw` method is very similar to the `deposit` method:

```
public void withdraw(double amount)
{
    double newBalance = balance - amount;
    balance = newBalance;
}
```

96

97

SYNTAX 3.5 The `return` Statement

```
return expression;
or
return;
```

Example:

```
return balance;
```

Purpose:

To specify the value that a method returns, and exit the method immediately. The return value becomes the value of the method call expression.

There is only one method left, `getBalance`. Unlike the `deposit` and `withdraw` methods, which modify the instance fields of the object on which they are invoked, the `getBalance` method returns an output value:

```
public double getBalance()
{
    return balance;
}
```

The `return` statement is a special statement that instructs the method to terminate and return an output to the statement that called the method. In our case, we simply return the value of the `balance` instance field. You will later see other methods that compute and return more complex expressions.

Use the `return` statement to specify the value that a method returns to its caller.

We have now completed the implementation of the `BankAccount` class—see the code listing below. There is only one step remaining: testing that the class works correctly. That is the topic of the next section.

ch03/account/BankAccount.java

```
1   /**
2      A bank account has a balance that can be
changed by
3      deposits and withdrawals.
4   */
5   public class BankAccount
6   {
7         /**
8            Constructs a bank account with a
zero balance.
9          */
10        public BankAccount()
11        {
12              balance = 0;
13        }
14
15      /**
16              Constructs a bank account with a
given balance.
17              @param initialBalance the initial
balance
18      */
19      public BankAccount(double initialBalance)
20      {
21            balance = initialBalance;
22      }
23
24      /**
25            Deposits money into the bank
account.
26            @param amount the amount to deposit
27      */
28      public void deposit(double amount)
29      {
30            double newBalance = balance +
amount;
31            balance = newBalance;
```

97

98

```
 32       }
 33
 34       /**
 35             Withdraws money from the bank
account.
 36             @param amount the amount to withdraw
 37       */
 38       public void withdraw(double amount)
 39       {
 40             double newBalance = balance -
amount;
 41             balance = newBalance;
 42       }
 43
 44       /**
 45             Gets the current balance of the
bank account.
 46             @return the current balance
 47       */
 48       public double getBalance()
 49       {
 50             return balance;
 51       }
 52
 53       private double balance;
 54   }
```

SELF CHECK

9. The `Rectangle` class has four instance fields: `x`, `y`, `width`, and `height`. Give a possible implementation of the `getWidth` method

10. Give a possible implementation of the `translate` method of the `Rectangle` class.

98

99

How To 3.1: Implementing a Class

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

You will often be asked to implement a class. For example, a homework assignment might ask you to implement a `CashRegister` class.

Step 1 Find out which methods you are asked to supply.

In the cash register example, you won't have to provide every feature of a real cash register—there are too many. The assignment should tell you *which aspects* of a cash register your class should simulate. You should have received a description, in plain English, of the operations that an object of your class should carry out, such as this one:

- Ring up the sales price for a purchased item.
- Enter the amount of payment.
- Calculate the amount of change due to the customer.

For simplicity, we are looking at a very simple cash register here. A more sophisticated model would be able to compute sales tax, daily sales totals, and so on.

Step 2 Specify the public interface.

Turn the list in Step 1 into a set of methods, with specific types for the parameters and the return values. Many programmers find this step simpler if they write out method calls that are applied to a sample object, like this:

```
CashRegister register = new CashRegister();
register.recordPurchase(29.95);
register.recordPurchase(9.95);
register.enterPayment(50);
double change = register.giveChange();
```

Now we have a specific list of methods.

- `public void recordPurchase(double amount)`
- `public void enterPayment(double amount)`
- `public double giveChange()`

To complete the public interface, you need to specify the constructors. Ask yourself what information you need in order to construct an object of your class. Sometimes you will want two constructors: one that sets all fields to a default and one that sets them to user-supplied values.

In the case of the cash register example, we can get by with a single constructor that creates an empty register. A more realistic cash register would start out with some coins and bills so that we can give exact change, but that is beyond the scope of our assignment.

Thus, we add a single constructor:

- `public CashRegister()`

99

100

Step 3 Document the public interface.

