

THE NEXT STEP WITH JAVA

Unit 2

Chapter 6 3.5 hrs.
Introduction to Defining Classes

Chapter 7 5 hrs.
Control Statements Continued

Chapter 8 3.5 hrs.
Improving the User Interface

Chapter 9 3.5 hrs.
Introduction to HTML and Applets



Estimated Time for Unit: 15.5 hours

INTRODUCTION TO DEFINING CLASSES

OBJECTIVES

Upon completion of this chapter, you should be able to:

- Design and implement a simple class from user requirements.
- Organize a program in terms of a view class and a model class.
- Use visibility modifiers to make methods visible to clients and restrict access to data within a class.
- Write appropriate mutator methods, accessor methods, and constructors for a class.
- Understand how parameters transmit data to methods.
- Use instance variables, local variables, and parameters appropriately.
- Organize a complex task in terms of helper methods.

Estimated Time: 3.5 hours

VOCABULARY

Accessor
Actual parameter
Behavior
Constructor
Encapsulation
Formal parameter
Helper method
Identity
Instantiation
Lifetime
Mutator
Scope
State
Visibility modifier

We introduced basic object-oriented terminology in Chapter 1 and have used it repeatedly since then. Until now, we have focused on choosing among predefined classes to solve problems. We have shown how to declare variables of different classes, assign objects to these variables, and send them messages. In this chapter, we explore the internal workings of objects. We introduce the basic structure of class definitions so that you will be able to read and modify classes and create classes of your own. We restrict our focus to a few simple concepts and add more detail in later chapters.

6.1 The Internal Structure of Classes and Objects

As we stated in Chapter 1, an object is a runtime entity that contains data and responds to messages. A class is a software package or template that describes the characteristics of similar objects. These characteristics are of two sorts: variable declarations that define an object's data requirements (instance variables) and methods that define its behavior in response to messages.

The combining of data and behavior into a single software package is called *encapsulation*. An object is an instance of its class, and the process of creating a new object is called *instantiation*.

Classes, Objects, and Computer Memory

We begin our discussion of classes and objects by considering how the Java virtual machine (JVM) handles them. When a Java program is executing, the computer's memory must hold

- All class templates in their compiled form
- Variables that refer to objects
- Objects as needed

Each method's compiled byte code is stored in memory as part of its class's template. Memory for data, on the other hand, is allocated within objects. Although all class templates are in memory at all times, individual objects come and go. An object first appears and occupies memory when it is instantiated, and it disappears automatically when no longer needed. The JVM knows if an object is in use by keeping track of whether or not there are any variables referencing it. Because unreferenced objects cannot be used, Java assumes that it is okay to delete them from memory. Java does this during a process called *garbage collection*. In contrast, C++ programmers have the onerous responsibility of deleting objects explicitly. Forgetting to delete unneeded objects wastes scarce memory resources, and accidentally deleting an object too soon or more than once can cause programs to crash. In large programs, these mistakes are easy to make and difficult to find. Fortunately, Java programmers do not have to worry about this problem.

Three Characteristics of an Object

Three characteristics of objects must be emphasized. First, an object has *behavior* as defined by the methods of its class. Second, an object has *state*, which is another way of saying that at any particular moment its instance variables have particular values. Typically, the state changes over time in response to messages sent to the object. Third, an object has its own unique *identity*, which distinguishes it from all other objects in the computer's memory, even those that might momentarily have the same state. An object's identity is handled behind the scenes by the JVM and should not be confused with the variables that might refer to the object. Of the variables, there can be none, one, or several. When there are none, the garbage collector purges the object from memory. Shortly, we will see an example in which two variables refer to the same object.

Clients, Servers, and Interfaces

When messages are sent, two objects are involved—the sender and the receiver, also called the *client* and the *server*, respectively. A client's interactions with a server are limited to sending it messages, so consequently a client needs to know nothing about the internal workings of a server. A client needs to know only a server's *interface*, that is, the list of the methods supported by the server. The server's data requirements and the implementation of its methods are hidden from the client, an approach we referred to as *information hiding* in Chapter 1. Only the person who writes a class needs to understand its internal workings. In fact, a class's implementation details can be changed radically without affecting any of its clients, provided its interface remains the same.

EXERCISE 6.1

1. What is the difference between a class and an object?
2. What happens to an object's memory storage when it is no longer referenced by a variable?
3. List the three important characteristics of an object.
4. Describe the client-server relationship.
5. What is the interface of a class?

6.2 A Student Class

The first class we develop in this chapter is called `Student`. We begin by considering the class from a client's perspective. Later we show its implementation. From a client's perspective, it is enough to know that a `Student` object stores a name and three test scores and responds to the messages shown in Table 6-1.

TABLE 6-1
The interface for the `Student` class

METHODS	DESCRIPTIONS
<code>void setName(aString)</code>	Example: <code>stu.setName ("Bill");</code> sets the name of <code>stu</code> to Bill
<code>String getName()</code>	Example: <code>str = stu.getName();</code> returns the name of <code>stu</code>
<code>void setScore (whichTest, testScore)</code>	Example: <code>stu.setScore (3, 95);</code> sets the score on test 3 to 95 if <code>whichTest</code> is not 1, 2, or 3, then 3 is substituted automatically
<code>int getScore(whichTest)</code>	Example: <code>score = stu.getScore (3);</code> returns the score on test 3 if <code>whichTest</code> is not 1, 2, or 3, then 3 is substituted automatically
<code>int getAverage()</code>	Example: <code>average = stu.getAverage();</code> returns the average of the test scores
<code>int getHighScore()</code>	Example: <code>highScore = stu.getHighScore();</code> returns the highest test score
<code>String toString()</code>	Example: <code>str = stu.toString();</code> returns a string containing the student's name and test scores

Using Student Objects

Here is some code that illustrates how a client instantiates and manipulates `Student` objects. First, we declare several variables, including two variables of type `Student`.

```
Student s1, s2;           // Declare the variables
String str;
int i;
```

As usual, we do not use variables until we have assigned them initial values. We assign a new `Student` object to `s1` using the operator `new`:

```
s1 = new Student();       // Instantiate a student and associate it with the
                          // variable s1
```

It is important to emphasize that the variable `s1` is a reference to a `Student` object and is *not* a `Student` object itself.

A `Student` object keeps track of the name and test scores of an actual student. Thus, for a brand new `Student` object, what are the values of these data attributes? That depends on the class's internal implementation details, but we can find out easily by sending messages to the `Student` object via its associated variable `s1`:

```
str = s1.getName();
System.out.println (str);    // yields ""

i = s1.getHighScore();
System.out.println (i);      // yields 0
```

Apparently, the name was initialized to an empty string and the test scores to zero. Now we set the object's data attributes by sending it some messages:

```
s1.setName ("Bill");        // Set the student's name to "Bill"
s1.setScore (1,84);          // Set the score on test 1 to 84
s1.setScore (2,86);          //                               on test 2 to 86
s1.setScore (3,88);          //                               on test 3 to 88
```

Messages that change an object's state are called *mutators*. To see if the mutators worked correctly, we use other messages to access the object's state (called *accessors*):

```
str = s1.getName();          // str equals "Bill"
i = s1.getScore (1);          // i equals 84
i = s1.getHighScore();        // i equals 88
i = s1.getAverage();          // i equals 86
```

The object's string representation is obtained by sending the `toString` message to the object:

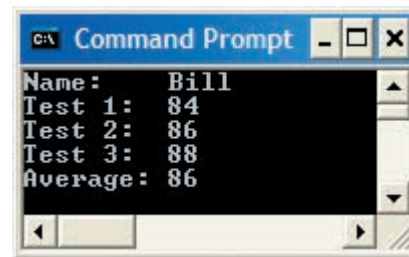
```
str = s1.toString();
// str now equals
// "Name:    Bill\nTest 1:  84\nTest2:  86\nTest3:  88\nAverage: 86"
```


When displayed in a terminal window (Figure 6-1), the string is broken into several lines as determined by the placement of the newline characters ('`\n`'). In addition to the explicit use of the `toString` method, there are other situations in which the method is called automatically. For instance, `toString` is called implicitly when a `Student` object is concatenated with a string or is an argument to the method `println`:

```
str = "The best student is: \n" + s1;
// Equivalent to: str = "The best student is: \n" + s1.toString();
System.out.println (s1);
// Equivalent to: System.out.println (s1.toString());
```

FIGURE 6-1

Implicit use of `toString` when a `Student` object is sent to a terminal window



Because of these valuable implicit uses of the `toString` method, we frequently include this method in the classes we write. If we forget, however, Java provides a very simple version of the method through the mechanism of inheritance (mentioned in Chapter 1). The simplified version does little more than return the name of the class to which the object belongs.

