

Chapter 4

Classes and Interfaces

Summing Up

The previous chapters presented several classes. Some classes are part of Java's library: `String`, `JFrame`, and `JButton`. Other Java classes are supplied by the author: `TextReader`, `BankAccount`, and `Grid`. You now understand how to construct instances of existing classes and send messages to those objects.

Coming Up

This chapter introduces Java classes and interfaces. You will learn to read and understand classes as collections of methods and data that are related in meaningful ways. You will also learn to implement methods that have access to those data (to the state of objects). You will also see a few object-oriented design guidelines that help explain why classes are designed the way they are. After studying this chapter, you will be able to

- understand how to read and use existing methods
- write your own methods
- understand how to read and use existing Java classes
- implement classes as collections of methods and instance variables
- apply some object-oriented design guidelines to help implement Java classes
- record a list of transactions
- understand how to implement a Java interface

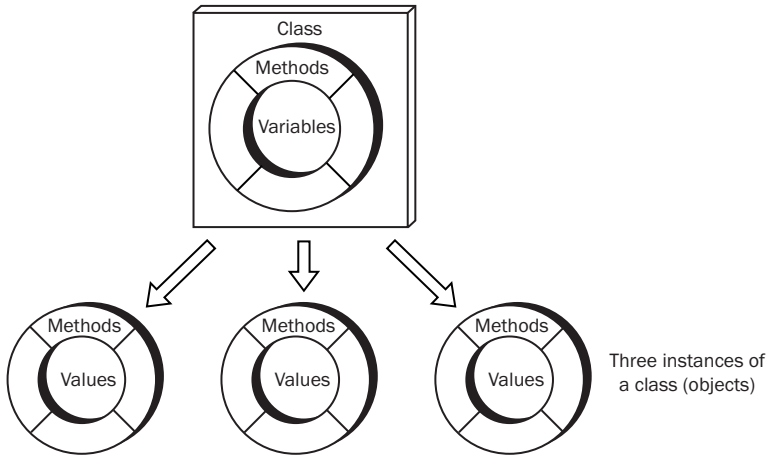
Exercises and programming projects reinforce design and implementation of Java classes.

4.1 Classes

Object-oriented programs use many instances of many different classes. They may be established Java classes, classes bought off the shelf, or classes designed by programmers to suit a particular application.

A class provides a blueprint for constructing objects. The class defines what messages are available to instances of the class. The class defines the values that are encapsulated (the state) in every object.

Figure 4.1: One class constructing three objects, each with its own set of values (state)



A Java class has **methods** that represent the messages each object will understand. Each instance of the class (the object) has its own set of **instance variables** to store the state of each object. Methods are declared `public` to allow other programmers to send messages. Instance variables are declared `private` to protect the data from unauthorized access (more on this later). As the picture above suggests, the state is protected by those methods. These methods are used to access the state or modify the state.

Methods

A Java class typically has many methods. There are two major components to a method:

1. the method heading (the method's signature)
2. the block (a set of curly braces with the code that completes the method's responsibility)

Several **modifiers** may begin a method heading, such as `public`, `private`, `protected`, `final`, and `static`. The examples shown here will use only the modifier `public`. Here is a simplified general form for a method heading.

General Form 4.1: A public method heading

public *return-type* *method-name*(*parameter-1*, *parameter-2*, *parameter-n*, ...)

Return-type represents the type of value returned from the method. The return type may be `void` to indicate that the method returns nothing (as in `JFrame`'s `setSize` method and `Grid`'s `move` method, for example). The return type can be any primitive type, such as `int` or `double` (`String`'s `length` method or `BankAccount`'s `withdraw` method, for example). Additionally, the return type can be any class reference type, such as `String`, `GregorianCalendar`, `BankAccount`, or `ArrayList`.

The *method-name* is any valid Java identifier. Since most methods need one or more pieces of information to get the job done, method headings may also specify parameters between the required parentheses (parameters are discussed in the next section).

Here are a few syntactically correct method headings:

Example Method Headings

```
public int compareTo( Object anotherString ) // String
public void withdraw( double withdrawalAmount ) // BankAccount
public void setSize( int width, int height ) // JFrame
public void setText( String newText ) // JTextField, JLabel
public String toString( ) // String
public int length( ) // String
```

The other part of a method is the body. A method body begins with a curly brace and ends with a curly brace. This is where variable declarations, object constructions, assignments, and messages go. For example, here is the very simple, yet complete, `deposit` method from the `BankAccount` class. This method has access to the parameter `depositAmount` and to the `BankAccount` instance variable named `my_balance` (instance variables are discussed later in this chapter).

```
public void deposit( double depositAmount ) // The method heading
{
    my_balance = my_balance + depositAmount; // The method body
}
```

Parameters

The **parameters** in a method heading specify the number and type of arguments that must be used in a message. A parameter is an identifier declared between the parentheses. For example, `depositAmount` in the `deposit` method above is a

parameter of type `double`. The programmer who wrote the method specifies the number and type of values the method will need to accomplish its responsibility.

A method may need zero, one, two, or even more arguments to do what it is supposed to do. “How much money do you want to withdraw from the `BankAccount` object?” “What is the beginning and ending index of the `substring` you want?” “What size window do you want?” “You want the square root of what number?” Parameters provide the mechanism to get the appropriate information—**arguments**—to the method precisely when it is needed—during the method call (message).

When a method is called during program execution, the value of each argument is passed on to the associated parameter. For each parameter in the method heading, you must supply one expression that can be assigned to the parameter’s type. Once the arguments are passed along to the method through these argument/parameter associations, the program control transfers to the method body. These parameters are known inside the body of the method where they are used while the code executes to do the proper thing for the given argument. The same method should work for a wide variety of argument values.

Here are some examples of passing arguments to parameters. It may help to read the arrow as an assignment statement. First, a `deposit` message to a `BankAccount` object requires the amount to be deposited (a `double`). The argument `123.45` is assigned to `depositAmount` and used inside the `deposit` method.

```
// Credit this account by the argument associated with depositAmount
public void deposit( double depositAmount )
                    ↑
    anAccount.deposit( 123.45 );
```

`Math.pow` requires two `double` arguments to be able to return a^b .

```
// Returns the first argument raised to the power of the second argument
public static double pow( double a, double b )
                        ↑       ↑
    Math.pow( 2.0,      3.0 )
```

To construct a `BankAccount` you must supply a `String` argument followed by a `double` argument. (*Note:* Later in this chapter you will see that a constructor is a special method that has the same name as its class and no return type.)

```
// The BankAccount Constructor--a method heading
public BankAccount( String accountID, double currentBalance )
                    ↑           ↑
BankAccount anAcct = new BankAccount( "Kim", 1576.48 );
```

The examples above have literal arguments. However, an argument may be any expression that evaluates to the parameter’s declared type. Assignment compatibility rules apply. So for example, an argument of type `int` may be passed as an argument associated with a `double` parameter in the method, but not vice versa.

When a method requires more than one argument, the first argument in the message will be assigned to the first parameter, the second argument will be assigned to the second parameter, and so on. In order to get correct results, the programmer must also order the arguments correctly. Whereas not supplying the correct number and type of arguments in a message results in a compiletime (syntax) error, supplying the correct number and type of arguments in the wrong order results in a logic error (the program does what you typed, not what you intended).

Reading Method Headings

When properly documented, method headings help the programmer use the method, explain what the method does, and sometimes state what can go wrong (exceptions could be thrown). All of these things allow the programmer to send messages to objects without knowing the details of the implementation. For example, to send a message to an object, the programmer must:

1. know the method name (remember case sensitivity, `toString` is not `tostring`)
2. supply the proper type and number of arguments
3. use the evaluation of the method correctly (what does the method return?)

All of this information is specified in the method heading. For example, the `substring` method of Java's `String` class takes two `int` arguments and evaluates to a `String`.

```
/** From the String class
 *
 * Return portion of this string indexed
 * from beginIndex through endIndex - 1
 */
public String substring( int beginIndex, int endIndex )
```

The method heading for `substring` provides the following information:

- what the method evaluates to: `String`
- the method name: `substring`
- how many arguments are required: 2
- what type of arguments are required: `int`

Since `substring` is a method of the `String` class, the message must begin with a `String` literal or a `String` reference variable before the dot.

```
// Could use String literals as a String object is constructed
System.out.println( "forever".substring( 0, 3 ) ); // for
System.out.println( "forever".substring( 3, 7 ) ); // ever

// Or more typically, send substring messages to String objects
String str = "small";
System.out.println( str.substring( 1, str.length( ) ) ); // mall
```

This following message asks the `String` object referenced by `fullName` to return its characters starting at `beginIndex`, which is the character 'M' (index 0), to `endIndex - 1`, which is the character 'y' (index 5).

```
String fullName = "Murphy, John";
String lastName = fullName.substring( 0, 6 );
```

A substring message requires two arguments that specify the portion of the `String` to return. This is specified in the method heading that has two parameters named `beginIndex` and `endIndex`. Both arguments are type `int` because the parameters in the `substring` method heading are declared as type `int`. When this message is sent, the argument 0 is assigned to `beginIndex` and the argument 6 is assigned to `endIndex`. `beginIndex` and `endIndex` are called “parameters” (discussed below).

```
public String substring( int beginIndex, int endIndex )
                        ↑                ↑
                        fullName.substring( 0, 6 );
```

Implementation of the substring method is not shown here

The first argument 0 is assigned to the first parameter `beginIndex`. The second argument 6 is assigned to the second parameter `endIndex`. Then control is transferred to the method body, where this information is used with the code there to return what it promises. Although the method bodies cannot be shown for Java's methods, the method bodies will be shown for the author-supplied classes of this textbook.

Self-Check

Use the following method heading to answer the questions that follow. This method is from Java's `String` class.

```
/** From the String class
 *
 * Concatenate str to the end of this String
 */
public String concat( String str )
```

4-1 Determine the following for the `concat` method:

- | | |
|-------------------------------|---|
| -a return type | -d first argument type (or class) |
| -b method name | -e second argument type (or class) |
| -c number of arguments | |

4-2 Assuming `String s = "abc";`, write the return value for each valid message or explain why the message is invalid.

- | | |
|--|---|
| -a <code>s.concat("xyz")</code> | -d <code>s.concat("x", "y")</code> |
| -b <code>s.concat()</code> | -e <code>s.concat("wx" + " yz")</code> |
| -c <code>s.concat(5)</code> | -f <code>String.concat("d")</code> |

- 4-3** Assuming `String s3 = "time";`, what does this message return?
`s3.concat("s");`
- 4-4** Assuming `String s4 = "hope";`, what does this message return?
`s4.concat("less");`
- 4-5** Assuming `String s5 = "";`, what does this message return?
`s5.concat("");`

More examples of methods will be shown next in the context of a class. The next section shows how to implement an entire class of related methods and the instance variables.

Why Those Particular Methods for `BankAccount`?

“Abstraction” refers to the practice of using and understanding something without full knowledge of its implementation. Abstraction allows programmers to concentrate on the behavior of the messages that manipulate state. For example, a programmer using the `String` class need not know the details of the internal data representation or how those methods are implemented in the hardware and software. Programmers can concentrate on the set of allowable messages—the set of methods implemented within the class.

This chapter presents some implementation issues that so far have been hidden. The first part of this chapter presents the now familiar `BankAccount` class at the implementation level. However, before examining class design, first consider some of the design of this textbook’s `BankAccount` class.

All `BankAccount` objects have the same methods. There could have been more, or there could have been fewer. The methods for `BankAccount` were chosen to keep the class simple and to provide a collection of methods that are easy to relate to.

The `BankAccount` methods are only a subset of the methods named by a group of students who were asked this question: “What should we be able to do with bank accounts?” The instance variables are also a subset of the potential state named by students who were asked this question: “What should bank accounts know about themselves?”

When asked, students usually suggest many additional `BankAccount` methods, such as `transfer`, `applyInterest`, and `printMonthlyStatement`, and many additional instance variables, such as type of account, record of transactions, address, social security number, and mother’s maiden name. These are not included here.

