SESSION 5

Programming Languages for Objects

Programming in the Large II: Objects and Classes

WHEREAS A SUBROUTINE represents a single task, an object can encapsulate both data (in the form of instance variables) and a number of different tasks or "behaviors" related to that data (in the form of instance methods). Therefore objects provide another, more sophisticated type of structure that can be used to help manage the complexity of large programs.

The first four sections of this chapter introduce the basic things you need to know to work with objects and to define simple classes. The remaining sections cover more advanced topics; you might not understand them fully the first time through. In particular, Section 5.5 covers the most central ideas of object-oriented programming: inheritance and polymorphism. However, in this textbook, we will generally use these ideas in a limited form, by creating independent classes and building on existing classes rather than by designing entire hierarchies of classes from scratch.

Objects, Instance Methods, and Instance Variables

OBJECT-ORIENTED PROGRAMMING (OOP) represents an attempt to make programs more closely model the way people think about and deal with the world. In the older styles of programming, a programmer who is faced with some problem must identify a computing task that needs to be performed in order to solve the problem. Programming then consists of finding a sequence of instructions that will accomplish that task. But at the heart of object-oriented programming, instead of tasks we find objects -- entities that have behaviors, that hold information, and that can interact with one another. Programming consists of designing a set of objects that somehow model the problem at hand. Software objects in the program can represent

real or abstract entities in the problem domain. This is supposed to make the design of the program more natural and hence easier to get right and easier to understand.

To some extent, OOP is just a change in point of view. We can think of an object in standard programming terms as nothing more than a set of variables together with some subroutines for manipulating those variables. In fact, it is possible to use object-oriented techniques in any programming language. However, there is a big difference between a language that makes OOP possible and one that actively supports it. An object-oriented programming language such as Java includes a number of features that make it very different from a standard language. In order to make effective use of those features, you have to "orient" your thinking correctly.

As I have mentioned before, in the context of object-oriented programming, subroutines are often referred to as methods. Now that we are starting to use objects, I will be using the term "method" more often than "subroutine."

5.1.1 Objects, Classes, and Instances

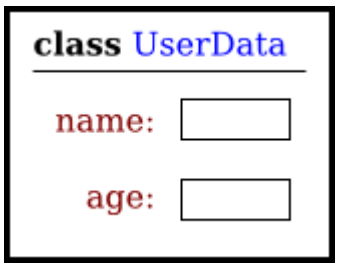
Objects are closely related to classes. We have already been working with classes for several chapters, and we have seen that a class can contain variables and methods (that is, subroutines). If an object is also a collection of variables and methods, how do they differ from classes? And why does it require a different type of thinking to understand and use them effectively? In the one section where we worked with objects rather than classes, Section 3.9, it didn't seem to make much difference: We just left the word "`static`" out of the subroutine definitions!

I have said that classes "describe" objects, or more exactly that the non-static portions of classes describe objects. But it's probably not very clear what this means. The more usual terminology is to say that objects belong to classes, but this might not be much clearer. (There is a real shortage of English words to properly distinguish all the concepts involved. An object certainly doesn't "belong" to a class in the same way that a member variable "belongs" to a class.) From the point of view of programming, it is more exact to say that classes are used to create objects. A class is a kind of factory -- or blueprint -- for constructing objects. The non-static parts of the class specify, or describe, what variables and methods the objects will contain. This is part of the explanation of how objects differ from classes: Objects are created and destroyed as the program runs, and there can be many objects with the same structure, if they are created using the same class.

Consider a simple class whose job is to group together a few static member variables. For example, the following class could be used to store information about the person who is using the program:

```
class UserData {
    static String name;
    static int age;
}
```