Objects, Assignment, and Aliasing

We close this demonstration by associating a `Student` object with the variable `s2`. Rather than instantiating a new student, we assign `s1` to `s2`:

```
s2 = s1; // s1 and s2 now refer to the same student
```

The variables `s1` and `s2` now refer to the *same* `Student` object. This might come as a surprise because we might reasonably expect the assignment statement to create a second `Student` object equal to the first, but that is not how Java works. To demonstrate that `s1` and `s2` now refer to the same object, we change the student's name using `s2` and retrieve the same name using `s1`:

```
s2.setName ("Ann"); // Set the name
str = s1.getName(); // str equals "Ann". Therefore, s1 and s2 refer
// to the same object.
```

Table 6-2 shows code and diagrams that clarify the manner in which variables are affected by assignment statements. At any time, it is possible to break the connection between a variable and the object it references. Simply assign the value `null` to the variable:

```
Student s1;
s1 = new Student(); // s1 references the newly instantiated student
... // Do stuff with the student
s1 = null; // s1 no longer references anything
```

TABLE 6-2

How variables are affected by assignment statements

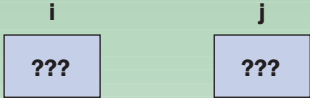
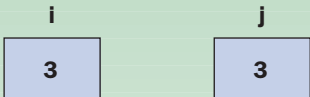
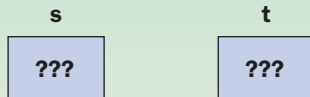
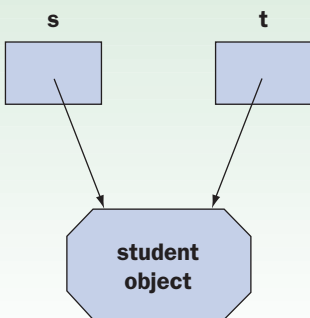
CODE	DIAGRAM	COMMENTS
<code>int i, j;</code>		i and j are memory locations that have not yet been initialized, but which will hold integers.
<code>i = 3;</code> <code>j = i;</code>		i holds the integer 3. j holds the integer 3.
<code>Student s, t;</code>		s and t are memory locations that have not yet been initialized, but which will hold references to Student objects.
<code>s = new Student();</code> <code>t = s;</code>		s holds a reference to a Student object. t holds a reference to the same Student object.

Table 6-2 demonstrates that assignments to variables of numeric types such as `int` produce genuine copies, whereas assignments to variables of object types do not.

Primitive Types, Reference Types, and the `null` Value

We mentioned earlier that two or more variables can refer to the same object. To better understand why this is possible, we need to consider how Java classifies types. In Java, all types fall into two fundamental categories:

1. *Primitive types*: `int`, `double`, `boolean`, `char`, and the shorter and longer versions of these
2. *Reference types*: all classes, for instance, `String`, `Student`, `Scanner`, and so on

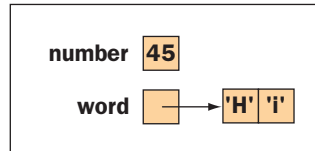
As we first pointed out in Chapter 2, variables in these two categories are represented differently in memory. A variable of a primitive type is best viewed as a box that contains a value of that primitive type. In contrast, a variable of a reference type is thought of as a box that contains

a pointer to an object. Thus, the state of memory after the following code is executed is shown in Figure 6-2.

```
int number = 45;
String word = "Hi";
```

FIGURE 6-2

The difference between primitive and reference variables

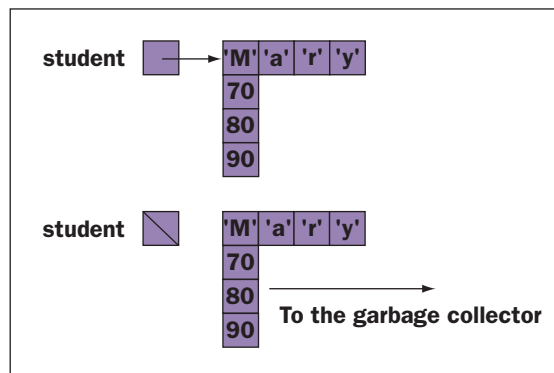


As previously mentioned, reference variables can be assigned the value `null`. If a reference variable previously pointed to an object, and no other variable currently points to that object, the computer reclaims the object's memory during garbage collection. This situation is illustrated in the following code segment and in Figure 6-3:

```
Student student = new Student("Mary", 70, 80, 90);
student = null;
```

FIGURE 6-3

The Student variable before and after it has been assigned the value `null`



A reference variable can be compared to the `null` value, as follows:

```
if (student == null)
    ...           // Don't try to run a method with that student!
else
    ...           // Process the student

while (student != null){
    ...           // Process the student
    ...           // Obtain the next student from whatever source
}
```


As we already know from Chapter 3, when a program attempts to run a method with an object that is *null*, Java throws a *null pointer exception*, as in the following example:

```
String str = null;

System.out.println (str.length()); // OOPS! str is null, so Java throws a
                                   // null pointer exception
```

The Structure of a Class Template

Having explored the `Student` class from a client's perspective, we now address the question of how to implement it. All classes have a similar structure consisting of four parts:

1. The class's name and some modifying phrases
2. A description of the instance variables
3. One or more methods that indicate how to initialize a new object (called *constructor* methods)
4. One or more methods that specify how an object responds to messages

The order of these parts can be varied arbitrarily provided part 1 (the class's name) comes first; however, for the sake of consistency, we will usually adhere to the order listed, which yields the following class template:

```
public class <name of class> extends <some other class>{

    // Declaration of instance variables
    private <type> <name>;
    ...

    // Code for the constructor methods
    public <name of class>() {
        // Initialize the instance variables
        ...
    }
    ...

    // Code for the other methods
    public <return type> <name of method> (<parameter list>){
        ...
    }
    ...
}
```

Some of the phrases used in the template need to be explained:

```
public class
```

Class definitions usually begin with the keyword `public`, indicating that the class is accessible to all potential clients. There are some alternatives to `public` that we ignore for now.

```
<name of class>
```

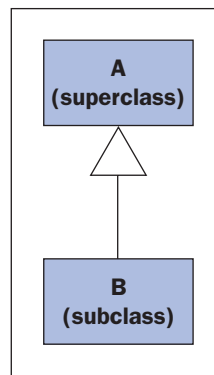
Class names are user-defined symbols, and thus they must adhere to the rules for naming variables and methods. It is common to start class names with a capital letter and variable and method names with a lowercase letter. There is one exception. Names of final variables (those that name constants) are usually completely capitalized.

extends <some other class>

Java organizes its classes in a hierarchy (see Chapter 1). At the root, or base, of this hierarchy is a class called `Object`. In the hierarchy, if class A is immediately above another class B, we say that A is the *superclass* or *parent* of B and B is a *subclass* or *child* of A (Figure 6-4). Each class, except `Object`, has exactly one parent and can have any number of children.

FIGURE 6-4

Relationship between superclass and subclass



When a new class is created, it is incorporated into the hierarchy by extending an existing class. The new class's exact placement in the hierarchy is important because a new class inherits the characteristics of its superclass through a process called *inheritance* (Chapter 1). The new class then adds to and modifies these inherited characteristics, or in other words, the new class *extends* the superclass. If the clause `extends <some other class>` is omitted from the new class's definition, then by default the new class is assumed to be a subclass of `Object`.

private <type> <name>

Instance variables are nearly always declared to be `private`. This prevents clients from referring to the instance variables directly. Making instance variables `private` is an important aspect of information hiding.

public <return type> <name of method>

Methods are usually declared to be `public`, which allows clients to refer to them.

`private` and `public` are *visibility modifiers*. If both `private` and `public` are omitted, the consequences vary with the circumstances. Without explaining why, suffice it to say that in most situations, omitting the visibility modifier is equivalent to using `public`. In most situations, we use `private` for instance variables unless there is some compelling reason to declare them `public`.

To illustrate the difference between `private` and `public`, suppose the class `Student` has a private instance variable `name` and a public method `setName`. Then

```
Student s;
s = new Student();
s.name = "Bill";    // Rejected by compiler because name is private
s.setName ("Bill") // Accepted by compiler because setName is public
```

As a final note concerning our class template, notice that the constructor does not have a return type, or the name of the type of the value that it returns. All other methods do.

Implementation of the `Student` Class

Adhering to the format of our class template, we now implement the `Student` class. It is important to realize that other implementations are acceptable provided they adhere to the interface standards already established for student classes. Following is the code:

```
/* Student.java
Manage a student's name and three test scores.
*/
public class Student {

    // Instance variables
    // Each student object has a name and three test scores
    private String name;           // Student name
    private int test1;             // Score on test 1
    private int test2;             // Score on test 2
    private int test3;             // Score on test 3

    // Constructor method

    public Student(){
        // Initialize a new student's name to the empty string and the test
        // scores to zero.
        name = "";
        test1 = 0;
        test2 = 0;
        test3 = 0;
    }

    // Other methods

    public void setName (String nm){
        // Set a student's name
        name = nm;
    }
    public String getName (){
        // Get a student's name
        return name;
    }

    public void setScore (int i, int score){
        // Set test i to score
```

```

        if      (i == 1) test1 = score;
        else if (i == 2) test2 = score;
        else      test3 = score;
    }

    public int getScore (int i){
    // Retrieve score i
        if      (i == 1) return test1;
        else if (i == 2) return test2;
        else      return test3;
    }

    public int getAverage(){
    // Compute and return the average
        int average;
        average = (int) Math.round((test1 + test2 + test3) / 3.0);
        return average;
    }

    public int getHighScore(){
    // Determine and return the highest score
        int highScore;
        highScore = test1;
        if (test2 > highScore) highScore = test2;
        if (test3 > highScore) highScore = test3;
        return highScore;
    }

    public String toString(){
    // Construct and return a string representation of the student
        String str;
        str = "Name:      " + name  + "\n" +      // "\n" denotes a newline
              "Test 1:   " + test1 + "\n" +
              "Test 2:   " + test2 + "\n" +
              "Test 3:   " + test3 + "\n" +
              "Average: " + getAverage();
        return str;
    }
}

```

We explore the structure and behavior of methods in more detail later in this chapter. For now, the meaning of the code is fairly obvious. All the methods, except the constructor method, have a return type, although the return type may be `void`, indicating that the method in fact returns nothing. To summarize: When an object receives a message, the object activates the corresponding method. The method then manipulates the object's data as represented by the instance variables.