The design of this class was affected by the intention of keeping it as simple as possible while retaining some realism. However, a group of object-oriented designers developing a large-scale application in the banking domain would likely retain

many of the methods and states recognized by the students. There is rarely one single design that is correct for all circumstances. Design decisions should be influenced by the desired outcomes. With `BankAccount`, the most important outcome was an easy-to-relate-to example with easy-to-understand methods. This outcome influenced the design.

Designing anything requires making decisions in an effort to make the thing “good.” “Good” might mean having a software component that is easily maintainable, or it might mean having classes designed for reuse in other applications, or it might mean a system that is very robust—one that can recover from almost any disastrous event. “Good” might mean a design that results in something that is easier to use, or that runs very fast, or that is prettier, or . . . There is rarely ever a single perfect design. There are usually tradeoffs. Design is an iterative process that evolves with time. Design is influenced by personal opinion, evolving research, the domain (banking, information systems, process control, or radar, for example), and a variety of other influences.

4.2 Putting It All Together in a Class

Even though different in methods and states, virtually all classes have these things in common:

- private instance variables that store the state of the objects
- constructors that initialize the state
- messages to modify the state of objects
- messages to provide access to the current state of objects

These commonalities guide the effective use of these and similar types of objects. These commonalities also guide implementation of Java classes. These common traits could even be considered a pattern. Simplified General Form 4.1 shows a simplified general form of a Java class.

Many classes begin with `public class` followed by the class name. The instance variables and methods follow within a set of curly braces. The methods and state should have some sort of meaningful connection. You shouldn't put a `blastOff` method or an instance variable that refers to a `Jukebox` object, for example, in `BankAccount`.

Simplified General Form 4.1: A Java class

```

public class class-name
{
    // Instance variables (every instance of this class will get its own)
    private class name;
    private primitive-type identifier;

    // Constructor(s) (methods with the same name as the class)
    public class-name( parameters )
    { // ...
    }
    // Any number of methods
    public return-type method-name-1( parameters )
    { // ...
    }
    public return-type method-name-2( parameters )
    { // ...
    }
    public return-type method-name-n( parameters )
    { // ...
    }
    // Each class may have a method for testing
    public static void main( String[] args )
    { // ...
    }
} // End class-name

```

Java allows one main method in each class. Although optional, a main method provides a convenient way to test the class you are writing. It is recommended that you place a main method in every class you write. For example, here is a simplified version of the `BankAccount` class that was used to construct `BankAccount` objects several times in the preceding chapter:

Figure 4.2: A simplified version of the `BankAccount` class

```

// This class models a minimal bank account.

public class BankAccount
{
    // Instance variables (every BankAccount object will get its own)
    private String my_ID;
    private double my_balance;

```

```

// Initialize instance variables during construction
public BankAccount( String initID, double initBalance )           ❶
{
    my_ID = initID;
    my_balance = initBalance;
}

// Credit this account by depositAmount (hopefully positive)
public void deposit( double depositAmount )                       ❷
{
    my_balance = my_balance + depositAmount;
}

// Debit this account by withdrawalAmount (hopefully positive)
public void withdraw( double withdrawalAmount )                   ❸
{
    my_balance = my_balance - withdrawalAmount;
}

// Return the BankAccount data that identifies this object
public String getID( )                                           ❹
{
    return my_ID;
}

// Return the BankAccount's current balance
public double getBalance( )                                       ❺
{
    return my_balance;
}

// Numbers indicate methods that execute when messages are sent
public static void main( String[] args )

{ // Show the relationship between messages and methods
    BankAccount acctOne = new BankAccount( "01543C", 100.00 );   ❶
    acctOne.deposit( 50.0 );                                       ❷
    acctOne.withdraw( 25.0 );                                       ❸
    System.out.println( acctOne.getID( ) );                         ❹
    System.out.println( acctOne.getBalance( ) );                     ❺
}

} // End class BankAccount

```

Instance Variables

Recall that a `BankAccount` object stores the data necessary to manipulate one account at a bank. Each `BankAccount` object stores some unique identification and an account balance. `BankAccount` methods include making deposits, making withdrawals, and accessing the ID and the current balance.

The private instance variables represent the state. `BankAccount` has two private instance variables: `my_ID` (a `String`) and `my_balance` (a `double`). Every `BankAccount` object knows its own ID and its own current balance.

Notice that the instance variables are not declared within a method. They are declared within the set of curly braces that bounds the class. This means that the instance variables are known throughout the class. Every method has access to these instance variables. If you look at the `BankAccount` class again, you will notice that every method references one of the instance variables (the methods and data are related). Because the instance variables are declared `private`, programs using instances of the class cannot access the instance variables directly. This is good. The class safely encapsulated the state. The only way to change the state of an instance or access the state of an instance is through public methods.

The question now is this: “How do you initialize the state of an object?” The answer: “With a constructor.”

Constructors

The `BankAccount` class of Figure 4.2 shows that all `BankAccount` method headings are public. They also have return types (including `void` to mean return nothing). Some have parameters. However, do you notice something different about the method named `BankAccount`?

First, the method named `BankAccount` has no return type. It also has the same name as the class! This special method is named a **constructor** because this is the method that gets called when objects are constructed. When a constructor is called, memory is allocated for the object. Then the instance variables are initialized using the arguments supplied by the calling program. Here are some object constructions that result in executing the class’s constructor:

```
Grid aGrid = new Grid( 10, 10, 5, 4, Grid.EAST );
String aString = new String( "The initial value of this object" );
BankAccount anAcct = new BankAccount( "Katey Jo", 10.00 );
DecimalFormat numberFormatter = new DecimalFormat( "###,##0.0" );
```

Here is the general form for constructing an object.

General Form 4.2: Constructing objects (from Chapter 3)

```
new class-name( initial-state );
```

The *class-name* is the name of the class. The *object-name* is any valid Java identifier. The *initial-state* is zero or more arguments supplied in the instance creation. When a constructor is encountered with arguments, the initial state is passed on to help initialize the private instance variables of the newly constructed object. The constructor finally returns a reference to that new object. This reference value is often assigned to an object reference of the same type. That is why you often see the class name on both sides of `=`. For example, the following code constructs a

BankAccount object with an initial ID of "Patricia Patterson" and an initial balance of 507.34. After the constructor is done, the reference to this new BankAccount object is assigned to the reference variable named one.

```
BankAccount one = new BankAccount( "Patricia Patterson", 507.34 );
```

Implementing Constructors

A constructor is a special type of method that always has the same name as the class. It never has a return type. The following code implements BankAccount's two-parameter constructor:

```
// A Constructor method to initialize BankAccount objects
public BankAccount( String ID, double initialBalance )
{
    my_ID = ID;
    my_balance = initialBalance;
}
```

This method executes whenever a BankAccount gets constructed with two arguments (a String followed by a number).

In the following code, the ID "Stein" is passed to the parameter ID, which in turn is assigned to the private instance variable my_ID. The starting balance of 250.55 is also passed to the parameter named initialBalance, which in turn is assigned to the private instance variable my_balance.

```
// Call the two-parameter constructor
BankAccount anInitializedAccount = new BankAccount( "Stein", 250.55 );
System.out.println( anInitializedAccount.getID( ) );
System.out.println( anInitializedAccount.getBalance( ) );
```

Output

```
Stein
250.55
```

After an object is constructed, it can respond to messages. Some of these methods access state. Other methods modify state. These are called **accessing methods** and **modifying methods**.

Self-Check

4-6 Write the output from the code below:

```
public class SampleClass
{
    private double my_first;
    private double my_second;
```

```
public SampleClass( double first, double second )
{
    my_first = first;
    my_second = second;
    System.out.println ( "Instance variables: " +
                          my_first + " " + my_second );
}

public static void main ( String[] args )
{ // Test this class
    SampleClass sc1 = new SampleClass( 1.23, 4.56 );
    SampleClass sc2 = new SampleClass( 1.0, 2.0 );
} // End main

} // End SampleClass
```

Modifying Methods

Many methods modify the state of objects. For example, consider `BankAccount`'s `deposit` method, which modifies the private instance variable `my_balance`.

```
// A method that modifies the state of this BankAccount
public void deposit( double depositAmount )
{
    my_balance = my_balance + depositAmount;
}
```

When the following `deposit` message is sent, the argument (157.42) is assigned to the parameter `depositAmount`.

```
anAcct.deposit( 157.42 );
```

The code in the body of the method executes. In this case, the private instance variable `my_balance` is increased by the value of the argument.

`BankAccount`'s `withdraw` method is another method that modifies the state of any `BankAccount` object. Specifically, a `withdraw` message debits `withdrawalAmount` from `my_balance`.

```
// A method that modifies the state of this BankAccount
public void withdraw ( double withdrawalAmount )
{
    my_balance = my_balance - withdrawalAmount;
}
```

When the following `withdraw` message is sent, the argument (50.00) is assigned to the parameter `withdrawalAmount`, which would then be subtracted from `anAcct`'s balance:

```
anAcct.withdraw( 50.00 );
```

Self-Check

4-7 Write the output when this class is run.

```
public class SampleClass
{
    private double my_first;
    private double my_second;

    public SampleClass( double first, double second )
    {
        my_first = first;
        my_second = second;
    }

    public void increaseBy( double increment )
    {
        my_first = my_first + increment;
        my_second = my_second + increment;
        System.out.println( "Changed to: " + my_first + " "
                               + my_second );
    }

    public static void main( String[] args )
    { // Test this class
        SampleClass sc1 = new SampleClass( 1.2, 3.4 );
        sc1.increaseBy( 1.0 );
        sc1.increaseBy( 2.0 );
        sc1.increaseBy( 3.0 );
    }

} // End SampleClass (version 2)
```

Accessing Methods

Some methods provide access to the state of the objects. Some accessing methods simply return the value of individual instance variables with Java's `return` statement.

```
// A method that provides access to the state of this BankAccount
public String getID( )
{
    return my_ID;
}
```

```
// A method that provides access to the state of this BankAccount
public double getBalance( )
{
    return my_balance;
}
```

The Java `return` statement allows a method to return information about the object's state. The expression after `return` is what the message evaluates to. For example, to view the current balance of `anAcct` (the value of the private instance variable `my_balance`), the programmer could write:

```
System.out.println( "Current balance: " + anAcct.getBalance( ) );
```

Whereas a method receives input via the arguments in the method call, a method communicates values back to the message sender through the **return** statement. Here is its general form:

General Form 4.3: Return statement

return *expression*;

When a `return` statement is encountered, the *expression* that follows `return` replaces the message. This is how a message evaluates to a value. Whereas a `void` method returns nothing (see `BankAccount deposit`), any method that has a return type other than `void` *must* return a value of the same type as the return type. So a method declared to return a `String` (as in `BankAccount getID`) must return a reference to a `String`. A method declared to return a `double` (as in `BankAccount getBalance`) must return a primitive `double` value. Fortunately, the compiler will complain if you forget to return the proper type of value.

Some accessing methods use the instance variables to do more complex processing. For example, `WeeklyEmployee`'s `getIncomeTax` method (see Chapter 6's programming projects) is quite complex. This method does not simply return the value of an instance variable. Instead, the instance variables are used along with complex United States Internal Revenue Service tax tables and evaluations of other methods such as `getGrossPay`.