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

```
/**
    A cash register totals up sales and computes
change due.
*/
public class CashRegister
{
      /**
          Constructs a cash register with no money
in it.
      */
      public CashRegister()
      {
      }
      /**
          Records the sale of an item.
          @param amount the price of the item
      */
      public void recordPurchase(double amount)
      {
      }

      /**
```

```
            Enters the payment received from the
    customer.
            @param amount the amount of the payment
         */
         public void enterPayment(double amount)
         {
         }

         /**
            Computes the change due and resets the
    machine for the next customer.
            @return the change due to the customer
         */
         public double giveChange()
         {
         }
    }
```

Step 4 Determine instance fields.

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 cash register example, you would want to keep track of the total purchase amount and the payment. You can compute the change due from these two amounts.

```
public class CashRegister
{
    . . .
    private double purchase;
    private double payment;
}
```

100

101

Step 5 Implement constructors and methods.

Implement the constructors and methods in your class, one at a time, starting with the easiest ones. For example, here is the implementation of the `recordPurchase` method:

```
    public void recordPurchase(double amount)
    {
        double newTotal = purchase + amount;
        purchase = newTotal;
    }
```

Here is the `giveChange` method. Note that this method is a bit more sophisticated—it computes the change due, and it also resets the cash register for the next sale.

```
    public double giveChange()
    {
        double change = payment - purchase;
        purchase = 0;
        payment = 0;
        return change;
    }
```

If you find that you have trouble with the implementation, you may need to rethink your choice of instance fields. It is common for a beginner to start out with a set of fields that cannot accurately reflect the state of an object. Don't hesitate to go back and add or modify fields.

Once you have completed the implementation, compile your class and fix any compiler errors.

Step 6 Test your class.

Write a short tester program and execute it. The tester program can carry out the method calls that you found in Step 2.

```
    public class CashRegisterTester
    {
        public static void main(String[] args)
        {
              CashRegister register = new
    CashRegister();
              register.recordPurchase(29.50);
              register.recordPurchase(9.25);
              register.enterPayment(50);
              double change = register.giveChange();
              System.out.println(change);
              System.out.println("Expected: 11.25");
```

```
        }
    }
```

The output of this test program is:

```
11.25
Expected: 11.25
```

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

101

102

3.6 Unit Testing

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

A unit test verifies that a class works correctly in isolation, outside a complete program.

In the long run, your class may become a part of a larger program that interacts with users, stores data in files, and so on. However, before integrating a class into a program, it is always a good idea to test it in isolation. Testing in isolation, outside a complete program, is called *unit testing*.

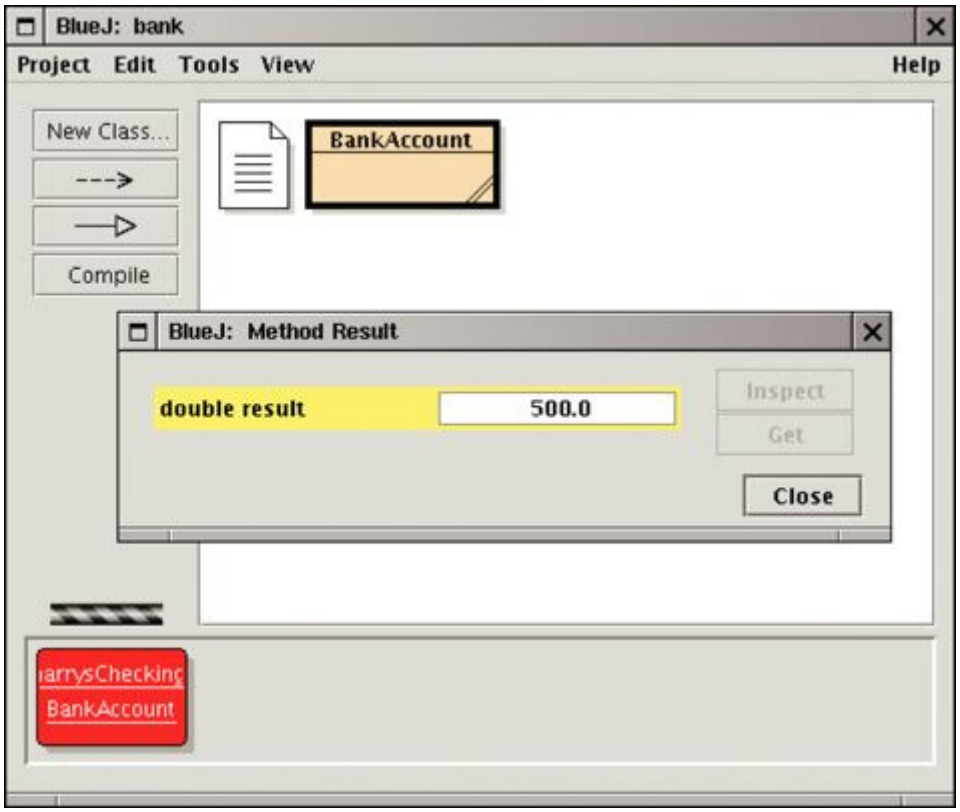
To test your class, you have two choices. Some interactive development environments have commands for constructing objects and invoking methods (see Advanced Topic 2.1). Then you can test a class simply by constructing an object, calling methods, and verifying that you get the expected return values. Figure 6 shows the result of calling the `getBalance` method on a `BankAccount` object in BlueJ.