Constructors

The principal purpose of a constructor is to initialize the instance variables of a newly instantiated object. Constructors are activated when the keyword `new` is used and at no other time. A constructor is never used to reset instance variables of an existing object.

A class template can include more than one constructor, provided each has a unique parameter list; however, all the constructors must have the same name—that is, the name of the class. The constructors we have seen so far have had empty parameter lists and are called *default constructors*.

If a class template contains no constructors, the JVM provides a primitive default constructor behind the scenes. This constructor initializes numeric variables to zero and object variables to null, thus indicating that the object variables currently reference no objects. If a class contains even one constructor, however, the JVM no longer provides a default constructor automatically.

To illustrate these ideas, we add several constructors to the `Student` class. The following code lists the original default constructor and two additional ones:

```
// Default constructor -- initialize name to the empty string and
// the test scores to zero.
public Student(){
    name = "";
    test1 = 0;
    test2 = 0;
    test3 = 0;
}

// Additional constructor -- initialize the name and test scores
// to the values provided.
public Student(String nm, int t1, int t2, int t3){
    name = nm;
    test1 = t1;
    test2 = t2;
    test3 = t3;
}

// Additional constructor -- initialize the name and test scores
// to match those in the parameter s.
public Student(Student s){
    name = s.name;
    test1 = s.test1;
    test2 = s.test2;
    test3 = s.test3;
}
```

A class is easier to use when it has a variety of constructors. Following is some code that shows how to use the different `Student` constructors. In a program, we would use the constructor that best suited our immediate purpose:

```
Student s1, s2, s3;
s1 = new Student(); // First student object has
// name "" and scores 0,0,0

s2 = new Student ("Bill",70,80,90); // Second student object has
// name "Bill" and scores 70,80,90

s3 = new Student (s2); // Third student object also has
// name "Bill" and scores 70,80,90
```

```
s3.setName ("Ann");           // Third student object now has
s3.setScore (1,75);           // name "Ann" and scores 75,80,90
```

There are now three completely separate student objects. For a moment, two of them had the same state—that is, the same values for their instance variables—but that changed in the last two lines of code.

Chaining Constructors

When a class includes several constructors, the code for them can be simplified by *chaining* them. For example, the three constructors in the `Student` class each do the same thing—initialize the instance variables. We can simplify the code for the first and third constructors by calling the second constructor. To call one constructor from another constructor, we use the notation

```
this(<parameters>);
```

Thus, the code for the constructors shown earlier becomes

```
// Default constructor -- initialize name to the empty string and
// the test scores to zero.
public Student(){
    this("", 0, 0, 0);
}

// Additional constructor -- initialize the name and test scores
// to the values provided.
public Student(String nm, int t1, int t2, int t3){
    name = nm;
    test1 = t1;
    test2 = t2;
    test3 = t3;
}

// Additional constructor -- initialize the name and test scores
// to match those in the parameter s.
public Student(Student s){
    this(s.name, s.test1, s.test2, s.test3);
}
```

EXERCISE 6.2

1. What are mutators and accessors? Give examples.
2. List two visibility modifiers and describe when they are used.
3. What is a constructor?
4. Why do we include a `toString` method with a new user-defined class?
5. How can two variables refer to the same object? Give an example.

EXERCISE 6.2 Continued

6. Explain the difference between a primitive type and a reference type, and give an example of each.
7. What is the `null` value?
8. What is a null pointer exception? Give an example.
9. How does a default constructor differ from other constructors?
10. How does Java handle the initialization of instance variables if no constructors are provided?
11. What is the purpose of a constructor that expects another object of the same class?

6.3 Editing, Compiling, and Testing the Student Class

To use the `Student` class, we must save it in a file called `Student.java` and compile it by typing

```
javac Student.java
```

in a terminal window. If there are no compile-time errors, the compiler creates the byte code file `Student.class`. Once the `Student` class is compiled, applications can declare and manipulate `Student` objects provided that one of the following is true:

- The code for the application and `Student.class` are in the same directory.
- The `Student.class` is part of a package (see Appendix G).

Following is a small program that uses and tests the `Student` class. Figure 6-5 shows the results of running such a program.

```
// Example 6.1: Test program for Student class

public class TestStudent{

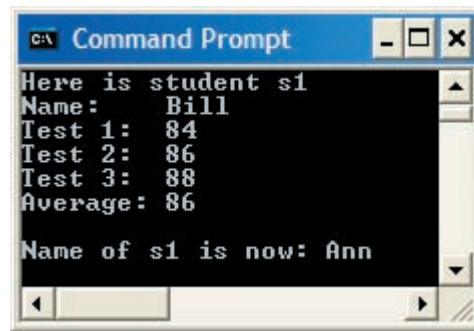
    public static void main (String[] args){
        Student s1, s2;

        s1 = new Student();    // Instantiate a student object
        s1.setName("Bill");    // Set the student's name to "Bill"
        s1.setScore(1,84);     // Set the score on test 1 to 84
        s1.setScore(2,86);     // on test 2 to 86
        s1.setScore(3,88);     // on test 3 to 88
        System.out.println("\nHere is student s1\n" + s1);

        s2 = s1;               // s1 and s2 now refer to the same object
        s2.setName("Ann");     // Set the name through s2
        System.out.println("\nName of s1 is now: " + s1.getName());
    }
}
```

FIGURE 6-5

Output from the TestStudent program



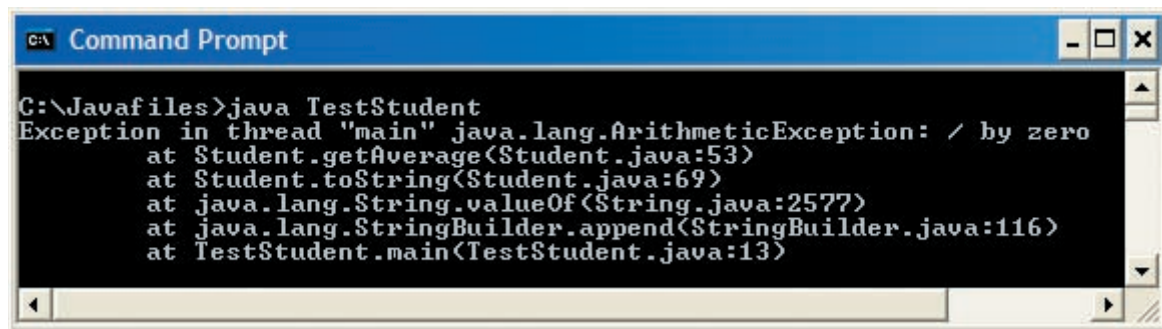
Finding the Location of Run-Time Errors

Finding run-time errors in programs is no more difficult when there are several classes instead of just one. To illustrate, we introduce a run-time error into the `Student` class and then run the `TestStudent` program again. Following is a listing of the modified and erroneous lines of code. Figure 6-6 shows the error messages generated when the program runs.

```
public int getAverage(){
    int average = 0;
    average = (int) Math.round((test1 + test2 + test3) / average);
    return average;
}
```

FIGURE 6-6

Divide by zero run-time error message



The messages indicate that:

- an attempt was made to divide by zero in the `Student` class's `getAverage` method (line 53),
- which was called from the `Student` class's `toString` method (line 69),
- which was called by some methods we did not write,
- which, finally, was called from the `TestStudent` class's `main` method (line 13).

Following are the lines of code mentioned:

```
Student getAverage line 53:
    average = (int) Math.round ((test1 + test2 + test3) / average);
Student toString line 69:
    "Average: " + getAverage();
TestStudent main line 13:
    System.out.println ("\nHere is student s1\n" + s1);
```

We can now unravel the error

- In line 13 of main, the concatenation (+) of s1 makes an implicit call s1.toString().
- In line 69 of toString, the getAverage method is called.
- In line 53 of getAverage, a division by zero occurs.