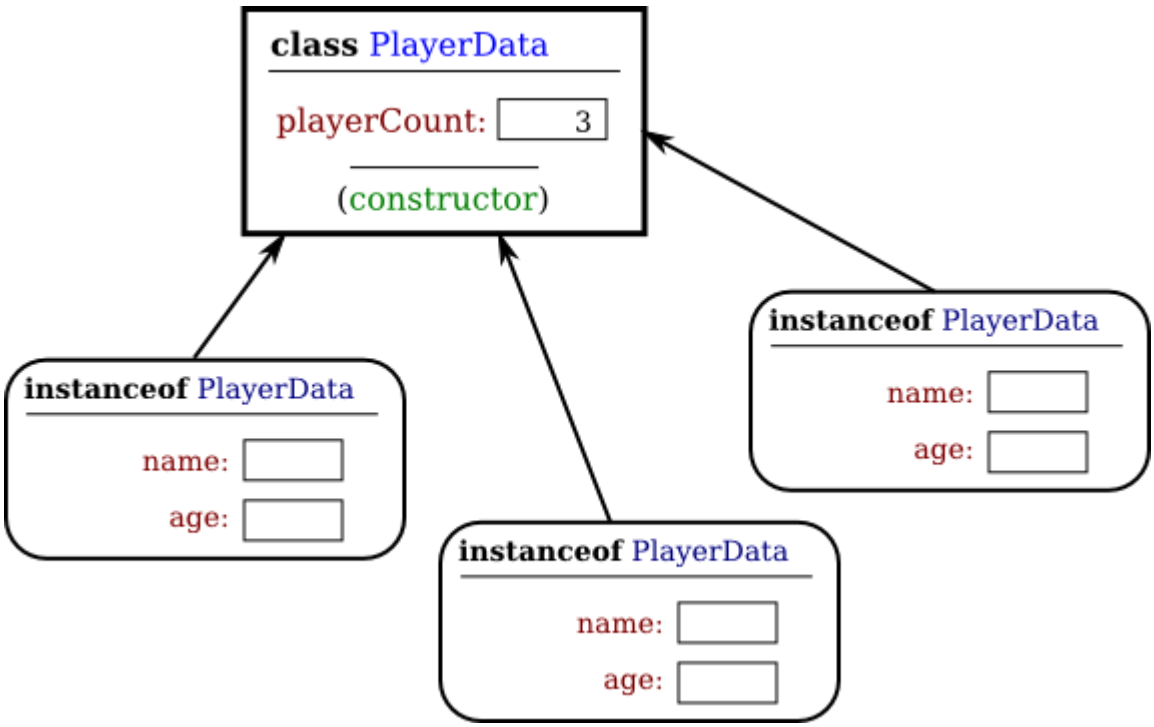
In a program that uses this class, there is only one copy of each of the variables `UserData.name` and `UserData.age`. When the class is loaded into the computer, there is a section of memory devoted to the class, and that section of memory includes space for the values of the variables `name` and `age`. We can picture the class in memory as looking like this:



An important point is that the static member variables are part of the representation of the class in memory. Their full names, `UserData.name` and `UserData.age`, use the name of the class, since they are part of the class. When we use class *UserData* to represent the user of the program, there can only be **one** user, since we only have memory space to store data about one user. Note that the class, `UserData`, and the variables it contains exist as long as the program runs. (That is essentially what it means to be "static.") Now, consider a similar class that includes some non-static variables:

```
class PlayerData {
   static int playerCount;
   String name;
   int age;
}
```

I've also included a static variable in the *PlayerData* class. Here, the static variable `playerCount` is stored as part of the representation of the class in memory. It's full name is `PlayerData.playerCount`, and there is only one of it, which exists as long as the program runs. However, the other two variables in the class definition are non-static. There is no such variable as `PlayerData.name` or `PlayerData.age`, since non-static variables do not become part of the class itself. But the *PlayerData* class can be used to create objects. There can be many objects created using the class, and each one will have its **own** variables called `name` and `age`. This is what it means for the non-static parts of the class to be a template for objects: Every object gets its own copy of the non-static part of the class. We can visualize the situation in the computer's memory after several object have been created like this:



Note that the static variable `playerCount` is part of the class, and there is only one copy. On the other hand, every object contains a `name` and an `age`. An object that is created from a class is called an instance of that class, and as the picture shows every object "knows" which class was used to create it. I've shown class *PlayerData* as containing something called a "constructor;" the constructor is a subroutine that creates objects.

Now there can be many "players" because we can make new objects to represent new players on demand. A program might use the *PlayerData* class to store information about multiple players in a game. Each player has a name and an age. When a player joins the game, a new `PlayerData` object can be created to represent that player. If a player leaves the game, the `PlayerData` object that represents that player can be destroyed. A system of objects in the program is being used to dynamically model what is happening in the game. You can't do this with static variables! "Dynamic" is the opposite of "static."

An object that is created using a class is said to be an instance of that class. We will sometimes say that the object belongs to the class. The variables that the object contains are called instance variables. The methods (that is, subroutines) that the object contains are called instance methods. For example, if the `PlayerData` class, as defined above, is used to create an object, then that object is an instance of the `PlayerData` class, and `name` and `age` are instance variables in the object.

My examples here don't include any methods, but methods work similarly to variables. Static methods are part of the class; non-static, or instance, methods become part of objects created

from the class. It's not literally true that each object contains its own copy of the actual compiled code for an instance method. But logically an instance method is part of the object, and I will continue to say that the object "contains" the instance method.