Alternatively, you can write a *tester class*. A tester class is a class with a `main` method that contains statements to run methods of another class. A tester class typically carries out the following steps:

To test a class, use an environment for interactive testing, or write a tester class to execute test instructions.

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.
4. Print the expected results.

Figure 6



The Return Value of the `getBalance` Method in BlueJ

102

103

The `MoveTester` class in Section 2.8 is a good example of a tester class. That class runs methods of the `Rectangle` class—a class in the Java library.

Here is a class to run methods of the `BankAccount` class. The `main` method constructs an object of type `BankAccount`, invokes the `deposit` and `withdraw` methods, and then displays the remaining balance on the console.

We also print the value that we expect to see. In our sample program, we deposit $2,000 and withdraw $500. We therefore expect a balance of $1500.

ch03/account/BankAccountTester.java

```
1  /**
2     A class to test the BankAccount class.
3  */
4  public class BankAccountTester
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       System.out.println("Expected: 1500");
17    }
18 }
```

Output

```
1500
Expected: 1500
```

To produce a program, you need to combine the `BankAccount` and the `BankAccountTester` 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.

Many students are surprised that such a simple program contains two classes. However, this is normal. The two classes have entirely different purposes. The `Bank-Account` class describes objects that compute bank balances. The `BankAccountTester` class runs a test that puts a `BankAccount` object through its paces.

103

104

SELF CHECK

11. When you run the `BankAccountTester` program, how many objects of class `BankAccount` are constructed? How many objects of type `BankAccountTester`?
12. Why is the `BankAccountTester` class unnecessary in development environments that allow interactive testing, such as BlueJ?