Case Study 1: Student Test Scores

Request

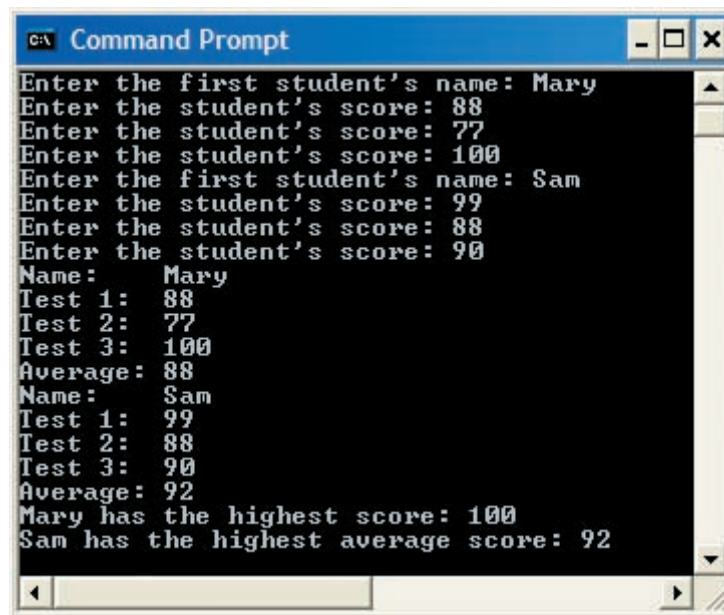
Write a program that allows the user to compare test scores of two students. Each student has three scores.

Analysis

A user's interaction with the program is shown in Figure 6-7.

FIGURE 6-7

The user interface for the student test scores program



As you can see, the program does the following:

1. Prompts the user for the data for the two students
2. Displays the information for each student, followed by the names of the students with the highest score and the highest average score

As a standard part of analysis, we determine which classes are needed to support the application, and we delineate each class's overall responsibilities. The nature of the current problem suggests the use of two classes:

1. `student`: Not surprisingly, the `student` class presented earlier exactly fits the needs of this program.
2. `studentApp`: This class supports the user interface and declares and manipulates two `student` objects.

In general, it is a good idea to divide the code for most interactive applications into at least two sets of classes. One set of classes, which we call the *view*, handles the interactions with the human users, such as input and output operations. The other set of classes, called the *model*, represents the data used by the application. One of the benefits of this separation of responsibilities is that one can write different views for the same data model, such as a terminal-based view and a graphical-based view, without changing a line of code in the data model. Alternatively, one can write different representations of the data model without altering a line of code in the views. In most of the case studies that follow, we apply this framework, called the *model/view pattern*, to structure the code.

Design

During analysis, we decided to base the implementation on two classes: `student` and `studentApp`. Now, during design, we specify the characteristics of these classes in detail. This involves determining the data requirements of each class and the methods that will be needed by the clients of the classes. This process is usually straightforward. To illustrate, let us pretend for the moment that we have not already written the `student` class.

Designing the Student Class

We know from the work completed during analysis that a `student` object must keep track of a name and three test scores. The high score and the average can be calculated when needed. Thus, the data requirements are clear. The `student` class must declare four instance variables:

```
private String name;
private int test1;
private int test2;
private int test3;
```

To determine the `student` class's methods, we look at the class from the perspective of the clients who will be sending messages to `student` objects. In this application, the user interface is the only client. There are some clues that help us pick the appropriate methods:

The user interface needs to instantiate two `student` objects. This indicates the need for a constructor method, which we always include anyway.

When the user enters input data, the view needs to tell each `student` object its name and three test scores. This can be handled by two mutator methods: `setName(theName)` and `setScore(whichTest, testScore)`.

The view needs to ask the `student` objects for their complete information, the highest score, and the average score. This suggests four accessor methods: `toString()`, `getScore(whichTest)`, `getHighScore()`, and `getAverage()`.

We summarize our findings in the following *class summary* box:

```
Class:
    Student
Private Instance Variables:
    String name
    int test1
    int test2
    int test3
Public Methods:
    constructors
    void setName (theName)
    String getName()
    void setScore (whichTest, testScore)
    int getScore (whichTest)
    int getAverage()
    int getHighScore()
    String toString()
```

Normally, we would complete a class's design by writing pseudocode for methods whose implementation is not obvious, so we skip this step here.

Designing the StudentApp Class

The following is the class summary box for the `StudentApp` class:

```
Class:
    StudentApp
Public Methods:
    static void main (args)
```

Implementation

The code for the `student` class has already been presented. Code for the `StudentApp` class follows:

```
import java.util.Scanner;

public class StudentApp{

    public static void main (String[] args){
        // Instantiate the students and the keyboard object
        Student student1 = new Student();
```

```

Student student2 = new Student();
Scanner reader = new Scanner(System.in);

String name;
int score;

// Input the first student's data
System.out.print("Enter the first student's name: ");
name = reader.nextLine();
student1.setName(name);
for (int i = 1; i <= 3; i++){
    System.out.print("Enter the student's score: ");
    score = reader.nextInt();
    student1.setScore(i, score);
}
// Consume the trailing newline character
reader.nextLine();

// Input the second student's data
System.out.print("Enter the second student's name: ");
name = reader.nextLine();
student2.setName(name);
for (int i = 1; i <= 3; i++){
    System.out.print("Enter the student's score: ");
    score = reader.nextInt();
    student2.setScore(i, score);
}

// Output the two students' information
System.out.println(student1);
System.out.println(student2);

// Output the student with the highest score
if (student1.getHighScore() > student2.getHighScore()){
    name = student1.getName();
    score = student1.getHighScore();
}else{
    name = student2.getName();
    score = student2.getHighScore();
}
System.out.println(name + " has the highest score: " + score);

// Output the student with the highest average score
if (student1.getAverage() > student2.getAverage()){
    name = student1.getName();
    score = student1.getAverage();
}else{
    name = student2.getName();
    score = student2.getAverage();
}
System.out.println(name + " has the highest average score: " +
                    score);
}
}

```


6.4 The Structure and Behavior of Methods

As mentioned in earlier chapters, a method is a description of a task that is performed in response to a message. The purpose of this section is to examine more closely some related concepts such as parameters, return types, and local variables.

The Structure of a Method Definition

Methods generally have the following form:

```
<visibility modifier> <return type> <method name> (<parameter list>){
    <implementing code>
}
```

Note the following points:

- The visibility modifier `public` is used when the method should be available to clients of the defining class. The visibility modifier `private` should be used when the method is merely a “helper” used by other methods within the class. We say more about helper methods shortly.
- The return type should be `void` when the method returns no value. A `void` method is often a mutator—that is, a method that modifies an object’s variables. If not `void`, the return type can be any primitive or reference type. Methods that return a value often are accessors, that is, methods that allow clients to examine the values of instance variables.
- Method names have the same syntax as other Java identifiers. The programmer should be careful to use names that describe the tasks that the methods perform, however; the names of verbs or verb phrases, such as `getName`, are usually appropriate for methods.
- As mentioned earlier in this book, parentheses are required whether or not parameters are present. The parameter list, if present, consists of one or more pairs of type names and parameter names, separated by commas.

A method’s implementing code can be omitted. In that case, the method is called a *stub*. Stubs are used to set up skeletal, incomplete, but running programs during program development. For example, here is a class that contains only variable declarations and method stubs:

```
public class SomeClass{

    private int someVariable1, someVariable2;

    public void mutator1(int valueIn){}

    public void mutator2(int valueIn){}

    public int accessor1(){
        return 0;
    }
}
```

Return Statements

If a method has a return type, its implementing code must have at least one `return` statement that returns a value of that type. There can be more than one `return` statement in a method; however, the first one executed ends the method. Following is an example of a method that has two `return` statements but executes just one of them:

```
boolean odd(int i){
    if (i % 2 == 0)
        return false;
    else
        return true;
}
```

A `return` statement in a `void` method quits the method and returns nothing.

Formal and Actual Parameters

Parameters listed in a method's definition are called *formal parameters*. Values passed to a method when it is invoked are called *arguments* or *actual parameters*. As an example, consider the following two code segments:

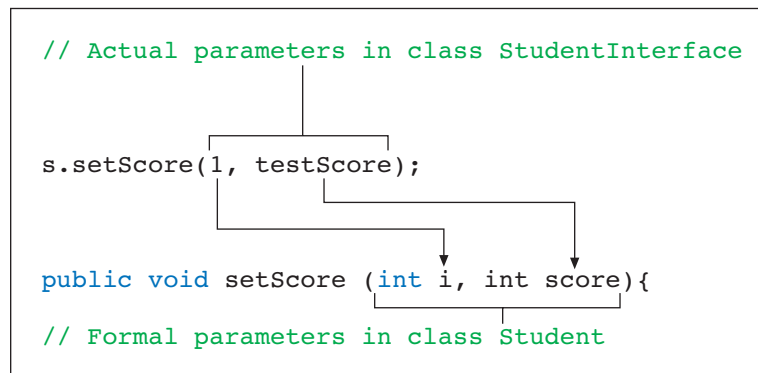
// Client code

```
Student s = new Student();
Scanner reader = new Scanner(System.in);
System.out.print("Enter a test score:");
int testScore = reader.nextInt();
s.setScore(1, testScore); // 1 and testScore are actual parameters
```

// Server code

```
public void setScore (int i, int score){ // i and score are formal parameters
    if      (i == 1) test1 = score;
    else if (i == 2) test2 = score;
    else      test3 = score;
}
```

In our example, the literal `1` and the variable `testScore` are the actual parameters and the names `i` and `score` are the formal parameters. When a method is called, the value of the actual parameter is automatically transferred to the corresponding formal parameter immediately before the method is activated. Thus, the number `1` and value of `testScore` are transferred to `i` and `score` immediately before `setScore` is activated (see Figure 6-8). It is important to understand that the variable `testScore` and the parameter `score` are otherwise completely independent of each other. For instance, changing the value of `score` would have no effect on the value of `testScore`.