Self-Check

4-8 Write the output from the `main` method below using the modified `SampleClass`.

```
// The final version of a class meant for self-check only

public class SampleClass
{
    private double my_first;
    private double my_second;

    public SampleClass( double first, double second )
    {
        my_first = first;
        my_second = second;
    }
}
```

```

public void increaseBy( double increment )
{
    my_first = my_first + increment;
    my_second = my_second + increment;
}

public double getFirst( )
{
    return my_first;
}

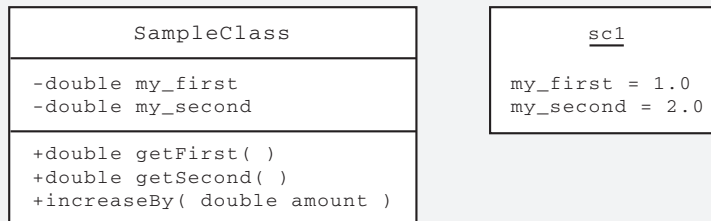
public double getSecond( )
{
    return my_second;
}

public double firstPlusSecond( )
{
    return my_first + my_second;
}

public static void main( String[] args )
{ // Test this class
    SampleClass sc1 = new SampleClass( 1.0, 2.0 );
    System.out.println( "First: " + sc1.getFirst( ) );
    System.out.println( "Second: " + sc1.getSecond( ) );
    System.out.println( "Sum: " + sc1.firstPlusSecond( ) );
    sc1.increaseBy( 3.4 );
    System.out.println( "First: " + sc1.getFirst( ) );
    System.out.println( "Second: " + sc1.getSecond( ) );
    System.out.println( "Sum: " + sc1.firstPlusSecond( ) );
}
} // End SampleClass (version 3)

```

- 49** Given the class diagram and instance diagram when `sc1` is constructed, draw the instance diagram of `sc1` at the end of the program to indicate the final state of the object.



Naming Conventions

A method that modifies state is typically given a name that indicates that the message will change the state of the object. This is easily accomplished if the designer of the class provides a descriptive name for the method. The method name should describe—as best as possible—what the method actually does. It should also help to distinguish modifying methods from accessing methods. Use verbs for modifying method names: `withdraw`, `deposit`, `borrowBook`, and `returnBook`, for example. Give accessing methods names to indicate that the messages will return some useful information about the objects: `getBorrower` and `getBalance`, for example. Considering that the constructor has the same name as the class, some guidelines can now be established for designing and reading classes. These three categories of methods can be distinguished by following these naming conventions:

Method	Name to Use
Constructor	Same name as the class
Modifies state	Identifier that could be used as a verb
Accesses state	Identifier that could be used as a noun
	<i>Note:</i> Because some words can be used either as verbs or nouns (“balance,” for example), some programmers precede modifying methods with <code>set</code> and accessing methods with <code>get</code> .

Above all, always use identifiers that accurately describe what the method does. For example, don’t use `with` as the name of the method to withdraw money from a `BankAccount`.

public or private

One of the considerations in the design of any class is declaring methods and instance variables with the most appropriate access mode, either `public` or `private`. Whereas the **public** methods of a class can be called by a programmer from outside of the class, the **private** instance variables are only known in the class methods. For example, the `BankAccount` instance variable named `my_balance` is known only to the methods of the class. On the other hand, any method declared as `public` is known where the object was constructed.

Access Mode	Where the Identifier Can Be Accessed (where the identifier is known)
<code>public</code>	In all other methods of the class In any method body where the object gets constructed
<code>private</code>	In all methods of a class where an instance is declared Only from the methods of the same class

Although instance variables representing state could be declared as `public`, it is highly recommended that all instance variables be declared as `private`. There are several reasons for this. The consistency helps simplify some design decisions. More importantly, when instance variables are made `private`, the state can be modified only through a method. This prevents other code from indiscriminately changing the state of objects. For example, it is impossible to accidentally make a credit to `acctOne` like this:

```
BankAccount acctOne = new BankAccount( "Mine", 100.00 );

// A compiletime error occurs: attempting to modify private data
acctOne.my_balance = acctOne.my_balance + 100000.00; // <- ERROR
```

or a debit like this:

```
// A compiletime error occurs at this attempt to modify private data
acctOne.my_balance = acctOne.my_balance - 100.00; // <- ERROR
```

This represents a widely held principle of software development—data should be hidden. Making instance variables `private` is one more step on the way to well-designed object-oriented programs.

4.3 Bank Teller — A Transaction Class

One of the required features of the bank teller system is the ability to show the most recent 10 transactions for any account. This section makes progress towards that goal by showing how a collection of banking transactions can be maintained with one `ArrayList` object. This section also provides another example of a Java class while introducing class constants.

`Transaction` objects maintain records of withdrawals and deposits that occur while the bank teller system is running. To accomplish this, a `Transaction` class was designed to store the account ID, the transaction date, some sort of transaction code, the amount of the transaction, and the balance of the account after the transaction has occurred. There currently is no modifying method. To demonstrate how `Transaction` objects are constructed, consider the following code, which builds a small list of `Transaction` objects:

```
// Construct two Transaction objects and store
// references to them in an ArrayList.
import java.util.ArrayList;

public class AFewTransactions
{
    public static void main( String[] args )
    {
        ArrayList transactionList = new ArrayList( );
        BankAccount currentAccount = new BankAccount( "OneAccount", 1000.00 );
```

```

Transaction aTransaction;
double amount;

// Perform a withdraw transaction and record it
amount = 150.00;
currentAccount.withdraw( amount );
// Class constants WITHDRAW_CODE and
// DEPOSIT_CODE are explained below
aTransaction = new Transaction( currentAccount,
                               amount,
                               Transaction.WITHDRAW_CODE );
transactionList.add( aTransaction );

// Perform a deposit transaction and record it
amount = 327.55;
currentAccount.deposit( amount );
aTransaction = new Transaction( currentAccount,
                               amount,
                               Transaction.DEPOSIT_CODE );
transactionList.add( aTransaction );

System.out.println( transactionList.toString() );
}
}

```

Output

```

[Fri Oct 05 15:17:01 MST 2001 W 150.0 OneAccount 850.0,
Fri Oct 05 15:17:01 MST 2001 D 327.55 OneAccount 1177.55]

```

This Transaction class exists mostly to store the data of a transaction. There is no modifying method (but one might appear later if needed). Currently, this Transaction class has enough to allow for a reasonable list of Transaction objects (but not the most recent 10 transactions—that comes later, in Chapter 7).

Two instance variables are initialized by copying the arguments: transactionAmount and code. Two instance variables are initialized by asking for information from the BankAccount object passed to the constructor: accountID and balanceAfterTransaction. Two other instance variables are initialized without relying on the arguments. The transactionDate is set to the computer system’s clock. It does not rely on arguments supplied by the calling code.

```

// A Transaction class to allow a list of transactions.
// This class will be modified in Chapter 7.
// Its main purpose is to store information about single
// transactions in such a way that a list of these can be maintained.
// Each transaction will need to be verified within a few business days.
import java.util.GregorianCalendar;

public class Transaction
{
    public static final String WITHDRAW_CODE = "W";
    public static final String DEPOSIT_CODE = "D";

```

```

private String accountID;
private double transactionAmount;
private String transactionCode;
private double balanceAfterTransaction;
private GregorianCalendar transactionDate;

// Construct a Transaction object for various types of transactions
public Transaction( BankAccount theAccount,
                  double amountOfTransaction,
                  String code )
{
    accountID = theAccount.getID( );
    balanceAfterTransaction = theAccount.getBalance( );
    transactionAmount = amountOfTransaction;
    transactionCode = code;
    transactionDate = new GregorianCalendar( ); // The current date
}

// Return the account ID of this account. This will be used later
// when looking for all transactions for one particular account.
public String getAccountID( )
{
    return accountID;
}

// Return a textual representation of this transaction
public String toString( )
{ // getTime returns a Date object that has a toString method.
  // The Date object's toString( ) provides more detail
  // such as a local time zone.
  String dateAsString = transactionDate.getTime( ).toString( );
  return dateAsString + " " + transactionCode + " " +
         transactionAmount + " " + accountID + " " +
         balanceAfterTransaction;
}
}

```

Class Constants

Two class constants were used in the `Transaction` class. A **class constant** is an identifier declared to represent a value that cannot change. Whereas `Transaction` objects could have been just as easily constructed with `String` literals like this:

```

new Transaction( currentAccount, amount, "W" );
new Transaction( currentAccount, amount, "D" );

```

The presence of these two class constants in the `Transaction` class:

```

public static final String WITHDRAW_CODE = "W";
public static final String DEPOSIT_CODE = "D";

```

allows a programmer to use more descriptive, easier-to-remember names.

```
new Transaction( currentAccount, amount, Transaction.WITHDRAW_CODE );
new Transaction( currentAccount, amount, Transaction.DEPOSIT_CODE );
```

A class constant is declared `public` when you want other code to use a descriptive name rather than some code that has no meaning. The keyword `static` signifies that there is precisely one value for all objects of the class.¹ There is no reason to have a copy of a class constant stored in every instance of the class. The other modifier is `final`. When an identifier is declared `final`, no new value can be assigned to it. This prevents other code from accidentally modifying a value that many different pieces of code rely on.

Here are some class constants (also known as “fields”) as they would be declared inside the Java classes `Math`, `Font`, `BorderLayout`, `JFrame`, and `GregorianCalendar` and the `Grid` class. Although not required, programmers have adopted the convention of making the identifiers for class constants all upper case. If there is more than one word, the words are separated with the underscore character (`_`).

```
public static final int ITALIC = 1;           // Font
public static final int BOLD = 2;           // Font
public static final int JANUARY = 0;        // GregorianCalendar
public static final int DAY_OF_MONTH = 5;   // GregorianCalendar
public static final int WEST = 3;          // Grid
public static final String WEST = "West";   // BorderLayout
public static final int EXIT_ON_CLOSE = 3;  // JFrame
```

Although any primitive or reference type can be made to be `static` and `final` (a class constant), many class constants are `int` values representing a code that may or may not be arbitrary. The following program shows the values of the class constants. To be used outside the class where it is declared, the identifier must be preceded by the class name and a dot.

```
import java.util.*;      // GregorianCalendar
import java.awt.*;      // Font, BorderLayout
import javax.swing.*;   // JFrame

public class ShowSomeConstants
{
    public static void main( String[] args )
    {
        System.out.println( "          PI: " + Math.PI );
        System.out.println( "          ITALIC: " + Font.ITALIC );
        System.out.println( "          BOLD: " + Font.BOLD );
        System.out.println( " DAY_OF_WEEK: " +
            GregorianCalendar.DAY_OF_WEEK );
    }
}
```

-
1. Be very careful with the keyword `static`. If you want an instance variable and accidentally declare it to be `static`, you no longer have an instance variable. There is only one per class. For example, if `my_ID` and `my_balance` were declared `static`, every instance of `BankAccount` would have the same ID and the same balance—not at all desirable.

```

System.out.println( "    JANUARY: " +
    GregorianCalendar.JANUARY );
System.out.println( " DAY_OF_MONTH: " +
    GregorianCalendar.DAY_OF_MONTH );
System.out.println( "    WEST: " + Grid.WEST );
System.out.println( " Another WEST: " + BorderLayout.WEST );
System.out.println( "EXIT_ON_CLOSE: " + JFrame.EXIT_ON_CLOSE );
}
}

```

Output

```

    PI: 3.141592653589793
    ITALIC: 2
    BOLD: 1
    DAY_OF_WEEK: 7
    JANUARY: 0
    DAY_OF_MONTH: 5
    WEST: 3
    Another WEST: West
    EXIT_ON_CLOSE: 3

```

Class constants are used for a variety of purposes. In the case of `Transaction`, they are used to ensure that the correct codes are written from any location. Class constants also help avoid the use of magic numbers, such as 0.0612 and 19. The values actually represent the United States social security tax rate and the maximum credits a student may take in a semester at a certain university. These values would be better described as class constants in the appropriate class.

```

public class Employee
{
    public static final double SOCIAL_SECURITY_TAX_RATE = 0.062;
    ...
}

public class RegistrarPolicies
{
    public static final int MAXIMUM_CREDITS_PER_SEMESTER = 19;
    ...
}

```

Self-Check

Use this Java class and the class diagram to answer the questions that follow.

```

// A class to model a library book
public class LibraryBook
{
    // Instance variables
    private String my_author;
    private String my_title;
    private String my_borrower;

```

```

// Construct a LibraryBook object while
// initializing instance variables
public LibraryBook( String initTitle, String initAuthor )
{
    my_title = initTitle;
    my_author = initAuthor;
    my_borrower = null; // When my_borrower == null,
                        // no one has the book
}

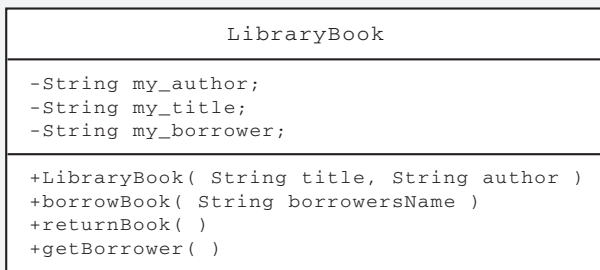
// Records the borrower's name
public void borrowBook( String borrowersName )
{
    my_borrower = borrowersName;
}

// The book becomes available. When
// null, no one is borrowing it.
public void returnBook( )
{
    my_borrower = null;
}

// Return the borrower's name if the book has been
// checked out or return null if the book is available
public String getBorrower( )
{
    return my_borrower;
}
}

```

Class Diagram



- 4-10** What is the name of the class shown above?
- 4-11** Except for the constructor, name all of the methods.
- 4-12** `getBorrower` returns what type of object?
- 4-13** `borrowBook` returns a reference to what type of value?
- 4-14** What class of argument must be part of all `borrowBook` messages?
- 4-15** How many arguments are required to initialize one `LibraryBook` object?