PRODUCTIVITY HINT 3.2: Using the Command Line Effectively

If your programming environment allows you to accomplish all routine tasks using menus and dialog boxes, you can skip this note. However, if you must 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 (including Linux, Mac OS X, UNIX, and Windows) have a *command line interface* to interact with the computer. (In Windows XP, you can get a command line window by selecting "Run …" from the Start menu and typing `cmd`.) 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 commands, such as

```
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, by using the up and down arrow keys to recall old commands and the left and right arrow keys to edit lines. You can also perform *file completion*. For example, to select the file `BankAccount.java`, you only need to type the first couple of letters and then hit the "Tab" key.

The details depend on your operating system and its configuration—experiment on your own, or ask a "power user" for help.

3.7 Categories of Variables

We close this chapter with two sections of a more technical nature, examining variables and parameters in some detail.

You have seen three different categories of variables in this chapter:

1. *Instance fields* (sometimes called *instance variables*), 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

104

105

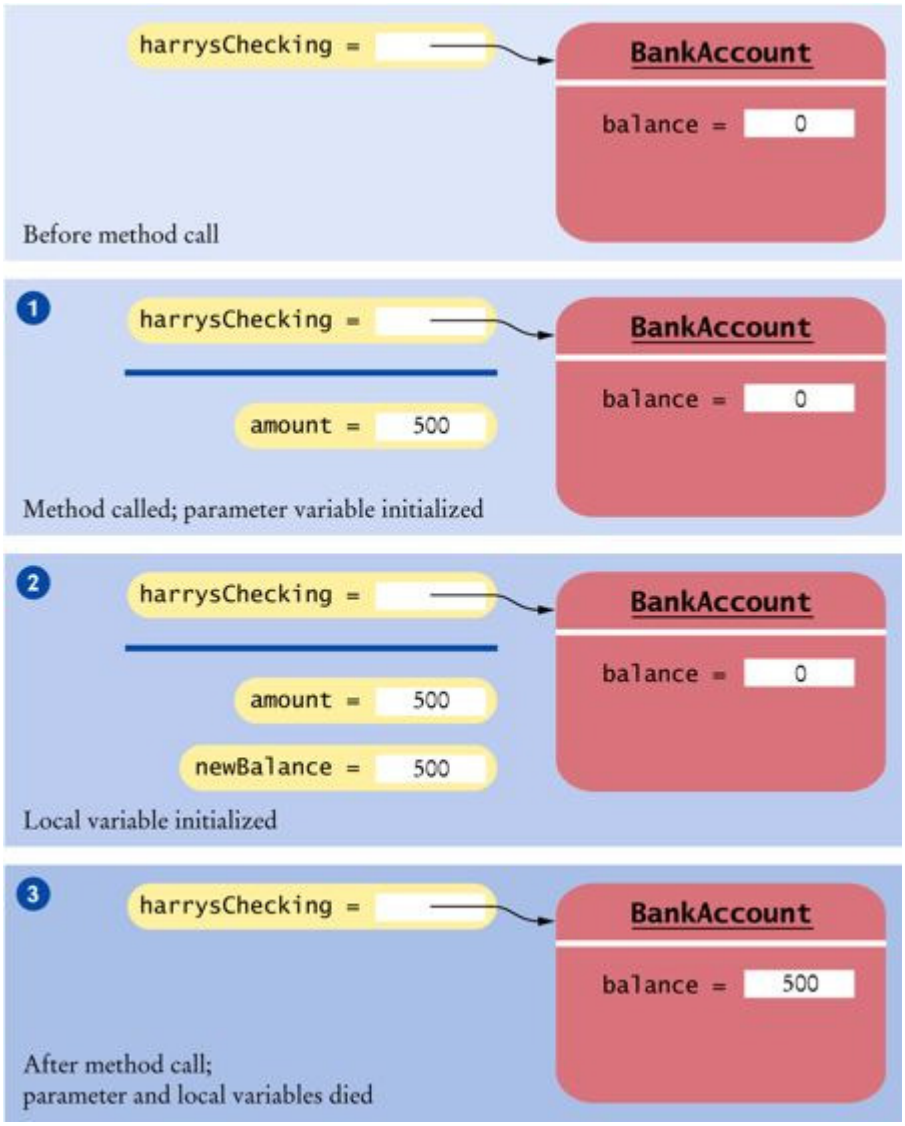
These variables are similar in one respect—they all hold values that belong to specific types. But they have a couple of important differences. The first difference is their *lifetime*.

Instance fields belong to an object. Parameter variables and local variables belong to a method—they die when the method exits.

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. The fields stay alive until no method uses the object any longer. (The Java virtual machine contains an agent called a *garbage collector* that periodically reclaims objects when they are no longer used.)

Local and parameter variables belong to a method. When the method runs, these variables come to life. When the method exits, they die immediately (see Figure 7).

Figure 7



Lifetime of Variables

105

106

For example, if you call

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

then a parameter variable called `amount` is created and initialized with the parameter value, 500. When the method returns, the `amount` variable dies. The same holds for the local variable `newBalance`. When the `deposit` method reaches the line

```
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,

```
balance = newBalance;
```

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

The second major difference between instance fields 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.