FIGURE 6-8
Parameter passing

As mentioned in Chapter 3, when a method has multiple parameters, the caller must provide the right number and types of values. That is, the actual parameters must match the formal parameters in position and type. The rules for matching the types of a formal and an actual parameter are similar to those for assignment statements. The actual parameter's type must be either the same as or less inclusive than the type of the corresponding formal parameter. For example, the method `Math.sqrt`, which has a single formal parameter of type `double`, can receive either a `double` or an `int` as an actual parameter from the caller.

Parameters and Instance Variables

The purpose of a parameter is to pass information to a method. The purpose of an instance variable is to maintain information in an object. These roles are clearly shown in the method `setScore`. This method receives the score in the formal parameter `score`. This value is then transferred to one of the instance variables `test1`, `test2`, or `test3`.

Local Variables

Occasionally, it is convenient to have temporary working storage for data in a method. The programmer can declare *local variables* for this purpose. A good example occurs in the method `getAverage`. This method declares a variable `average`, assigns it the result of computing the average of the integer instance variables, and returns its value:

```

public int getAverage(){
    int average;
    average = (int) Math.round((test1 + test2 + test3) / 3.0);
    return average;
}

```

Note that there is no need for the method to receive data from the client, so we do not use a parameter. Likewise, there is no need for the object to remember the average, so we do not use an instance variable for that.

Helper Methods

Occasionally, a task performed by a method becomes so complex that it helps to break it into subtasks to be solved by several other methods. To accomplish this, a class can define one or

more methods to serve as *helper methods*. These methods are usually private because only methods already defined within the class need to use them. For example, it is helpful to define a debug method when testing a class. This method expects a string and a double as parameters and displays these values in the terminal window. Following is the code:

```
private void debug(String message, double value){
    System.out.println(message + " " + value);
}
```

This method can be called from any other method in the class to display information about the state of an integer or double variable. For example, the `Student` method `getAverage` might use this method as follows:

```
public int getAverage(){
    int average;
    average = (int) Math.round((test1 + test2 + test3) / 3.0);
    debug("Average:", average);
    return average;
}
```

The advantage to this approach is that debugging statements throughout the class can be turned on or off by commenting out a single line of code:

```
private void debug(String message, double value){
    // System.out.println(message + " " + value);
}
```

We see other examples of helper methods in later chapters.

EXERCISE 6.4

1. Explain the difference between formal parameters and actual parameters.
2. How does Java transmit data by means of parameters?
3. Define a method `sum`. This method expects two integers as parameters and returns the sum of the numbers ranging from the first integer to the second one.
4. What is the purpose of local variables?

6.5 Scope and Lifetime of Variables

As we have seen repeatedly, a class definition consists of two principal parts: a list of instance variables and a list of methods. When an object is instantiated, it receives its own complete copy of the instance variables, and when it is sent a message, it activates the corresponding method in its class. Thus, it is the role of objects to contain data and to respond to messages, and it is the role of classes to provide a template for creating objects and to store the code for methods. When a method is executing, it does so on behalf of a particular object, and the method

has complete access to the object's instance variables. From the perspective of the methods, the instance variables form a common pool of variables accessible to all of the class's methods. Instance variables are usually private, which does not restrict access by the class's methods, but does block access from clients. In contrast, variables declared within a method are accessible only within that method and are called *local variables*.

Scope of Variables

The *scope* of a variable is that region of the program within which it can validly appear in lines of code. The scope of a parameter or a local variable is restricted to the body of the method that declares it, whereas the scope of a private instance variable is all the methods in the defining class. Fortunately, the compiler flags as an error any attempt to use variables outside of their scope. Following is an example that illustrates the difference between local and private scope:

```
public class ScopeDemo {

    private int iAmPrivate;

    public void clientMethod (int parm){

        int iAmLocal;
        ...
    }

    private int helperMethod (int parm1, int parm2){

        int iAmLocalToo;
        ...
    }
    ...
}
```

Table 6-3 shows where each of the variables and parameters can be used (that is, its scope):

TABLE 6-3
Variables and their scope

VARIABLE	helperMethod	clientMethod
iAmPrivate	Yes	Yes
parm	No	Yes
iAmLocal	No	Yes
parm1 and parm2	Yes	No
iAmLocalToo	Yes	No

Notice that formal parameters are also local in scope; that is, their visibility is limited to the body of the method in which they are declared.

Block Scope

A method's code can contain nested scopes. Variables declared within any compound statement enclosed in braces are said to have *block scope*. They are visible only within

the code enclosed by braces. For example, consider the following `for` loop to sum 10 input numbers. The accumulator variable `sum` is declared above the loop so the program can access it after the loop terminates. The loop declares its control variable `i` within its header and a local variable `number` to accept input within its body. The variables `i` and `number` thus have block scope, which is appropriate because they are needed only within the loop, and not outside it:

```
int sum = 0;
Scanner reader = new Scanner(System.in);
for (int i = 1; i <= 10; i++){
    System.out.print("Enter a number: ");
    int number = reader.nextInt();
    sum += number;
}
System.out.println("The sum is " + sum);
```

Lifetime of Variables

The *lifetime* of a variable is the period during which it can be used. Local variables and formal parameters exist during a single execution of a method. Each time a method is called, it gets a fresh set of formal parameters and local variables, and once the method stops executing, the formal parameters and local variables are no longer accessible. Instance variables, on the other hand, last for the lifetime of an object. When an object is instantiated, it gets a complete set of fresh instance variables. These variables are available every time a message is sent to the object, and they, in some sense, serve as the object's memory. When the object ceases to exist, the instance variables disappear as well.

Duplicating Variable Names

Because the scope of a formal parameter or local variable is restricted to a single method, the same name can be used within several different methods without causing a conflict. Whether or not we use the same name in several different methods is merely a matter of taste. When the programmer reuses the same local name in different methods, the name refers to a different area of storage in each method. In the next example, the names `iAmLocal` and `parm1` are used in two methods in this way:

```
public class ScopeDemo {

    private int iAmPrivate;

    public void clientMethod (int parm1){

        int iAmLocal;
        ...
    }

    private int helperMethod (int parm1, int parm2){

        int iAmLocal;
        ...
    }
    ...
}
```


A local name and a global variable name can also be the same, as shown in the following code segment:

```
public class ScopeDemo {

    private int iAmAVariable;

    public void someMethod (int parm){

        int iAmAVariable;
        ...
        iAmAVariable = 3;           // Refers to the local variable
        this.iAmAVariable = 4;      // Refers to the instance variable
        ...
    }

    public void someOtherMethod(int iAmAVariable){
        ...
        this.iAmAVariable = iAmAVariable;    // Assign the value of the
                                              // parameter
        ...
                                              // to the instance variable
    }
}
```

In this example, the local variable `iAmAVariable` is said to *shadow* the instance variable with the same name. Shadowing is considered a dangerous programming practice because it greatly increases the likelihood of making a coding error. When the variable name is used in the method, it refers to the local variable, and the instance variable can be referenced only by prefixing `this.` to the name. The programmer uses the symbol `this` to refer to the current instance of a class within that class's definition.

When to Use Instance Variables, Parameters, and Local Variables

The only reason to use an instance variable is to store information within an object. The only reason to use a parameter is to transmit information to a method. The only reason to use a local variable is for temporary working storage within a method. A very common mistake is to misuse one kind of variable for another. Following are the most common examples of these types of mistakes.

MISTAKE 1: INSTANCE VARIABLE USED FOR TEMPORARY WORKING STORAGE

This is perhaps the most common mistake programmers make with variables. As we have seen, an instance variable is in fact not temporary, but survives the execution of the method. No harm may be done. If more than one method (or the same method on different calls) uses the same variable for its temporary storage, however, these methods might share information in ways that cause subtle bugs. For instance, suppose we decide to include an instance variable `sum` to

compute the average score in the `Student` class. We also decide to compute the sum with a loop that uses the method `getScore` as follows:

```
private int sum;
...

public int getAverage(){
    for (int i = 1; i <= 3; i++)
        sum += getScore(i);
    return (int) Math.round(sum / 3.0);
}
```

The method is quite elegant but contains an awful bug. It runs correctly only the first time. The next time the method is called, it adds scores to the sum of the previous call, thus producing a much higher average than expected.

MISTAKE 2: LOCAL VARIABLE USED TO REMEMBER INFORMATION IN AN OBJECT

As we have seen, this intent cannot be realized because a local variable disappears from memory after its method has executed. This mistake can lead to errors in cases in which the programmer uses the same name for a local variable and an instance variable and believes that the reference to the local variable is really a reference to the instance variable (see our earlier discussion of shadowing). All that is required to cause this error is the use of a type name before the first assignment to the variable when it is used in a method. Following is an example from the `Student` class:

```
public void setName (String nm){
    // Set a student's name
    String name = nm;      // Whoops! we have just declared name local.
}
```

In this case, the variable name has been accidentally “localized” by prefixing it with a type name. Thus, the value of the parameter `nm` is transferred to the local variable instead of the instance variable, and the student object does not remember this change.