- 4-16** Initialize one `LibraryBook` object using your favorite book and author.
- 4-17** Send the message that borrows your favorite book. Use your own name as the borrower.
- 4-18** Write the message that reveals the name of the person who borrowed your favorite book (or `null` if no one has borrowed it).
- 4-19** Given the following program, draw two instance diagrams for `aBook` immediately after it is constructed and then immediately after the `borrowBook` message. Your instance diagrams should indicate the changed state of the object.

```
// Construct a LibraryBook object and
// send all possible messages
public class TestLibBook
{
    public static void main( String[] args )
    {
        LibraryBook aBook =
            new LibraryBook( "Little Drummer Girl", "John LeCarre" );

        aBook.borrowBook( "Chris Miller" );
        System.out.println( "Borrower: " + aBook.getBorrower( ) );
        aBook.returnBook( );
        System.out.println( "Borrower: " + aBook.getBorrower( ) );
    }
}
```

- 4-20** Write the output generated by the program above.
- 4-21** Which methods have the same name as the class?
- 4-22** What do accessing methods do?
- 4-23** What do modifying methods do?
- 4-24** What do constructors do?
- 4-25** What do instance variables do?
- 4-26** Can all methods in a class reference the `private` instance variables of that class?
- 4-27** Can a program that constructs an object directly reference the `private` instance variables of that object?
- 4-28** Write a class constant that represents the number of days in a leap year.

4.4 A Few Object-Oriented Design Guidelines

An object-oriented **design guideline** is a guideline intended to help produce good object-oriented software. This section introduces the first of several guidelines.

One particular decision made while designing classes involves determining the level of access other objects have to instance variables and methods. More specifically, the programmer has to decide if instance variables should be declared `public` or `private`. The following design guideline suggests that it is good **object-oriented design** to protect the state of objects from accidental changes from the outside. This is encapsulation. This is good.

Object-Oriented Design Guideline 4-1

All data should be hidden within its class.²

Although instance variables could be made `public`, the convention used in the classes shown so far—and in any good class—is this: “Hide the instance variables.” Java instance variables are easily hidden when declared as `private` inside the class. This simplifies one design decision that must be made in any new classes that you develop. The state of all instances of the class will then be protected from accidental and improper alteration. With instance variables declared `private`, the state of any object can be altered only through a message. For example, it becomes impossible to accidentally make a false debit like this:

```
// If my_balance were public, what would the balance be after this?  
acctOne.my_balance = acctOne.my_balance - acctOne.my_balance;  
// A compiletime error occurs at this attempt to modify private data
```

However, if `my_balance` had been declared as a `public` instance variable, the compiler would not protest. The resulting program would allow you to destroy the state without checking on a few things or triggering some transaction. The hidden balance is more properly modified only when the transaction is allowed according to some policy. What happens, for instance, if a `withdrawalAmount` exceeds the account balance in a `withdraw` message? Some accounts allow this by transferring money from a savings account. Other bank accounts may generate loans in increments of `100.00` or some other amount.

With `my_balance` declared as `private`, users of the class must instead send a `withdraw` message. The code relies on the `BankAccount` to determine if the withdrawal is to be allowed. Perhaps the `BankAccount` object will ask some other object if the withdrawal is to be allowed. Perhaps it will delegate authority to some unseen `BankManager` object, for example. Or perhaps the `BankAccount` object

2. Some design guidelines (including this one) are from Arthur J. Riel, *Object-Oriented Design Heuristics* (Boston, MA: Addison Wesley, 1996). Riel cataloged and/or wrote 60 such guidelines.

itself will decide what to do. Although this textbook’s implementation of `BankAccount` doesn’t do much, real-world withdrawals certainly do.

By hiding data and other details, all credits and debits must “go through the proper channels.” This might be quite complex. For example, each withdrawal or deposit may be recorded in a transaction file to help prepare monthly statements for each `BankAccount`. The `withdraw` and `deposit` methods may have additional processing to prevent unauthorized credits and debits. Part of the hidden red tape might include manual verification of a deposit or a check-clearing method at the host bank; there may be some sort of human or computer intervention before any credit is actually made. Such additional processing and protection within the `deposit` and `withdrawal` methods help give `BankAccount` a “safer” design. Because all hidden processing and protection is easily circumvented when instance variables are exposed through `public` access, the object designer must enforce proper object use and protection by hiding the instance variables. Simply declare all instance variables `private`.

Cohesion within a Class

The set of methods specified in a class should be strongly related. A class has instance variables. The instance variables should be strongly related. In fact, all elements of a class should have a persuasive affiliation with each other. These ideas relate to the preference for high **cohesion** (which means solidarity, hanging together, adherence, or unity) within a class. For example, don’t expect a `BankAccount` object to understand the message `areYouPreheated`. This may be an appropriate message for an `oven` object, but certainly not for a `BankAccount` object. Here is one guideline related to the desirable attribute of cohesion.

Object-Oriented Design Guideline 4-2

Keep related data and behavior in one place.

The `BankAccount` class should hide certain policies, such as handling withdrawal requests greater than the balance. The system’s design improves when behavior and data combine to accomplish the withdrawal algorithm. This makes for nice clean messages like this:

```
anAccount.withdraw( withdrawalAmount );
```

This code relies on the `BankAccount` object to determine what should happen. The behavior should be built into the object that has the necessary data. Perhaps the algorithm allows a withdrawal amount greater than the balance, with the extra cash coming as a loan or as a transfer from a savings account. A bank account class might have many hidden actions that are triggered during every withdrawal message.

Self-Check

- 4-29 Does a programmer need to know the names of instance variables to use objects of a class?
- 4-30 Does a programmer need to know the names of methods to use objects?
- 4-31 Describe where the `public` methods of a class are known.
- 4-32 Describe where the `private` methods of a class are known.
- 4-33 Give one justification for making the instance variables of a class `private`.
- 4-34 If the designer of a class changed the instance variable named `my_balance` to `myBalance`, would the code *using* the object need to be changed?
- 4-35 If the method name `withdraw` were changed to `withdrawThisAmount` after the class were already in use by dozens of programs, would these dozens of programs need to be changed?
- 4-36 Should a `BankAccount` object understand the message `areTheBrakesLockingUp`?

4.5 Java Interfaces

A Java **interface** contains a set of method headings followed by semicolons rather than method implementations—code within `{ }`. To be useful, an interface must be implemented by at least one Java class. The implementing class will add instance variables, constructors, and the method bodies for each method specified in the interface.

There are several reasons Java has interfaces as part of the language. In this section, the Java `interface` is introduced to help you implement some of your first classes in the Programming Projects section. You will also see two Java interfaces represent the design of two classes for a new system, which is partially developed in an upcoming section. In Chapter 5, you will see that Java interfaces are required to implement programs that respond to button clicks and text-field input. In Chapter 6, you will see how another Java interface allows a collection in an `ArrayList` to be arranged in order (sorted).

Java has 422 interfaces. Of the 1,732 Java classes, 646 classes (about 37%) implement one or more interfaces. Considering the large number of interfaces in Java and the high percentage of Java classes that implement the interfaces, interfaces are an important part of the Java programming language. However, in this textbook you will not be asked to write the interfaces themselves (like the `TimeTalker` interface below). Instead, you will be asked to write a class that implements an `interface`. The interface will be given to you or it will be one of Java's 422 interfaces. First, consider a very simple interface.

A Java interface begins with a heading that is similar to a Java `class` heading, except the keyword `interface` is used. A Java interface cannot have constructors or instance variables. A Java interface will be implemented by one or more Java classes that add instance variables and have their own constructors. A Java interface specifies the method headings that someone decided would represent what each object of the class must be able to do.

Here is a sample Java interface that has only one method. Although not very useful, it provides a simple first example of an interface and the classes that implement the interface in different ways:

```
// File name: TimeTalker.java
// An interface that will be implemented by several classes
public interface TimeTalker
{
    // Return a representation of how each implementing class tells time
    public String whatTimeIsIt( );
}
```

For each interface, there are usually two or more classes that implement it. Here are three classes that implement the `TimeTalker` interface. One instance variable has been added to store the name of any `TimeTalker`. A constructor was also needed to initialize this instance variable with anybody's name.

```
// File name: FiveYearOld.java
// Someone who cannot read time yet
public class FiveYearOld implements TimeTalker
{
    String my_name;

    public FiveYearOld( String name )
    {
        my_name = name;
    }

    public String whatTimeIsIt( )
    {
        return my_name + " says it's morning time";
    }
}
```

```
// File name: DeeJay.java
// A "hippy dippy" DJ who always mentions the station
public class DeeJay implements TimeTalker
{
    String my_name;

    public DeeJay( String name )
    {
        my_name = name;
    }
}
```

```

    public String whatTimeIsIt( )
    {
        return my_name +
            " says it's 7 oh 7 on your favorite oldies station";
    }
}

// File name: FutureOne.java
// A "Star-Trekker" who speaks of star dates
public class FutureOne implements TimeTalker
{
    String my_name;

    public FutureOne( String name )
    {
        my_name = name;
    }

    public String whatTimeIsIt( )
    {
        return my_name + " says it's star date 78623.23";
    }
}

```

One reason Java has interfaces is to allow many classes to be treated as the same type. To demonstrate that this is possible, consider the following code, which stores references to three different types of objects as `TimeTalker` variables.

```

public class ThreeTypesOfTimeTalkers
{
    public static void main( String[] args )
    {
        // These three objects can be referenced by variables
        // of the interface type that they implement.
        TimeTalker youngOne = new FiveYearOld( "Kieran" );
        TimeTalker dj = new DeeJay( "WolfMan Jack" );
        TimeTalker captainKirk = new FutureOne( "Jim" );

        System.out.println( youngOne.whatTimeIsIt() );
        System.out.println( dj.whatTimeIsIt() );
        System.out.println( captainKirk.whatTimeIsIt() );
    }
}

```

Output

```

Kieran says it's morning time
WolfMan Jack says it's 7 oh 7 on your favorite oldies station
Jim says it's star date 78623.23

```

Because each class implements the `TimeTalker` interface, references to instances of these three classes—`FiveYearOld`, `DeeJay`, and `FutureOne`—can be stored in

the reference type variable `TimeTalker`. They can all be considered to be of type `TimeTalker`. However, the same message to the three different classes of `TimeTalker` objects results in three different behaviors. The same message executes three different methods in three different classes.