Instance fields are initialized to a default value, but you must initialize local variables.

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 5.2.5. 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.

Self Check

13. What do local variables and parameter variables have in common? In which essential aspect do they differ?

14. During execution of the `BankAccountTester` program in the preceding section, how many instance fields, local variables, and parameter variables were created, and what were their names?

COMMON ERROR 3.1: 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.

106

107

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 a modified version of the `BankAccount` class:

```
public class BankAccount
{
    public BankAccount() {} // No statements
    . . .
    private double balance;
    private String owner;
}
```

The `balance` is set to 0, and the `owner` field is set to a `null` reference. This is a problem—it is illegal to call methods on the `null` reference.

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 in a class, 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.

3.8 Implicit and Explicit Method Parameters

In Section 2.4, you learned that a method has an implicit parameter—the object on which the method is invoked—and explicit parameters, which are enclosed in parentheses. In this section, we will examine these parameters in greater detail.

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

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

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

```
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.

107

108

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.

> The implicit parameter of a method is the object on which the method is invoked. The `this` reference denotes the implicit parameter.

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` was set to 500 (see Figure 8).

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 8.) 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 closely at the implementation of the `deposit` method. The statement

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

actually means

```
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:

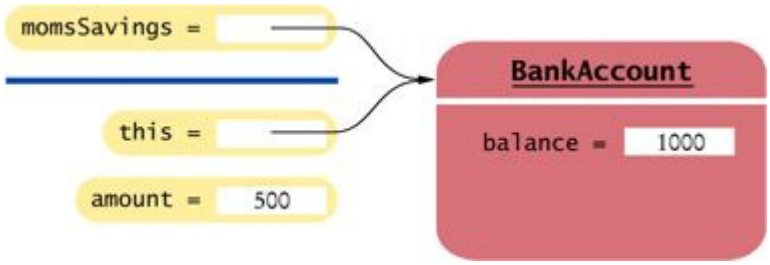
Use of an instance field name in a method denotes the instance field of the implicit parameter.

```
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.

You have now seen how to use objects and implement classes, and you have learned some important technical details about variables and method parameters. In the next chapter, you will learn more about the most fundamental data types of the Java language.

Figure 8



The Implicit Parameter of a Method Call

108

109

SELF CHECK

15. How many implicit and explicit parameters does the `withdraw` method of the `BankAccount` class have, and what are their names and types?

16. In the `deposit` method, what is the meaning of `this.amount`? Or, if the expression has no meaning, why not?

17. How many implicit and explicit parameters does the `main` method of the `BankAccount-Tester` class have, and what are they called?

COMMON ERROR 3.2: Trying to Call a Method Without an Implicit Parameter

Suppose your `main` method contains the instruction

```
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`:

```
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:

```
public class BankAccount
{  . . .
    public void monthly-Fee()
    {
        withdraw(10); // Withdraw $10 from this
account
    }
}
```

That means to withdraw from the same bank account object that is carrying out the `monthly-Fee` operation. In other words, the implicit parameter of the `withdraw` method is the (invisible) 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:

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

109

110

ADVANCED TOPIC 3.1: Calling One Constructor from Another

Consider the `BankAccount` class. It has two constructors: a constructor without parameters to initialize the balance with zero, and another constructor to supply an initial balance. 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:

```
public 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 3.1: Electronic Voting Machines

In the 2000 presidential elections in the United States, votes were tallied by a variety of machines. Some machines processed cardboard ballots into which voters punched holes to indicate their choices (see Punch Card Ballot figure). When voters were not careful, remains of paper—the now infamous "chads"—were partially stuck in the punch cards, causing votes to be miscounted. A manual recount was necessary, but it was not carried out everywhere due to time constraints and procedural wrangling. The election was very close, and there remain doubts in the minds of many people whether the election outcome would have been different if the voting machines had accurately counted the intent of the voters.

Subsequently, voting machine manufacturers have argued that electronic voting machines would avoid the problems caused by punch cards or optically scanned forms. In an electronic voting machine, voters indicate their preferences by pressing buttons or touching icons on a computer screen. Typically, each voter is presented with a summary screen for review before casting the ballot. The process

with curved boundaries requires advanced mathematical tools. Realistic modeling of textures and biological entities requires extensive knowledge of mathematics, physics, and biology.