MISTAKE 3: METHOD ACCESSES DATA BY DIRECTLY REFERENCING AN INSTANCE VARIABLE WHEN IT COULD USE A PARAMETER INSTEAD

Methods can communicate by sharing a common pool of variables or by the more explicit means of parameters and return values. Years of software development experience have convinced computer scientists that the second approach is better even though it seems to require more programming effort. There are three reasons to prefer the use of parameters:

1. Suppose that several methods share a pool of variables and that one method misuses a variable. Then other methods can be affected, and the resulting error can be difficult to find. For example, as the following code segment shows, if method `m1` mistakenly sets the variable `x` to 0 and if method `m2` uses `x` as a divisor, then when the program is run, the computer will signal an error in `m2`, even though the source of the error is in `m1`.

```
// Server class

public class ServerClass{

    private int x;

    public void m1(){

        ...
        x = 0;           // The real source of the error
    }

    public void m2(){
        int y = 10 / x;  // Exact spot of run-time error
    }

    ...
}

// Client class

public class ClientClass{
    private s = new ServerClass();
    public void m3(){
        s.m1();          // Misuse of x occurs, but is hidden from client
        s.m2();          // Run-time error occurs
    }
    ...
}
```

2. It is easier to understand methods and the relationships between them when communications are explicitly defined in terms of parameters and return values.
3. Methods that access a pool of shared variables can be used only in their original context, whereas methods that are passed parameters can be reused in many different situations. Reuse of code boosts productivity, so programmers try to create software components (in this case, methods) that are as reusable as possible. The method `Math.sqrt` is a good example of a context-independent method.

To summarize, it is a good idea to keep the use of instance variables to a minimum, using them only when necessary to track the state of objects, and to use local variables and parameters wherever possible.

EXERCISE 6.5

1. What are the lifetimes of an instance variable, a local variable, and a parameter?
2. What is shadowing? Give an example and describe the problems that shadowing might cause in a program.

EXERCISE 6.5 Continued

3. Consider the following code segment:

```
public class SomeClass{

    private int a, b;

    public void aMutator(int x, y){
        int c, d;
        <lots of code goes here>
    }
}
```

- a. List the instance variables, parameters, and local variables in this code.
- b. Describe the scope of each variable or parameter.
- c. Describe the lifetime of each variable or parameter.

6.6 Graphics and GUIs: Images, a Circle Class, and Mouse Events

Realistic graphics applications display many kinds of images, including those that are created under program control and those that are simply loaded from existing image files. As the program-constructed images grow in number and complexity, it becomes useful to delegate responsibility for managing some of their attributes and behavior to the images themselves. This section discusses how to use image files, how to define classes that represent geometric shapes, and how to make graphics programs respond to a user's mouse manipulations.

Extra Challenge



This Graphics and GUIs section gives you the opportunity to explore concepts and programming techniques required to develop modern graphics applications and graphical user interfaces. This material is not required in order to proceed with the other chapters of the book.

Loading Images from Files and Displaying Them

Images are commonly stored in files in JPEG, GIF, or PNG format. You can obtain such image files from a variety of sources including a digital camera, a flatbed scanner, a screen capture program, or a Web site (but be careful of copyright laws, as discussed in Chapter 1). Once an image file is available, a Java application can load the image into RAM for program use by running the following code, where `fileName` is the name of an image file or its pathname:

```
ImageIcon image = new ImageIcon(fileName);
```

This code creates an `ImageIcon` object with a bitmap for the data in the image. Any images that come from files are typically loaded at program startup and passed to panels when they are instantiated. The panels are then responsible for displaying the images. This differs from the approach we took in Chapter 5, where we wanted to process images at the pixel level. Here we merely want to display an image in a panel.

Our first example program has a main application class that loads an image of a cat named “Smokey” and passes the image to a new `ColorPanel`. The main window is shown in Figure 6-9. Here is the code for the application class:

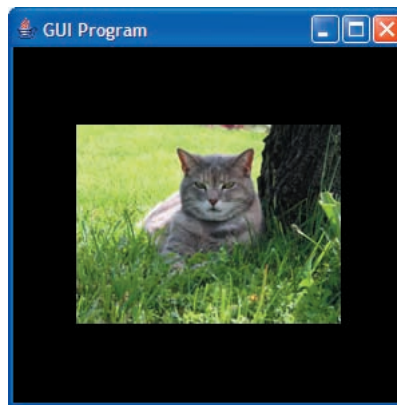
```
// Example 6.2: Loading an image from a file

import javax.swing.*;
import java.awt.*;

public class GUIWindow{

    public static void main(String[] args){
        JFrame theGUI = new JFrame();
        theGUI.setTitle("GUI Program");
        theGUI.setSize(300, 300);
        theGUI.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        ImageIcon image = new ImageIcon("smokey.jpg");
        ColorPanel panel = new ColorPanel(Color.black, image);
        Container pane = theGUI.getContentPane();
        pane.add(panel);
        theGUI.setVisible(true);
    }
}
```

FIGURE 6-9
Displaying an image



The `ColorPanel` receives the image icon at instantiation and saves a reference to it in an instance variable. When the panel paints itself, it paints the image icon. The code for painting the image appears in the panel’s `paintComponent` method and has the following form:

```
anImageIcon.paintIcon(this, g, x, y)
```

where `this` refers to the panel itself, `g` is the panel’s graphics context, and `x` and `y` are the panel coordinates of the image’s upper-left corner.

The methods `getIconWidth()` and `getIconHeight()` return the width and height, respectively, of an image in pixels. These methods can be used with the `getWidth()` and `getHeight()`

methods of the panel class to compute the position of the image relative to the center of the panel. Here is the code for a `ColorPanel` class that displays a centered image:

```
// Displays an image centered in the panel

import javax.swing.*;
import java.awt.*;

public class ColorPanel extends JPanel{

    private ImageIcon image;

    public ColorPanel(Color backColor, ImageIcon i){
        setBackground(backColor);
        image = i;
    }

    public void paintComponent(Graphics g){
        super.paintComponent(g);
        int x = (getWidth() - image.getIconWidth()) / 2;
        int y = (getHeight() - image.getIconHeight()) / 2;
        image.paintIcon(this, g, x, y);
    }
}
```

This `ColorPanel` is different from panels discussed in earlier chapters in that it maintains another object, an image, as part of its state. Another difference is that this panel displays the image by asking an object to paint itself. We see this pattern repeated in the examples that follow.

Geometric Shapes

Graphics examples in earlier chapters displayed geometric shapes such as rectangles and ovals by running the corresponding drawing or filling methods of the `Graphics` class. There are several reasons why it will now be useful to implement each shape as a distinct object with its own methods:

1. A shape has its own attributes, such as a color, a position, and a size. Defining a class for a shape allows a user to manipulate it by changing its color, position, or size.
2. Defining shape classes allows us to program with more specific shapes than those implied by the drawing methods of the `Graphics` class. For example, a circle is a specific type of oval, but we usually think of a circle in terms of its center point and radius rather than a corner point, width, and height. Similar remarks also hold true for triangles and other more complex shapes.
3. If a shape already knows about its own attributes, it just needs a graphics context in order to display itself. Users can then more easily display shapes by asking them to display themselves, using a single `draw(aGraphicsObject)` or `fill(aGraphicsObject)` method.
4. Programs that use multiple shapes in complex arrangements can more easily compose these and manipulate them by using instances of shape classes.

Defining a Circle Class

Circles have a color, a center point, and a radius. In addition to modifying any of these attributes, a user can ask a circle to draw itself (just the circumference) or fill itself (like a solid disk) in its current color. Finally, a user can determine whether or not a circle contains a given point (x,y). The methods of our Circle class are listed in Table 6-4.

TABLE 6-4
Methods in class Circle

METHOD	WHAT IT DOES
Circle(int x, int y, int r, Color c)	Constructor; creates a circle with center point (x, y), radius r, and color c
int getX()	Returns the x coordinate of the center
int getY()	Returns the y coordinate of the center
int getRadius()	Returns the radius
Color getColor()	Returns the color
void setX(int x)	Modifies the x coordinate of the center
void setY(int y)	Modifies the y coordinate of the center
void setRadius(int r)	Modifies the radius
Color setColor(Color c)	Modifies the color
void draw(Graphics g)	Draws an outline of the circle in the graphics context
void fill(Graphics g)	Draws a filled circle in the graphics context
boolean containsPoint(int x, int y)	Returns true if the point (x, y) lies in the circle or false otherwise
void move(int xAmount, int yAmount)	Moves the circle by xAmount horizontally to the right and yAmount vertically downward; Negative amounts move to the left and up.

Implementation of the Circle Class

For the most part, the implementation of the Circle class is trivial. The constructor receives the coordinates of the center point, the radius, and the color from the user and assigns these values to instance variables. We focus on just two methods: draw and containsPoint. The draw method uses drawOval to draw the circle. The drawOval method expects the position and extent of the circle's bounding rectangle, which can be derived from the circle's center and radius, as shown in the following code:

```
public void draw (Graphics g){
    // Save the current color of the graphics context
    // and set color to the circle's color.
    Color oldColor = g.getColor();
    g.setColor(color);
```

```

// Translate the circle's position and radius
// to the bounding rectangle's top left corner, width, and height.
g.drawOval(centerX - radius, centerY - radius, radius * 2, radius * 2);

// Restore the color of the graphics context.
g.setColor(oldColor);
}

```

To determine if a point is in a circle, we consider the familiar equation for all points on the circumference of a circle:

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \quad (\text{Eq. 1})$$

or

$$(x - x_c)^2 + (y - y_c)^2 - r^2 = 0 \quad (\text{Eq. 2})$$

where (x_c, y_c) is the circle's center and r is its radius. A point (x, y) is then in the circle if the left side of Equation 2 is less than or equal to 0. For example, given a circle of radius 2 and center $(0, 0)$, the point $(1, 1)$ produces the result

$$1^2 + 1^2 - 2^2 = -2$$

implying that the point is in the circle.