In the section that follows, the Java interface plays a part in the design and implementation of two specific Java classes. The interface specifies the exact method headings of the classes that need to be implemented. These interfaces capture design decisions—what instances of the class should be able to do—that were made by a team of programmers.

Self-Check

4-37 Write two classes that implement this interface:

```
public interface BarnyardAnimal
{
    public String sound( );
}
```

The following program must generate the output exactly as shown.

```
public class OldMacDonald
{
    public static void main( String[] args )
    {
        BarnyardAnimal a1 = new Chicken( "cluck" );
        BarnyardAnimal a2 = new Cow( "moo" );

        System.out.println( "With a " + a1.sound( ) + " "
            + a1.sound( ) + " here," );
        System.out.println( "and a " + a2.sound( ) + " "
            + a2.sound( ) + " there." );
        System.out.println( "Here a " + a1.sound( ) + "," );
        System.out.println( "there a " + a2.sound( ) + "," );
        System.out.println( "everywhere a " + a1.sound( )
            + " " + a1.sound( ) + "." );
    }
}
```

Output

```
With a cluck cluck here,
and a moo moo there.
Here a cluck,
there a moo,
everywhere a cluck cluck.
```

4.6 Another System, More Classes, More Interfaces

This section provides another example of implementing Java classes that are part of something bigger. As was done in Chapter 3, first a system is specified. From this problem specification, some objects are found that cannot be instantiated from existing Java classes—they are specific to this particular application. Two interfaces are implemented by two Java classes. Whereas the bank teller application presented in Chapter 3 will be implemented over many chapters, this system will not.

The student affairs office has decided to put some newfound activity fee funds toward a music jukebox in the student center. The jukebox will allow a student to play individual songs. No money will be required. Instead, a student will swipe a magnetic ID card through a card reader. Students will each be allowed to play up to 1,500 minutes of “free” jukebox music in their academic careers at the school. A student may select a CD from the available collection of CDs and then an individual song from that CD. A student is not allowed to play an entire CD.

An additional desired feature is to keep a top-10 list. So it will be necessary to keep track of how often each song is played.

When using an object-oriented approach, first find the objects. The nouns in the system specification are potential objects. Here is a team’s first pass at finding real-world objects.³ In reality, some of these ended up as part of the bigger system and some were eliminated and another object was added later.

Potential Objects—A First Pass at Finding Objects

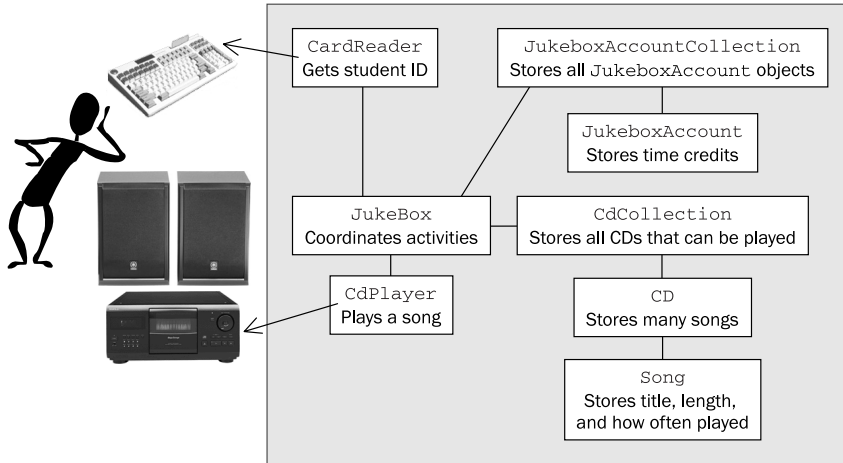
Somewhat Sure These Will Play a Role		Not So Sure, But Considering
jukebox	ID card	activity fee funds
student	CD	money
song	collection of CDs	jukebox music
card reader	top-10 list	

When this system was analyzed and designed, the team of programmers decided to use the name `JukeboxAccount` to represent the students that use the system. The name `Student` sounded more like an object related to the registrar’s office, with name, address, course taken, transcript, and so on. The team then

3. Many of the decisions on how to model this system’s objects are based on the work of some honors students and the author as they analyzed and designed this system in the spring of 1997. For the sake of brevity, these design decisions will be presented without explanation and captured as Java interfaces.

came up with the following picture to represent the major objects and their responsibilities.

Figure 4.3: Major objects to model the cashless jukebox (the big picture)



The team answered questions such as “What are other things the system must do?” and “Which object should take on a particular responsibility?” Responsibilities convey the purpose of an object and its role in the system. The proper assignment of responsibilities is one of the most important skills an object-oriented designer can develop. These responsibilities can be implemented as methods of a class. The responsibilities require some data along with a way to initialize the data.

While working on this system, the team members often reminded themselves that each object should be responsible for two things:

1. the actions the object can perform
2. the knowledge the object must remember

Therefore, the team was often asking these two questions:

1. What should the object be able to do?
2. What should the object know?

The team role-played in an effort to determine the responsibilities of each object. Role-playing happens when people act out the roles of objects. The team members became the objects. During the role-play, notes were taken. The class designs were summarized as Java interfaces—collections of method headings that need to be implemented by Java classes.

The Song Class

The `CdCollection` is a collection of CDs, which are in turn collections of Songs. The programming team first designed the simplest of these three classes: `Song`. It was decided that each `Song` should know its title and its playing time (in minutes and seconds, perhaps). In addition, to help maintain a top-10 list, each `Song` should keep track of how often it is played. Here is the resulting interface for `Song`:

```
public interface SongInterface
{
    // Provide access to the title of this Song
    public String getTitle( );

    // Find out how long it takes to play this song on a CD player
    public int getPlaytime( );

    // Notify this Song that it has been played
    public void recordOnePlay( );

    // Find out how often this Song has been played
    public int getTimesPlayed( );
}
```

Much analysis and design is usually done before arriving at a Java interface. Writing interfaces is a topic beyond the scope of this book. Instead, you will occasionally be given an interface and asked to write a class that implements the interface. In the chapter that follows, you will have to implement one interface (`ActionListener`) with one method (`ActionPerformed`), so your programs can respond to button clicks and text-field input. In Chapter 6, “Selection,” you will implement a different interface (`Comparable`) with one method (`compareTo`) to allow an `ArrayList` to have its elements arranged in order.

A Java interface conveniently summarizes what the objects of a class must be able to do. If you are new to programming, the interface provides the important, yet difficult-to-design, method headings. However, for now, two interfaces will be given. Then two classes can be more easily written to implement the interface. Here is one algorithm for implementing interfaces (other algorithms will also work):

1. Write the class heading with the keyword **implements** followed by the interface name.
2. Write the instance variables that are needed (this may not always be necessary).
3. Write the constructor to initialize the instance variables (this may not always be necessary).
4. Write the methods using the same exact method headings provided in the interface, replacing `;` with the correct actions written between `{` and `}`.

5. Test the class. If one is not given to you, write a test driver (a main method) that constructs some objects and sends every possible message to the objects.

1. Write the class heading.

The class heading consists of `public class` and the class name `Song`. The constructor, the instance variables, and the methods will go between the curly braces. To help you implement the class, add the keyword `implements` followed by the name of the interface that you are implementing, in this case, `implements SongInterface`.

```
// File name: Song.java
public class Song implements SongInterface
{
}
```

2. Add the instance variables.

The programming team wondered about how to store the playing time. How should this value be stored? Is a `Time` object needed? The team decided that a `Song`'s playing time could simply be stored as an `int` to represent the total number of seconds. This may require occasional conversions, such as total seconds into minutes and seconds. However, it will be easier to deduct playing time when dealing with seconds than do "time arithmetic." An `int` variable can store the number of seconds required to play the track. This is why the return value from `getPlayTime` was specified as an `int`.

With the title of a `Song` stored as a `String`, the playing time of one track stored as an `int`, and an `int` to keep track of how often a song is played, the instance variables can now be added.

```
// Instance variables
private String my_title;
private int my_playTime; // Seconds required to play this song
private int my_timesPlayed; // How often this song was played
```

These three values will represent the state of each `Song` object. This is what each instance of the class must remember.

3. Write the constructor.

To initialize a `Song` object, you need to pass the title and playing time as arguments. An object construction for the first track on a Beatle's greatest hits album that has a playing time of 2 minutes and 49 seconds and that has been played once could look like the following:

```
Song selection = new Song( "Day Tripper", (2 * 60 + 49), 1 );
```

The constructor needs the `String` parameter to initialize the title of the song. The second argument is an `int` parameter for initializing a `Song`'s playing time in seconds. The third argument specifies how often the song has already been played.

The constructor can now be written. It will initialize the state of any `Song` object to the values supplied as arguments. Constructors often initialize instance variables.

```
public Song( String trackTitle, int playTime, int timesPlayed )
{
    my_title = trackTitle;
    my_playTime = playTime;
    my_timesPlayed = timesPlayed;
}
```

4. Write the methods.

Even after adding instance variables and the constructor, the `Song` class that implements the `SongInterface` will not compile. It does not have the two methods of `SongInterface`. By adding `implements SongInterface`, the programmer is promising that any instance of this class will have the methods specified in the interface. If you don't write all method headings exactly as specified, the compiler will complain.

You can continue implementing the interface with a class by retyping the method headings as specified in the interface and replacing `;` with `{ }`. Remember to write all methods specified by the interface, even if they do nothing at first. The complete implementation of all methods will be shown in the completed class below.

If you do not have all methods specified in the interface, you will receive a compiletime error message like this:

```
Song should be declared abstract; it does not define recordOnePlay()
```

The same error appears even if you have the method heading the same except for one detail, such as a return type, one difference in parameters, or a case sensitivity issue.

5. Test the class.

When implementing a class, you should test it immediately. The testing can be done by adding a `main` method to the class. In this `main` method, one or more instances can be constructed. At a minimum, every possible message should be sent to at least one object. The output from running the program can then be examined. Verify that each method is doing what you want. A method like this with the sole purpose of testing a class is called a **test driver**. It can be a `main` method in a separate class or a `main` method in the class itself. In the `Song` class, the test driver is written in the classes that it is testing. Java allows every class to have one `main` method.

```
// A test driver may be a method inside the class that it is testing
public static void main( String[] args )
{ // Test drive Song
    Song t1 = new Song( "Day Tripper", (2 * 60) + 49, 0 );
    Song t2 = new Song( "We Can Work It Out", (2 * 60) + 15, 35 );
    // .. continue ...
```

Putting this all together and adding comments results in the following Java class:

```
// Objects of this class store information for one track on one musical
// CD. The methods return information about the state of the Song objects.

public class Song implements SongInterface
{
    // Instance variables
    private String my_title;
    private int my_playTime;    // Seconds required to play the track
    private int my_timesPlayed; // How often this song has been chosen

    // Construct Song objects so each knows its title and playing time
    // trackTitle: The title of the song
    // playTime: The number of seconds required to play the song
    public Song( String trackTitle, int playTime, int timesPlayed )
    {
        my_title = trackTitle;
        my_playTime = playTime;
        my_timesPlayed = timesPlayed;
    }

    // Provide access to the title of this Song
    public String getTitle( )
    {
        return my_title;
    }

    // Find out how long it takes to play this song on a CD player
    public int getPlaytime( )
    {
        return my_playTime;
    }

    // Notify this Song that it has been played
    public void recordOnePlay( )
    {
        my_timesPlayed = my_timesPlayed + 1;
    }

    // Find out how often this Song has been played
    public int getTimesPlayed( )
    {
        return my_timesPlayed;
    }

    // Provide a peek at the state of this object.
    // Return the title and playing time (minutes:seconds) for this song.
    public String toString( )
    {
        return my_title + " " + my_playTime / 60 + ":" + my_playTime % 60;
    }
}
```

```

public static void main( String[] args )
{ // Test drive Song
  Song t1 = new Song( "Day Tripper", (2 * 60) + 49, 0 );
  Song t2 = new Song( "We Can Work It Out", (2 * 60) + 15, 35 );
  Song t3 = new Song( "Paperback Writer", (2 * 60) + 18, 0 );

  System.out.println( "t3's title: " + t3.getTitle( ) );
  System.out.println( "t3's playing time in seconds: " +
    t3.getPlaytime( ) );

  System.out.println( );
  t1.recordOnePlay( );
  t1.recordOnePlay( ); // t1 was played twice
  t3.recordOnePlay( );
  System.out.println( "t1's times played: " + t1.getTimesPlayed( ) );
  System.out.println( "t2's times played: " + t2.getTimesPlayed( ) );
  System.out.println( "t3's times played: " + t3.getTimesPlayed( ) );
  System.out.println( );
  System.out.println( t1.toString( ) );
  System.out.println( t2.toString( ) );
  System.out.println( t3.toString( ) );
}

} // End the Song class

```

The test driver constructs three `Song` objects, sends all possible accessing and modifying messages to them, and at the end displays the `toString` version of each. When the above class runs, it generates the following output.