122

123

CHAPTER SUMMARY

1. In order to implement a class, you first need to know which methods are required.
2. A method definition contains an access specifier (usually `public`), a return type, a method name, parameters, and the method body.
3. Constructors contain instructions to initialize objects. The constructor name is always the same as the class name.
4. Use documentation comments to describe the classes and public methods of your programs.
5. Provide documentation comments for every class, every method, every parameter, and every return value.
6. An object uses instance fields to store its state—the data that it needs to execute its methods.
7. Each object of a class has its own set of instance fields.
8. You should declare all instance fields as private.
9. Encapsulation is the process of hiding object data and providing methods for data access.
10. Constructors contain instructions to initialize the instance fields of an object.
11. Use the `return` statement to specify the value that a method returns to its caller.
12. A unit test verifies that a class works correctly in isolation, outside a complete program.

13. To test a class, use an environment for interactive testing, or write a tester class to execute test instructions.
14. Instance fields belong to an object. Parameter variables and local variables belong to a method—they die when the method exits.
15. Instance fields are initialized to a default value, but you must initialize local variables.
16. The implicit parameter of a method is the object on which the method is invoked. The `this` reference denotes the implicit parameter.
17. Use of an instance field name in a method denotes the instance field of the implicit parameter.
18. It is a good idea to make a class for any part of a drawing that that can occur more than once.
19. To figure out how to draw a complex shape, make a sketch on graph paper.

123

124

FURTHER READING

1. http://verifiedvoting.org A site with information on voter-verifiable voting machines, founded by Stanford computer science professor David Dill.

REVIEW EXERCISES

★ **Exercise R3.1** Why is the `BankAccount(double initialBalance)` constructor not strictly necessary?

★ **Exercise R3.2** Explain the difference between

```
BankAccount b;
```

and

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

★ **Exercise R3.3** Explain the difference between

```
new BankAccount(5000);
```

Finally, 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."`

Test your class in BlueJ. Make objects that greet the world and Dave, and invoke methods on them.

130

131

ANSWERS TO SELF-CHECK QUESTIONS

1. The programmers who designed and implemented the Java library.
2. Other programmers who work on the personal finance application.
3. `harrysChecking.withdraw(harrysChecking.getBalance())`
4. Add an `accountNumber` parameter to the constructors, and add a `getAccount-Number` method. There is no need for a `setAccountNumber` method—the account number never changes after construction.
5.

```
/**
    Constructs a new bank account with a given
initial balance.
    @param accountNumber the account number for
this account
    @param initialBalance the initial balance for
this account
*/
```

6. The first sentence of the method description should describe the method—it is displayed in isolation in the summary table.
7. An instance field

   ```
   private int accountNumber;
   ```

   needs to be added to the class.
8. You can't tell from the public interface, but the source file (which is a part of the JDK) contains these definitions:

```
private int x;
private int y;
private int width;
private int height;
```

9.

```
public int getWidth()
{
      return width;
}
```

10. There is more than one correct answer. One possible implementation is as follows:

```
public void translate(int dx, int dy)
{
    int newx = x + dx;
    x = newx;
    int newy = y + dy;
    y = newy;
}
```

11. One `BankAccount` object, no `BankAccountTester` object. The purpose of the `BankAccountTester` class is merely to hold the `main` method.

12. In those environments, you can issue interactive commands to construct `BankAccount` objects, invoke methods, and display their return values.

13. Variables of both categories belong to methods—they come alive when the method is called, and they die when the method exits. They differ in their initialization. Parameter variables are initialized with the call values; local variables must be explicitly initialized.

14. One instance field, named `balance`. Three local variables, one named `harrysChecking` and two named `newBalance` (in the `deposit` and `withdraw` methods); two parameter variables, both named `amount` (in the `deposit` and `withdraw` methods).

15. One implicit parameter, called `this`, of type `BankAccount`, and one explicit parameter, called `amount`, of type `double`.