The following method results from this design:

```

public boolean containsPoint (int x, int y){
    int xSquared = (x - centerX) * (x - centerX);
    int ySquared = (y - centerY) * (y - centerY);
    int radiusSquared = radius * radius;
    return xSquared + ySquared - radiusSquared <= 0;
}

```

Here is a partial listing of the Circle class:

```

// Circle.java: Represents a circle

import java.awt.*;

public class Circle{

    private int centerX, centerY, radius;
    private Color color;

    public Circle(int x, int y, int r, Color c){
        centerX = x;
        centerY = y;
        radius = r;
        color = c;
    }
}

```

```

public void draw(Graphics g){
    Color oldColor = g.getColor();
    g.setColor(color);
    // Translates circle's center to rectangle's origin for drawing.
    g.drawOval(centerX - radius, centerY - radius,
               radius * 2, radius * 2);
    g.setColor(oldColor);
}

public void fill(Graphics g){
    Color oldColor = g.getColor();
    g.setColor(color);
    // Translates circle's center to rectangle's origin for drawing.
    g.fillOval(centerX - radius, centerY - radius,
               radius * 2, radius * 2);
    g.setColor(oldColor);
}

public boolean containsPoint(int x, int y){
    int xSquared = (x - centerX) * (x - centerX);
    int ySquared = (y - centerY) * (y - centerY);
    int radiusSquared = radius * radius;
    return xSquared + ySquared - radiusSquared <= 0;
}

public void move(int xAmount, int yAmount){
    centerX = centerX + xAmount;
    centerY = centerY + yAmount;
}
}

```

Using the Circle Class

Our next example program creates and displays two `Circle` objects (see Figure 6-10). The main window class just creates a `ColorPanel` with a background color and adds it to the window. The `ColorPanel`'s constructor sets two instance variables to different `Circle` objects. When the window is refreshed, the `ColorPanel`'s `paintComponent` method draws one circle and fills the other. Here is the code for `ColorPanel`:

```

// Example 6.3: Displays a circle and a filled circle

import javax.swing.*;
import java.awt.*;

public class ColorPanel extends JPanel{

    private Circle c1, c2;

    public ColorPanel(Color backColor){
        setBackground(backColor);
        c1 = new Circle(200, 100, 25, Color.red);
        c2 = new Circle(100, 100, 50, Color.blue);
    }
}

```

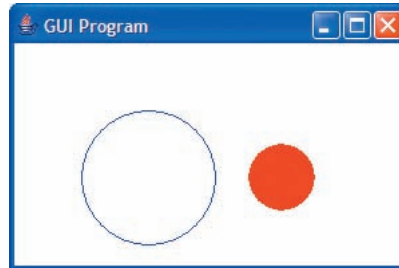
```

public void paintComponent(Graphics g){
    super.paintComponent(g);
    c1.fill(g);
    c2.draw(g);
}
}

```

FIGURE 6-10

Displaying two Circle objects



The Method repaint

Suppose we want to move a shape or image to a new position in a panel in response to a mouse click. Let's assume we have the coordinates of the new position (we see how to get these in the next subsection). There are then just two steps remaining:

1. Set the shape's position to the new position.
2. Refresh the panel, which has the effect of clearing the panel and redrawing the shape at its new position.

The method `repaint` is used to force a refresh of any GUI component, such as a panel. The method essentially invokes that object's `paintComponent` method. Thus, there are two kinds of situations in which `paintComponent` is called:

1. Automatically, by the JVM, at startup; or whenever the main window is altered
2. Under program control, by a call of `repaint`, when the program has made a change to the objects to be displayed

The code that calls `repaint` typically appears in a panel method that has modified one or more shapes in the panel. This method might have the following form:

```

public void modifySomeShapes(someParameters){
    // modify the attributes of one or more shapes
    repaint();
}

```

Responses to Mouse Events

Until now, we have limited our use of the mouse to clicking command buttons and editing data in I/O dialog boxes. Everyone who has used a drawing program knows that much more can be done with the mouse. Drawing applications usually detect and respond to the following mouse events: button presses and releases, mouse movement, and dragging the mouse (that is, moving the mouse while a button is depressed). In addition, a program can respond to the mouse's entry into and exit from a given region.

A program can detect and respond to mouse events by attaching *listener objects* to a panel. When a particular type of mouse event occurs in a panel, its listeners are informed. If a listener has a method whose parameter matches that type of event, the JVM automatically runs this method and passes the event object to it as a parameter. The event object contains the mouse's current panel coordinates. The code in the method carries out the program's response to that mouse event.

For example, let's say that a panel should display the coordinates of the mouse whenever its button is pressed in a panel. The panel's constructor sets some instance variables to default coordinates, say, (100,100), and attaches a new listener object to the panel. The class for this listener object, also defined within the panel class, implements the method `mousePressed`, which will be triggered whenever a mouse-pressed event occurs in that panel. This method receives an event object as a parameter from the JVM. The method resets the coordinates to the ones contained in the event object and calls `repaint` to refresh the panel. The panel's `paintComponent` method simply draws the coordinates.

Our next example program shows a `ColorPanel` class that responds to mouse presses by displaying the mouse's coordinates. The class includes a nested listener class, `PanelListener`, which we explain following the listing.

```
// Example 6.4: Tracks mouse presses by displaying
// the current mouse position

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;    //For the mouse events

public class ColorPanel extends JPanel{

    int x, y;    // Used to track mouse coordinates

    public ColorPanel(Color backColor){
        setBackground(backColor);
        // Establish the default coordinates
        x = 100;
        y = 100;
        // Instantiate and attach the panel's listener
        addMouseListener(new PanelListener());
    }

    public void paintComponent(Graphics g){
        super.paintComponent(g);
        // Draw the current coordinates
        g.drawString("(" + x + ", " + y + ")", x, y);
    }

    private class PanelListener extends MouseAdapter{

        public void mousePressed(MouseEvent e){
            // Obtain the current mouse coordinates and refresh
```

```

        x = e.getX();
        y = e.getY();
        repaint();
    }
}

```

The `ColorPanel` class imports the package `java.awt.event`, which includes the classes `MouseEvent` and `MouseAdapter`. The panel's constructor uses the method `addMouseListener` to attach a `PanelListener`. This class is defined internally within the `ColorPanel` class, so its methods will have access to all of the panel's instance variables and methods.

The `PanelListener` class extends the class `MouseAdapter`. `MouseAdapter` includes several methods that respond to mouse events by doing nothing (very simple default behavior). One of these methods is `mousePressed`. `PanelListener` overrides the default definition of `mousePressed` by redefining it. The method extracts the coordinates of the mouse from the `MouseEvent` parameter using the methods `getX()` and `getY()`. It resets the panel's instance variables `x` and `y` to these values and then repaints the panel. Note that the visibility modifier of the `PanelListener` class is `private` because the enclosing panel class is its only user.

When you want the panel to respond to other types of mouse events, you just include code for the corresponding method in your listener class. At runtime, the JVM takes care of feeding the event object to the appropriate method.

The `MouseAdapter` class includes methods for all but two mouse events, mouse motion and mouse dragging. The methods for these two events are included in the `MouseMotionAdapter` class. Thus, you will have to define a separate class that extends `MouseMotionAdapter` and include either or both of its methods if you want your program to respond to mouse motion events. The method `addMouseMotionListener` is used to add listeners of this type to a panel. Table 6-5 lists the methods and the corresponding mouse events in the `MouseAdapter` and `MouseMotionAdapter` classes.

TABLE 6-5
Methods for responding to mouse events

CLASS	METHOD	TYPE OF EVENT
<code>MouseAdapter</code>	<code>public void mouseEntered(MouseEvent e)</code> <code>public void mouseExited(MouseEvent e)</code> <code>public void mousePressed(MouseEvent e)</code> <code>public void mouseReleased(MouseEvent e)</code> <code>public void mouseClicked(MouseEvent e)</code>	Enter Exit Press Release Click
<code>MouseMotionAdapter</code>	<code>public void mouseMoved(MouseEvent e)</code> <code>public void mouseDragged(MouseEvent e)</code>	Move Drag

Dragging Circles

Our final program example puts together our ideas about shape objects and mouse events to create a simple program for dragging circles around in a window. The user selects a circle by pressing the mouse within it and then moves the circle by dragging it to a desired position.

The `ColorPanel` class has the same two instance variables for circles as in the previous version (see Example 6.2). To these we add instance variables for saving the mouse coordinates and for saving a reference to the selected circle. This last variable is `null` at startup and after a circle has been deselected.