Output

```

t3's title: Paperback Writer
t3's playing time in seconds: 138

t1's times played: 2
t2's times played: 35
t3's times played: 1

Day Tripper 2:49
We Can Work It Out 2:15
Paperback Writer 2:18

```

As with any class that implements an interface, there may be additional methods than those that you are required to implement. One method that is available to virtually all objects in Java is `toString`. A `toString` method allows you to easily print an object. A `toString` method provides a snapshot of an object's state. The Java API has this to say about `toString`:

`toString` returns a string representation of the object. In general, the `toString` method returns a string that “textually represents” this object. The result should be a concise but informative representation that is

easy for a person to read. It is recommended that all classes have this method.⁴

The test diver uses the `toString` method to provide a view of the changing state of objects. The test driver helps avoid errors that are more difficult to detect later on when more than one type of object exists. As much as possible, test all of the methods in a class to verify that it does what it is supposed to do. Let's now apply the same steps for implementing a given interface for another class for the Jukebox system.

The JukeboxAccount Class

After much discussion, the team decided all `JukeboxAccount` objects should have the following responsibilities:

1. `debitOneSong`: adjust the `JukeboxAccount` to reflect playing one song today
2. `canSelect`: return `true` if the user is allowed to play a song or `false` if the user is not

The team also decided that each `JukeboxAccount` should remember:

- the remaining time credit
- a way to identify the account with a unique `String` ID

The team considered the return type and parameters for the `debitOneSong` message that modifies the state. During a `debitOneSong` message, the `JukeboxAccount` will add one to how many songs the person has played. At the same time, the state of any `JukeboxAccount` object will be modified by reducing the time remaining by the amount of the playing time for the song that was played. This means `debitOneSong` needs to ask the `Song` object for its playing time. Therefore, `debitOneSong` was designed to have a `Song` parameter so it could accept any `Song` object. A `debitOneSong` message modifies `JukeboxAccount` objects. The method need not return anything. Therefore, the return type is specified as `void`.

After much heated debate, the team decided that `JukeboxAccount` should be responsible for knowing if it could play a song. After all, each `JukeboxAccount` object remembers how many seconds of playing time it has remaining.⁵ The return value from this `canSelect` message was specified as a `boolean`, so the result is either `true` or `false`. To summarize, here is the interface captured by the team for a `JukeboxAccount`:

4. Java™ 2 Platform, Standard Edition, v 1.3.1 API Specification.

5. When trying to determine who should be responsible for some action, assign the responsibility to the information expert. This object has the information necessary to fulfill the responsibility—Expert Pattern from Craig Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design* (Upper Saddle River, NJ: Prentice Hall, 1998).

```
public interface JukeboxAccountInterface
{
    // Provide access to this JukeboxAccount's ID string
    public String getID( );

    // A message that should be sent whenever
    // this JukeboxAccount plays one song
    public void debitOneSong( Song aSong );

    // Return true if this user is allowed to play aSong
    public boolean canSelect( Song aSong );
}
```

1. Write the class heading.

The `JukeboxAccount` begins with writing the class heading: `public class`, the class name, and, in this case, implements `JukeboxAccountInterface`.

```
// File name: JukeboxAccount.java

public class JukeboxAccount implements JukeboxAccountInterface
{
    // This will not compile.
    // Wanted: All methods of JukeboxAccountInterface
}
```

2. Add the instance variables.

At this point, it is useful to recall what `JukeboxAccount` objects needed to remember: the remaining time credit and a way to identify the account with a unique identifier. The unique ID can be a `String`. How to store the remaining time credit wasn't so clear at first.

The initial time credit for a new `JukeboxAccount` is 1,500 minutes (or $1,500 * 60 = 90,000$ seconds). It was decided before that it would be easier to maintain seconds rather than minutes and seconds. Therefore, each `JukeboxAccount` object could have an instance variable named `my_secondsRemaining` to maintain the remaining time credit. Then the instance variable `my_secondsRemaining` will be reduced by the `Song`'s playing time each time a user plays a `Song`. We can use an `int` here unless we plan to increase it to over 2,147,483,647 seconds, which is about 596523 hours, or 24855 days, or 68 years of continuous music. An `int` instance variable will certainly suffice. Here are the three instance variables to store the state of many `JukeboxAccounts`.

```
private String my_ID;
private int my_secondsRemaining;
```

3. Write the constructor.

The constructor's job will be to initialize these two instance variables. One convention used in this textbook is to have the unique identifier appear first in the parameter list. However, there is no specific ordering required. The three param-

eters could be in a different order. Once the parameters are set, all object constructions must follow the correct ordering of arguments.

```
public JukeboxAccount( String ID, int secondsOfPlayTime )
{
    my_ID = ID;
    my_secondsRemaining = secondsOfPlayTime;
}
```

4. Write the methods.

Often, but not always, a class will have methods that simply return the values of instance variables. One such method is `getID`, which simply returns the value of the private instance variable. This message is used later to locate a particular account in the `JukeboxAccountCollection` after a user swipes a card.

```
public String getID( )
{
    return my_ID;
}
```

This `debitOneSong` method adjusts the time remaining to indicate that this `JukeboxAccount` has played a song of a specific duration. Notice that a `JukeboxAccount` object is sending a message to the `Song` object to get the particular playing time of the song.

```
public void debitOneSong( Song theSong )
{
    my_secondsRemaining = my_secondsRemaining - theSong.getPlaytime( );
}
```

The third method examines its own state to determine if it can play a song. The method should return `true` whenever the `JukeboxAccount` has enough time credit (in actuality, it will be some time before `canSelect` returns `false`).

```
public boolean canSelect( Song theSong )
{
    return my_secondsRemaining >= theSong.getPlaytime( );
}
```

5. Test the class.

As the `JukeboxAccount` class was implemented, the programmers compiled and tested often—approximately one compile, run, and test as each method was added. The test driver got quite big. A test driver should at least send all possible `JukeboxAccount` messages. The test driver is included in the `JukeboxAccount` class. It constructs an object that can no longer select a song by starting with only 15 minutes rather than 1,500 minutes.

Here is the `JukeboxAccount` class in its entirety:

```

// JukeboxAccount is part of a jukebox system. Programmers can:
// 1. Ask for the account ID.
// 2. Ask a JukeboxAccount if it can play a specific track.
// 3. Update time remaining and songs played with debitOneSong.
public class JukeboxAccount
{ // Instance variables
  private String my_ID;
  private int my_secondsRemaining;

  // Construct JukeboxAccount objects with two arguments.
  // ID: The string that uniquely identifies this account.
  // secondsOfPlayTime: Seconds remaining to play songs.
  public JukeboxAccount( String ID, int secondsOfPlayTime )
  {
    my_ID = ID;
    my_secondsRemaining = secondsOfPlayTime;
  }

  // Return true if this user is allowed to play theSong
  public boolean canSelect( Song theSong )
  {
    boolean result;
    result = my_secondsRemaining >= theSong.getPlaytime( );
    return result;
  }

  // Adjust the state when this JukeboxAccount plays one song
  public void debitOneSong( Song theSong )
  {
    my_secondsRemaining = my_secondsRemaining - theSong.getPlaytime( );
  }

  // Provide access to this JukeboxAccount's ID
  public String getID( )
  {
    return my_ID;
  }

  // Provide a quick way to see the state of this object.
  public String toString( )
  {
    String result;
    result = my_ID + " has " + ( my_secondsRemaining / 60 ) + " minutes";
    return result;
  }

  public static void main( String[] args )
  {
    // Start this account with only 10 minutes of time remaining
    JukeboxAccount anAccount = new JukeboxAccount( "Olson", 10 * 60 );

    System.out.println( "ID: " + anAccount.getID( ) );
    Song selection = new Song( "A 5 minute song", 5 * 60, 0 );
    System.out.println( "Select? " + anAccount.canSelect( selection ) );
    System.out.println( anAccount.toString( ) );
  }
}

```

```

        anAccount.debitOneSong( selection );
        System.out.println( anAccount.toString( ) );
        anAccount.debitOneSong( selection );
        System.out.println( anAccount.toString( ) );
        anAccount.debitOneSong( selection );
        System.out.println( anAccount.toString( ) );
        System.out.println( "Select? " + anAccount.canSelect( selection ) );
    } // End test driver

} // End class JukeboxAccount

```

The `toString` method for `JukeboxAccount` concatenates a few values together to create a string that “textually” represents any object. Each class you write should add a `toString` method even if it is not requested. As seen before in `println` messages, the `+` operator will make a `String` from a `String` and other values, such as integers. When the above class is run, it generates the following output:

Output

```

ID: Olson
Select? true
Olson has 10 minutes
Olson has 5 minutes
Olson has 0 minutes
Select? false

```

The preceding discussion offered an algorithm on how a class can implement a given interface. The end-of-chapter programming projects offer interfaces to help you write the requested classes. You have the option of implementing the interface or simply writing the class using the provided method headings. In either case, the test drivers provided in the projects ensure that you have the requested methods.

Chapter Summary

- This chapter showed a class as a collection of methods that represent the messages available for any instance of the class (object).
- Method headings with documentation provide the following information:
 - whether or not the method is available to the outside world (public or private)
 - the return type (the kind of value that a message will evaluate to)
 - the method name that begins a valid method call
 - the parameter list (the number and type of arguments required)
 - documentation describing what the method does
- The familiar assignment rules apply for assigning arguments to parameters.
- A Java class contains:
 - a class heading, which may include `implements`
 - the instance variables, known collectively as the state

- one or more constructors to initialize the state
- methods with parameters and return types
- implementation of those methods in the method body
- Each instance of a class may store many values, which may be of different types. For example, each `BankAccount` object stores a `String` instance variable for the ID and a number for the balance. Other objects have state that is more complex.
- Instance variables are accessed and modified through the methods of the class.
- Constructors are called to build and initialize objects like this:

```
BankAccount anotherAccount( "Calissario", 4320.10 );
```
- Some methods change the state of the object (for example, `add`, `setSize`, `withdraw`, and `move`).
- Some methods provide access to the state of an object (for example, `get`, `getBalance`, and `toString`).
- Some methods ask other objects for help to fulfill their responsibility (for example, `Transaction` methods ask `GregorianCalendar` for the current date).
- The ramifications of adhering to the guideline “All data should be hidden within its class” include the following:
 - Programmers cannot accidentally (or intentionally) mess up the state. The compiler will complain. The code will not run.
 - Programmers need to implement additional accessing methods, `getBalance()`, for example, but this is sound software engineering practice.
- The ramifications of adhering to the guideline “Keep related data and behavior in one place” include the following:
 - a more intuitive design (things make more sense)
 - code is easier to maintain
- Instance variables should be declared `private`.
- A class should be designed to exhibit high cohesion:
 - The data should be used by the methods.
 - The methods should have a meaningful relationship to each other.
- Responsibilities discovered during role-playing can be implemented as methods. These actions are captured as a Java class.
- Knowledge responsibilities can be implemented as instance variables.
- Most classes need public methods, private instance variables, and a constructor to initialize the state of an object.
- An interface cannot have any instance variables or constructors.
- If a class implements an interface, the class must implement all methods of the interface using the same method headings specified by the interface.
- A class that implements an interface may add new methods.

Key Terms

accessing method	implements	parameter
argument	instance variable	private
class constant	interface	public
cohesion	method	return
constructor	modifier	static
design guideline	modifying method	substring
final	object-oriented design	test driver

Exercises

The answers to exercises marked with 🌟 are available in Appendix C. Use the following class to do exercises 1 through 6:

```
public class SillyClass
{
    private double my_leftOperand;
    private double my_rightOperand;

    public SillyClass( double leftOperand, double rightOperand )
    { // Construct an object with two arguments
        my_leftOperand = leftOperand;
        my_rightOperand = rightOperand;
    }

    public double sum( )
    {
        return my_leftOperand + my_rightOperand;
    }

    public double product( )
    {
        return my_leftOperand * my_rightOperand;
    }

    public double quotient( )
    {
        return my_leftOperand / my_rightOperand;
    }
}
```

1. Name the instance variables of the `SillyClass` class.
2. Name the methods of `SillyClass`.
3. How many arguments are required to construct an instance of `SillyClass`?
4. Using the method headings and documentation for the `One` class above, write the output generated by the following program:

```

public class TestSilly
{
    public static void main( String[] args )
    {
        SillyClass a = new SillyClass( 1.2, 2.0 );
        SillyClass b = new SillyClass( 6.0, -3.0 );
        System.out.println( "a:" + a.sum( ) + " "
            + a.product( ) + " " + a.quotient( ) );
        System.out.println( "b:" + b.sum( ) + " " + b.product( )
            + " " + b.quotient( ) );
    }
}

```

5. Add a method named `difference` that returns the difference between the two operands of any `SillyClass` object. The return value can never be negative.
6. Send a difference message named `b` to the `SillyClass` object.



7. Which of the following represent valid method headings?

- a. `public int large(int a, int b)`
- b. `public double(double a, double b)`
- c. `public int f(int a; int b;)`
- d. `public int f(a, int b)`
- e. `public double f()`
- f. `public String c(String a)`



8. Given the `ClickCounter` class, predict the output generated by the test driver.

```

public class ClickCounter
{
    public static final int START_COUNT = 0;
    private int my_count;
    private int my_maxCount;

    // Constructor
    public ClickCounter( int maxCount )
    {
        my_count = START_COUNT;
        my_maxCount = maxCount;
    }

    // Change the count of this clickCounter.
    // If at maximum, reset to 0, otherwise add 1.
    public void click( )
    {
        my_count = ( my_count + 1 ) %
            ( my_maxCount + 1 );
    }

    // Evaluate to this clickCounter's count
    public int getCount( )
    { // Return the count of this ClickCounter
        return my_count;
    }
}

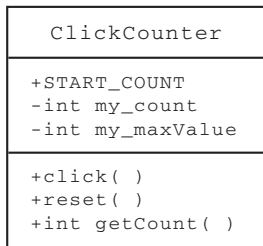
```



```

public static void main( String[] args )
{ // Test drive ClickCounter
  ClickCounter aCounter = new ClickCounter( 3 );
  // aCounter only counts from 0 to 3
  System.out.println( aCounter.getCount( ) );
  aCounter.click( );
  System.out.println( aCounter.getCount( ) );
  aCounter.click( );
  System.out.println( aCounter.getCount( ) );
  aCounter.click( );
  System.out.println( aCounter.getCount( ) );
  aCounter.click( );
  System.out.println( aCounter.getCount( ) );
  aCounter.click( );
  System.out.println( aCounter.getCount( ) );
}
}

```

Class Diagram



9.  With paper and pencil, add a reset method as if it were in the ClickCounter class. The method must always set my_count to 0.
10.  Send a reset message to the object constructed in the main method above.
11. Write the output when the Mystery class is run.

```

public class Mystery
{
  private int my_value;

  public Mystery( int v )
  {
    my_value = v;
  }

  public String toString( )
  {
    String result = "[" + my_value + "]; // Makes a string
    return result;
  }

  public static void main( String[] args )
  {
    Mystery aMystery = new Mystery( 93 );
  }
}

```

```

        System.out.println( aMystery.toString( ) );
        Mystery anotherMystery = new Mystery( -123 );
        System.out.println( anotherMystery );
    }
}

```



12. Using the Car and Mystery2 classes shown here, write the output generated by the main method in the CarAndMystery2 class.

```

public class CarAndMystery2
{
    public static void main( String[] args )
    {
        Car c1 = new Car( 0.0, 0.0 );
        Car c2 = new Car( 72.0, 3.0 );

        c1.measure( 100.0, 5.0 );
        System.out.println( "a " + c1.getMPG( ) );
        System.out.println( "b " + c2.getMPG( ) );

        Mystery2 m = new Mystery2( );
        c1.measure( 230.0, 10.0 );
        System.out.println( "c " + c1.getMPG( ) );
        m.huh( c1 );
        m.huh( c2 );
        System.out.println( "d " + m.huhTwo( ) );

        Car c3 = new Car( 128.0, 4.0 );
        m.huh( c3 );
        System.out.println( "e " + m.huhTwo( ) );
    }
}

public class Car
{
    private double my_miles;
    private double my_gallons;

    public Car( double miles, double gallons )
    {
        my_miles = miles;
        my_gallons = gallons;
    }

    public void measure( double miles, double gallons )
    {
        my_miles = my_miles + miles;
        my_gallons = my_gallons + gallons;
    }

    public double getMPG( )
    {
        return my_miles / my_gallons;
    }
}

```

```

public class Mystery2
{
    private double my_value;
    private int n;

    public Mystery2( )
    {
        my_value = 0.0;
        n = 0;
    }

    public void huh( Car aCar )
    {
        my_value = my_value + aCar.getMPG( );
        n = n + 1;
    }

    public double huhTwo( )
    {
        return my_value / n;
    }
}

```

Programming Tips

1. Make all instance variables private.

This protects the state of objects while hiding implementation details. Other programmers can use your class by looking at your documentation. Documented method headings provide information about how to use instances of the class. See Appendix A, “A Little Javadoc.”

2. Some programming projects require some author-supplied files.

These files are available on the accompanying diskette. To find any file in this textbook, remember that each public class is stored in a file that has the name.

class-name.java

Therefore, the `ClickCounter` class, which begins like this:

```
public class ClickCounter
```

is stored in a file named `ClickCounter.java`. Remember that Java is case sensitive. Errors occur when the file name and the class name differ in case.

3. Working with two or more files is more difficult than working with one.

Some programming projects require that you work with more than one file. This takes a little patience as you grow accustomed to working with multiple files. You will sometimes have a class in a file that will be used by another class in another file.

4. Consider building your test driver into each and every class.

When implementing classes, it is best to test each method as you go. This is facilitated by placing a `main` method in your class as you develop it—creating one file instead of two.

5. Classes are designed with values in mind.

When using someone else's class, remember that the design was influenced by what was important at the time to those particular programmers. This sometimes makes it more difficult to agree with the particular methods that were selected. When you become more familiar with the class or become aware of the values that influenced the design, the decisions may seem more logical.

6. Avoid magic numbers.

Whenever you need a numeric value in several places, consider writing it as a class constant that has a meaningful name. `SALES_TAX` is more meaningful than `0.045`, for example.

Programming Projects

4A LibraryBook

Write a `LibraryBook` class so the test driver below generates the output shown. A `LibraryBook` object stores the title and author of a library book (initialized by having the constructor take the book's title and author as arguments). The `toString` method must show the title and the borrower between `<` and `>` (see the output below). A `borrowBook` message changes the borrower's ID from `null` to the `String` argument passed to it. When the book is returned, the borrower is set back to `null`. The following test driver shows the behavior of one `LibraryBook` object. Run it to ensure that it generates the exact output shown with your class.

```
public class TestLibraryBook
{
    public static void main( String[] args )
    {
        LibraryBook aBook =
            new LibraryBook( "The Mythical Man Month", "Fred Brooks" );

        System.out.println( aBook.getTitle( ) );
        System.out.println( aBook.getAuthor( ) );
        System.out.println( aBook.toString( ) );
        System.out.println( );
        aBook.borrowBook( "Andrew Wilt" );
        System.out.println( aBook.toString( ) );
        aBook.returnBook( );
        System.out.println( aBook.toString( ) );
    }
}
```

Output

```
The Mythical Man Month
Fred Brooks
<The Mythical Man Month by Fred Brooks is borrowed by null>

<The Mythical Man Month by Fred Brooks is borrowed by Andrew Wilt>
<The Mythical Man Month by Fred Brooks is borrowed by null>
```

4B PersonWithAge

Write a `PersonWithAge` class so the test driver below generates the output shown. A `PersonWithAge` object stores a person's name and age. The `getName` method returns the person's name. The method named `yearsTo` returns how many years it takes to turn the age passed as an argument. The constructor takes the person's name and age as arguments. The class must also have a `toString` method to return the name and age of any person as a string (see the output below). The following test driver shows the behavior of a few `PersonWithAge` objects:

```
public class TestPersonWithAge
{
    public static void main( String[] args)
    { // Test drive PersonWithAge
        PersonWithAge one = new PersonWithAge( "Kim", 19 );
        PersonWithAge two = new PersonWithAge( "Devon", 21 );
        PersonWithAge tre = new PersonWithAge( "Chris", 79 );

        System.out.println( one.toString( ) );
        System.out.println( two.toString( ) );
        System.out.println( tre.toString( ) );
        System.out.println( );
        System.out.println( one.getName( ) + " turns 21 in "
            + one.yearsTo( 21 ) + " years" );

        System.out.println( two.getName( ) + " turns 40 in "
            + two.yearsTo( 40 ) + " years" );

        System.out.println( tre.getName( ) + " turns 65 in "
            + tre.yearsTo( 65 ) + " years" );
    }
}
```

Output

```
Kim is 19 years old
Devon is 21 years old
Chris is 79 years old

Kim turns 21 in 2 years
Devon turns 40 in 19 years
Chris turns 65 in -14 years
```

4C SimpleEmployee

Write a `SimpleEmployee` class so the test driver below generates the output shown. The constructor takes the employee's name (a `String`) and annual salary (a `double`) as arguments. The `toString` method returns a `String` that provides a textual representation of the object—the name and annual salary (see the output below). Make sure the salary has two decimal places and is preceded by \$ or whatever your currency symbol is. As for instance variables, each `SimpleEmployee` object must know the employee's name (a `String`) and annual salary (a `double`). Round the monthly salary and the annual salary to the nearest hundredth (multiply the monthly salary by 100.0, `Math.round` that result to the nearest integer, and then divide that by 100.0—this avoids division-by-zero errors when credits are 0.0). Run the following test driver and verify that you have all the methods specified. You must generate the output exactly as shown.

```
public class TestSimpleEmployee
{
    public static void main( String[] args )
    {
        SimpleEmployee empOne = new SimpleEmployee( "Jack", 52000.00 );
        SimpleEmployee empTwo = new SimpleEmployee( "Jill", 53000.00 );

        System.out.println( empTwo.getName( ) + " annual "
            + empTwo.getAnnualSalary( ) );

        System.out.println( empTwo.getName( ) + " monthly "
            + empTwo.getMonthlySalary( ) );

        System.out.println( empOne.getName( ) + " annual "
            + empOne.getAnnualSalary( ) );
        System.out.println( empOne.getName( ) + " monthly "
            + empOne.getMonthlySalary( ) );

        empTwo.giveRaise( 0.05 );
        empOne.giveRaise( 0.10 );
        System.out.println( empTwo.toString( ) );
        System.out.println( empOne.toString( ) );
    }
}
```

Output (the monthly salaries have two decimal places—use `DecimalFormat`)

```
Jill annual 53000.00
Jill monthly 4416.67
Jack annual 52000.00
Jack monthly 4333.33
Jill makes 55650.00 per year
Jack makes 57200.00 per year
```