Here are the types of mouse events and the associated responses of the program:

1. *Mouse press.* Save the current coordinates of the mouse. If one of the shapes contains those coordinates, save a reference to that shape (thereby selecting it).
2. *Mouse release.* Deselect the selected shape, if there is one, by setting the saved reference to `null`.
3. *Mouse drag.* Compute the x and y distances by using the current mouse coordinates and the saved mouse coordinates. If a shape is currently selected, move it using the distances and repaint. Finally, set the saved coordinates to the current mouse coordinates.

Here is a complete listing of the `ColorPanel` class for dragging shapes:

```
// Example 6.5: Displays a circle and a filled circle
// Allows the user to drag a circle to another position

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;           //For the mouse events

public class ColorPanel extends JPanel{

    private Circle c1, c2;
    private Circle selectedCircle; // Used to track selected shape
    private int x, y;             // Used to track mouse coordinates

    public ColorPanel(Color backColor){
        setBackground(backColor);
        c1 = new Circle(200, 100, 25, Color.red);
        c2 = new Circle(100, 100, 50, Color.blue);
        selectedCircle = null;
        addMouseListener(new PanelListener());
        addMouseMotionListener(new PanelMotionListener());
    }

    public void paintComponent(Graphics g){
        super.paintComponent(g);
        c1.fill(g);
        c2.draw(g);
    }

    private class PanelListener extends MouseAdapter{
```



```

public void mousePressed(MouseEvent e){
    // Select a circle if it contains the mouse coordinates
    x = e.getX();
    y = e.getY();
    if (c1.containsPoint(x, y))
        selectedCircle = c1;
    else if (c2.containsPoint(x, y))
        selectedCircle = c2;
}

public void mouseReleased(MouseEvent e){
    // Deselect the selected circle
    selectedCircle = null;
}
}

private class PanelMotionListener extends MouseMotionAdapter{

    public void mouseDragged(MouseEvent e){
        if (selectedCircle != null){
            // Compute the distance and move the selected circle
            int newX = e.getX();
            int newY = e.getY();
            int dx = newX - x;
            int dy = newY - y;
            selectedCircle.move(dx, dy);
            x = newX;
            y = newY;
            repaint();
        }
    }
}

```

In later chapters, we explore how to write programs that manipulate an arbitrary number of graphical objects.

EXERCISE 6.6

1. The program example that displayed an image in this section has a problem. If the image size is larger than that of the window, the user must resize the window to view the entire image. Describe how to fix the program so that the window size exactly fits the image size at startup.
2. Geometric shapes can be scaled up or down in size. For example, scaling by a factor of 3 triples the size of a shape, whereas scaling by a factor of 0.5 reduces the size of a shape by one-half. Write a method `scale` that expects a parameter of type `double` and scales a circle by that factor.
3. Explain how a program can detect and respond to the user's pressing of a mouse button.

SUMMARY

In this chapter, you learned:

- Java class definitions consist of instance variables, constructors, and methods.
- Constructors initialize an object's instance variables when the object is created. A default constructor expects no parameters and sets the variables to reasonable default values. Other constructors expect parameters that allow clients to set up objects with specified data.
- Mutator methods modify an object's instance variables, whereas accessor methods merely allow clients to observe the values of these variables.
- The visibility modifier `public` is used to make methods visible to clients, whereas the visibility modifier `private` is used to encapsulate or restrict access to variables and methods.
- Helper methods are methods that are called from other methods in a class definition. They are usually declared to be `private`.
- Variables within a class definition can be instance variables, local variables, or parameters. Instance variables are used to track the state of an object. Local variables are used for temporary working storage within a method. Parameters are used to transmit data to a method.
- A formal parameter appears in a method's signature and is referenced in its code. An actual parameter is a value passed to a method when it is called. A method's actual parameters must match its formal parameters in number, position, and type.
- The scope of a variable is the area of program text within which it is visible. The scope of an instance variable is the entire class within which it is declared. The scope of a local variable or a parameter is the body of the method within which it is declared.
- The lifetime of a variable is the period of program execution during which its storage can be accessed. The lifetime of an instance variable is the same as the lifetime of a particular object. The lifetime of a local variable and a parameter is the time during which a particular call of a method is active.

VOCABULARY *Review*

Define the following terms:

Accessor	Formal parameter	Mutator
Actual parameter	Helper method	Scope
Behavior	Identity	State
Constructor	Instantiation	Visibility modifier
Encapsulation	Lifetime	

REVIEW Questions

WRITTEN QUESTIONS

Write a brief answer to the following questions.

1. Explain the difference between a class and an instance of a class.
2. Explain the difference between the visibility modifiers `public` and `private`.
3. What are accessor and mutator methods?
4. Develop a design for a new class called `BaseballPlayer`. The variables of this class are
 - name (a `String`)
 - team (a `String`)
 - home runs (an `int`)
 - batting average (a `double`)

Express your design in terms of a class summary box. The class should have a constructor and methods for accessing and modifying all of the variables.

5. Explain how a parameter transmits data to a method.

6. What are local variables and how should they be used in a program?

PROJECTS

PROJECT 6-1

Add the extra constructors to the `Student` class of this chapter's first case study (Student Test Scores), and test these methods thoroughly with a `Tester` program.

PROJECT 6-2

A `Student` object should validate its own data. The client runs this method, called `validateData()`, with a `Student` object, as follows:

```
String result = student.validateData();
if (result == null)
    <use the student>
else
    System.out.println(result);
```

If the student's data are valid, the method returns the value `null`; otherwise, the method returns a string representing an error message that describes the error in the data. The client can then examine this result and take the appropriate action.

A student's name is invalid if it is an empty string. A student's test score is invalid if it lies outside the range from 0 to 100. Thus, sample error messages might be

```
"SORRY: name required"
```

and

```
"SORRY: must have 0 <= test score <= 100".
```

Implement and test this method.

PROJECT 6-3

Develop a new class for representing fractions. The numerator and denominator of a fraction are integers. The constructor expects these values as parameters. Define accessor methods to obtain the numerator and the denominator. Use the rules of fraction arithmetic to define methods to add, subtract, multiply, and divide fractions. Each of these methods expects a fraction object as a parameter. This object is considered to be the right operand of the operation. The left operand is the receiver object, that is, the one containing the instance variables for the numerator and denominator. Each arithmetic method builds a new instance of the fraction class with the results of its calculation and returns it as the method's value. Finally, include a `toString()`

method that returns a string of the form <numerator>/<denominator>. Write a tester program that exercises all of the methods. Here are the rules for fraction arithmetic:

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1d_2 + n_2d_1}{d_1d_2}$$

$$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1d_2 - n_2d_1}{d_1d_2}$$

$$\frac{n_1}{d_1} \times \frac{n_2}{d_2} = \frac{n_1n_2}{d_1d_2}$$

$$\frac{n_1/d_1}{n_2/d_2} = \frac{n_1d_2}{d_1n_2}$$

PROJECT 6-4

Redo the Lucky Sevens dice-playing program from the Case Study in Chapter 4 so that it uses dice objects. That is, design and implement a `Dice` class. Each instance of this class should contain the die's current side. There should be an accessor method for a die's current value. The method `roll` is the only mutator method. Be sure to test the `Dice` class in a simple tester program before incorporating it into the application.

PROJECT 6-5

Develop a new class called `BankAccount`. A bank account has an owner's name and a balance. Be sure to include a constructor that allows a client to supply the owner's name and an initial balance. A bank account needs accessors for the name and balance, mutators for making deposits and withdrawals, and a `toString` method. Test-drive your new class with a program similar to the one used to test the `Student` class in Section 6.3.

PROJECT 6-6

Patrons of a library can borrow up to three books. A patron, therefore, has a name and up to three books. A book has an author and a title. Design and implement two classes, `Patron` and `Book`, to represent these objects and the following behavior:

- The client can instantiate a book with a title and author.
- The client can examine but not modify a book's title or author.
- The client can ask a patron whether it has borrowed a given book (identified by title).
- The client can tell a patron to return a given book (identified by title).
- The client can tell a patron to borrow a given book.

The `Patron` class should use a separate instance variable for each book (a total of three). Each of these variables is initially `null`. When a book is borrowed, the patron looks for a book variable that is not `null`. If no such variable is found, the method returns `false`. If a `null` variable is found, it is reset to the new book and the method returns `true`. Similar considerations apply to the other methods. Use the method `aString.equals(aString)` to compare two strings for equality. Be sure to include appropriate `toString` methods in your `Book` and `Patron` classes and test the `Patron` and `Book` classes thoroughly using an appropriate tester program.

PROJECT 6-7

Write a program that allows the user to display 1, 2, or 4 images in a grid of panels. At program startup, the user is prompted for the number of images. If the input number is not 1, 2, or 4, the program quits with an error message. Otherwise, the program prompts the user for the name of each image file, loads the image, installs it in a `ColorPanel`, and adds the panel to a grid.

PROJECT 6-8

Define a `Rectangle` class to represent rectangles. Modify the program of Section 6.6 so that it uses two rectangles instead of two circles.

PROJECT 6-9

Write a program that displays an 8-by-8 grid of panels, all of which are initially colored white. When the user presses the mouse within a panel, its color should change to a randomly generated color.

CRITICAL *Thinking*

Explain how you could modify the fraction class created in Project 6-3 to display a fraction in a form that is reduced to lowest terms.