4D Course and Student Implement Interfaces

Implement two interfaces with classes named `Student` and `Course` so they work together to maintain a student's grade point average (GPA). All `Student` objects must be able to compute their own GPA, which is

$$\text{GPA} = \text{totalQualityPoints} / \text{totalUnits}$$

where

$$\text{qualityPoints} = \text{units} * \text{numericGrade}$$

This means that the `recordCourse` method must increment the total units (credits) and `qualityPoints` represented by the `Course` arguments. The interfaces are given for both classes. In addition to all methods specified in these interfaces, add a `toString` method for each class and add a test driver as a main method.

```
// Instances of this class store data about one university course
// 1. Course number
// 2. Units (or credits) of this course: 0.5 - 15.0
// 3. Numeric grade: 0.0 - 4.0
public interface CourseInterface
{
    // Return the course number of this course
    public String getCourseNumber( );

    // Return the number of units of this course
    public double getUnits( );

    // Return the numeric grade for this course
    public double getGrade( );

    // Return a textual representation of this Course object
    public String toString( );
}

// This interface represents a student at a university with very limited
// functionality. An instance of this Student class only maintains
// the total number of units and quality points for each student.
public interface StudentInterface
{
    // An accessing method for this student's name
    public String getName( );

    // An accessing method for this student's current GPA
    public double getGPA( );

    // An accessing method to show a state of this student
    public String toString( );

    // A modifying method that adds a newly completed course
    public void recordCourse( Course aCourse );
}
```

The test driver below sends all possible messages. It, along with its output, is intended to help you understand the behavior of the objects. When you complete both classes, the test driver must generate the output exactly as shown. In addition to the methods specified in the interfaces above, remember to add a `toString` method to both classes. Use the test driver output shown below to determine how you should implement `toString`. Run this test driver to test your two new classes: `Course` and `Student`.

```
public class TestCourseAndStudent
{
    // This is a test driver for Course and Student
    public static void main( String[] args )
    {
        // Test Course--start with a three-unit A
        Course CS1 = new Course( "CSc127A-042002", 3.0, 4.0 );

        System.out.println( CS1 );
        System.out.println( "units: " + CS1.getUnits( ) );
        System.out.println( "grade: " + CS1.getGrade( ) );

        // Construct a four-unit B
        Course CS2 = new Course( "CSc127A-012003", 4.0, 3.0 );
        // Construct a one-unit C
        Course PE101 = new Course( "PE101-022003", 1.0, 2.0 );
        System.out.println( );

        // Test the relationship between Student and Course
        Student s1 = new Student( "Devon" );
        s1.recordCourse( CS1 );
        System.out.println( s1 );
        System.out.println( "name: " + s1.getName( ) );
        System.out.println( " GPA: " + s1.getGPA( ) );
        System.out.println( );
        // Add two more courses
        s1.recordCourse( CS2 );
        s1.recordCourse( PE101 );
        System.out.println( s1.toString( ) );
    }
}
```

Output

```
CSc127A-042002 3.0 units with grade 4.0
units: 3.0
grade: 4.0
```

```
Devon's GPA is 4.0
name: Devon
GPA: 4.0
```

```
Devon's GPA is 3.25
```

4E A Few Presents

Write three classes named `First`, `Second`, and `Third` in three separate files that ensure the following program runs with no syntax errors. It must generate the output exactly as shown. Your classes will not need a constructor or instance variables. Here is the interface your classes need to implement. (*Note:* Class constants can also be declared in interfaces; they must have the `static` modifier.)

```
// File name: Present.java
public interface Present
{
    public static final String REFRAIN =
        " of Christmas, my true love gave to me: ";

    public String day( );
    public String whatSheGaveToMe( );
}

// File name: ThreeDays.java
public class ThreeDays
{
    public static void main( String[] args )
    {
        ChristmasPresent day1 = new First( "first" );
        ChristmasPresent day2 = new Second( "second" );
        ChristmasPresent day3 = new Third( "third" );
        // Verse 1
        System.out.println( "On the " + day1.day( ) + Present.REFRAIN );
        System.out.println( day1.whatSheGaveToMe( ) );
        // Verse 2
        System.out.println( "On the " + day2.day( ) + Present.REFRAIN );
        System.out.println( day2.whatSheGaveToMe( ) + ", " );
        System.out.println( "and " + day1.whatSheGaveToMe( ) );
        // Verse 3
        System.out.println( "On the " + day3.day( ) + Present.REFRAIN );
        System.out.println( day3.whatSheGaveToMe( ) + ", " );
        System.out.println( day2.whatSheGaveToMe( ) + ", " );
        System.out.println( "and " + day1.whatSheGaveToMe( ) );
    }
}
```

Output

```
On the first day of Christmas, my true love gave to me:
a partridge in a pair tree.
On the second day of Christmas, my true love gave to me:
two turtle doves,
and a partridge in a pair tree.
On the third day of Christmas, my true love gave to me:
three French hens,
two turtle doves,
and a partridge in a pair tree.
```

4F Parking Garage Ticket

Write a `ParkingGarageTicket` class so the test driver given below generates the output shown. The constructor takes a ticket number (an `int`) as a parameter; `getFees` takes two arguments, the current hour and the minute the ticket will be paid. The cost to park is \$1.50 per hour, whether 1 minute or 60 minutes of the hour is used. One hour and 1 minute of parking would cost \$3.00. Assume all times are on the same day. The `toString()` method must use the `GregorianCalendar` `getTime()` method to return the day, month, time, and year.

```
public static void main (String[] args)
{
    ParkingGarageTicket ticket = new ParkingGarageTicket( 101 );
    System.out.println( ticket.toString( ) );
    System.out.println( "Test getFee " + ticket.getTicketNumber( ) );

    GregorianCalendar time = new GregorianCalendar( );
    int hour = time.get( GregorianCalendar.HOUR_OF_DAY ); // 0..23
    int minute = time.get( GregorianCalendar.MINUTE ); // 0..59
    System.out.println( "1.5? " + ticket.getFee( hour, minute ) );
    System.out.println( "3.0? " + ticket.getFee( hour + 1, minute ) );
    System.out.println( "7.5? " + ticket.getFee( hour + 4, minute ) );
    System.out.println( "34.5? " + ticket.getFee( hour + 22, minute ) );
}
```

Output

```
Ticket#101 arrived Wed Jan 30 19:44:14 MST 2002
Test getFee 101
1.5? 1.5
3.0? 3.0
7.5? 7.5
34.5? 34.5
```

4G Designing WeeklyEmployee

Completely implement a class named `WeeklyEmployee` that is intended to be part of a payroll system (no interfaces are required in this code). Each instance of `WeeklyEmployee` must keep track of an employee's name, number (an `int`), hourly rate of pay, and number of hours worked during the current week. Each instance of `WeeklyEmployee` must be able to compute its own:

1. gross pay, which is defined as (hours worked * hourly rate of pay)
2. social security tax, which is defined as 6.2% of gross pay
3. Medicare tax, which is defined as 1.45% of gross pay

There is no overtime. A programmer must be able to set the hours worked each week since this changes from week to week. Because the time clock measures hours worked in tenths of an hour, hours worked must have a fractional part (you could

make this a double). In addition, a programmer using this class must be able to give any `WeeklyEmployee` object a raise in pay such as 0.03 for 3%, for example.

1. **Determine what each `WeeklyEmployee` must be able to do.** On paper, write a method heading for each responsibility and a comment indicating what the method does.
2. **Determine what each `WeeklyEmployee` must know.** Provide a name for each value and also the type (such as `int`, `double`, or `String`). Use meaningful identifiers that accurately describe what the variables will store.

<u>Type of Value</u>	<u>Variable Name</u>

3. **Write a class diagram.** This is a rectangle with the class heading at the top, instance variables next, followed by the method headings (use Java syntax for the method headings).
4. **Using the previous documents as guidelines, complete the `WeeklyEmployee` class as Java source code.** Remember to include all of these elements:
 - a. class heading
 - b. public class constants for 0.062 and 0.0145
 - c. instance variables
 - d. constructor to initialize the instance variables
 - e. all other methods
5. **Test the class.** Add a `main` method to your class that at a minimum constructs two instances of `WeeklyEmployee` and send every possible message to both objects. Keep modifying and fixing `WeeklyEmployee` as you test your class. Generate output that verifies that your objects are behaving correctly.

Answers to Self-Check Questions

- 4-1 -a `String` -d `String`
 -b `concat` -e There is no second argument possible.
 -c `one`
- 4-2 -a `"abcxyz"` -d Error—one too many arguments.
 -b Error—missing an argument. -e `"abcwxyz"`
 -c Error—an incorrect argument. -f Error—`concat` requires an object reference, not a class name.
- 4-3 `"times"`
 4-4 `"hopeless"`
 4-5 `""` (an empty string with length 0)

4-6 Instance variables: 1.23 4.56
Instance variables: 1.0 2.0

4-7 Changed to: 2.2 4.4
Changed to: 4.2 6.4
Changed to: 7.2 9.4

4-8 First: 1.0
Second: 2.0
Sum: 3.0
First: 4.4
Second: 5.4
Sum: 9.8

4-9

```

    sc1

    my_first = 4.4
    my_second = 5.4
```

4-10 LibraryBook

4-11 borrowBook getBorrower returnBook

4-12 a String (the borrower's name)

4-13 none (or void)

4-14 a String (the borrower's name)

4-15 Two: the author (a String) and the book title (a String)

4-16 LibraryBook aBook("Computing Fundamentals with Java", "Rick Mercer");

4-17 aBook.borrowBook("AnyFirstName AnyLastName");

4-18 System.out.println(aBook.getBorrower());

4-19 **Immediately after constructing**

Immediately after borrowBook message

```

    aBook

    my_author = "John LeCarre"
    my_title = "Little Drummer Girl"
    my_borrower = null
```

```

    aBook

    my_author = "John LeCarre"
    my_title = "Little Drummer Girl"
    my_borrower = "Chris Miller"
```

4-20 Borrower: Chris Miller
Borrower: null

4-21 The constructors

4-22 They allow access to the state of any object so humans or other objects can either inspect or use that state. An accessing method may directly return the value of a primitive instance variable or a reference to an object instance variable. Or the method may have to perform a little or a lot of processing to return some useful information about the state of the object. The method may even send messages to other objects.

4-23 Modify the state of the object.

4-24 Allow programmers to initialize objects with either the default state or their own initial values.

4-25 They store the state (the values) of any object. Each instance of the class stores its own copy of the instance variables. If there are 25,000 BankAccount objects, there are 25,000 IDs and balances.

4-26 Yes

4-27 No. The private instance variables are known only to the methods inside the class.

4-28 public static final int DAYS_IN_LEAP_YEAR = 366;

4-29 No

- 4-30 Yes
- 4-31 Methods declared with the `public` modifier are known in all methods of the class and in every block where an instance of the class is constructed or is specified as a parameter.
- 4-32 Methods declared with the `public` modifier can be accessed only by the methods of that class. They are known in every part of the class.
- 4-33 It prevents accidental modification of an object's state. The code relies on the `public` methods, not the internal implementation details. There is a design guideline for this.
- 4-34 No. The code does not access `private` data. This is another benefit of `private` instance variables—programmers do not need to know them and do not rely on them in case someone changes the implementation of the class.
- 4-35 Yes. This is why it is a good idea to design a class so carefully that it rarely, if ever, needs to be changed.
- 4-36 No. That would not be a use of high cohesion. The message is not related to a `BankAccount`.

```
4-37 public class Chicken implements BarnyardAnimal
{
    private String mySound;

    public Chicken( String sound )
    {
        mySound = sound;
    }

    public String sound( )
    {
        return mySound;
    }
}
```

```
public class Cow implements BarnyardAnimal
{
    private String mySound;

    public Cow( String sound )
    {
        mySound = sound;
    }

    public String sound( )
    {
        return mySound;
    }
}
```