Note that you should distinguish between the **source code** for the class, and the **class itself** (in memory). The source code determines both the class and the objects that are created from that class. The "static" definitions in the source code specify the things that are part of the class itself (in the computer's memory), whereas the non-static definitions in the source code specify things that will become part of every instance object that is created from the class. By the way, static member variables and static member subroutines in a class are sometimes called class variables and class methods, since they belong to the class itself, rather than to instances of that class.

As you can see, the static and the non-static portions of a class are very different things and serve very different purposes. Many classes contain only static members, or only non-static, and we will see only a few examples of classes that contain a mixture of the two.

5.1.2 Fundamentals of Objects

So far, I've been talking mostly in generalities, and I haven't given you much of an idea about what you have to put in a program if you want to work with objects. Let's look at a specific example to see how it works. Consider this extremely simplified version of a `Student` class, which could be used to store information about students taking a course:

```
public class Student {

   public String name;  // Student's name.
   public double test1, test2, test3;   // Grades on three tests.

   public double getAverage() {  // compute average test grade
      return (test1 + test2 + test3) / 3;
   }

}  // end of class Student
```

None of the members of this class are declared to be `static`, so the class exists only for creating objects. This class definition says that any object that is an instance of the `Student` class will include instance variables named `name`, `test1`, `test2`, and `test3`, and it will include an instance method named `getAverage()`. The names and tests in different objects will generally have different values. When called for a particular student, the method `getAverage()` will compute an average using **that student's** test grades. Different students can have different averages. (Again, this is what it means to say that an instance method belongs to an individual object, not to the class.)

In Java, a class is a **type**, similar to the built-in types such as int and boolean. So, a class name can be used to specify the type of a variable in a declaration statement, or the type of a formal

parameter, or the return type of a function. For example, a program could define a variable named `std` of type `Student` with the statement

```
Student std;
```

However, declaring a variable does **not** create an object! This is an important point, which is related to this Very Important Fact:

In Java, no variable can ever hold an object.
A variable can only hold a reference to an object.

You should think of objects as floating around independently in the computer's memory. In fact, there is a special portion of memory called the heap where objects live. Instead of holding an object itself, a variable holds the information necessary to find the object in memory. This information is called a reference or pointer to the object. In effect, a reference to an object is the address of the memory location where the object is stored. When you use a variable of object type, the computer uses the reference in the variable to find the actual object.

In a program, objects are created using an operator called `new`, which creates an object and returns a reference to that object. (In fact, the `new` operator calls a special subroutine called a "constructor" in the class.) For example, assuming that `std` is a variable of type `Student`, declared as above, the assignment statement

```
std = new Student();
```

would create a new object which is an instance of the class `Student`, and it would store a reference to that object in the variable `std`. The value of the variable is a reference, or pointer, to the object. The object itself is somewhere in the heap. It is not quite true, then, to say that the object is the "value of the variable `std`" (though sometimes it is hard to avoid using this terminology). It is certainly **not at all true** to say that the object is "stored in the variable `std`." The proper terminology is that "the variable `std` refers to or points to the object," and I will try to stick to that terminology as much as possible. If I ever say something like "std **is** an object," you should read it as meaning "std is a variable that refers to an object."

So, suppose that the variable `std` refers to an object that is an instance of class `Student`. That object contains instance variables `name`, `test1`, `test2`, and `test3`. These instance variables can be referred to as `std.name`, `std.test1`, `std.test2`, and `std.test3`. This follows the usual naming convention that when `B` is part of `A`, then the full name of `B` is `A.B`. For example, a program might include the lines

```
System.out.println("Hello, "  +  std.name  +  ".  Your test grades
are:");
System.out.println(std.test1);
System.out.println(std.test2);
System.out.println(std.test3);
```

This would output the name and test grades from the object to which `std` refers. Similarly, `std` can be used to call the `getAverage()` instance method in the object by saying `std.getAverage()`. To print out the student's average, you could say:

```
System.out.println( "Your average is "  +  std.getAverage() );
```

More generally, you could use `std.name` any place where a variable of type *String* is legal. You can use it in expressions. You can assign a value to it. You can even use it to call subroutines from the *String* class. For example, `std.name.length()` is the number of characters in the student's name.

It is possible for a variable like `std`, whose type is given by a class, to refer to no object at all. We say in this case that `std` holds a null pointer or null reference. The null pointer is written in Java as "`null`". You can store a null reference in the variable `std` by saying

```
std = null;
```

`null` is an actual value that is stored in the variable, not a pointer to something else. It is **not** correct to say that the variable "points to null"; in fact, the variable **is** null. For example, you can test whether the value of `std` is null by testing

```
if (std == null) . . .
```

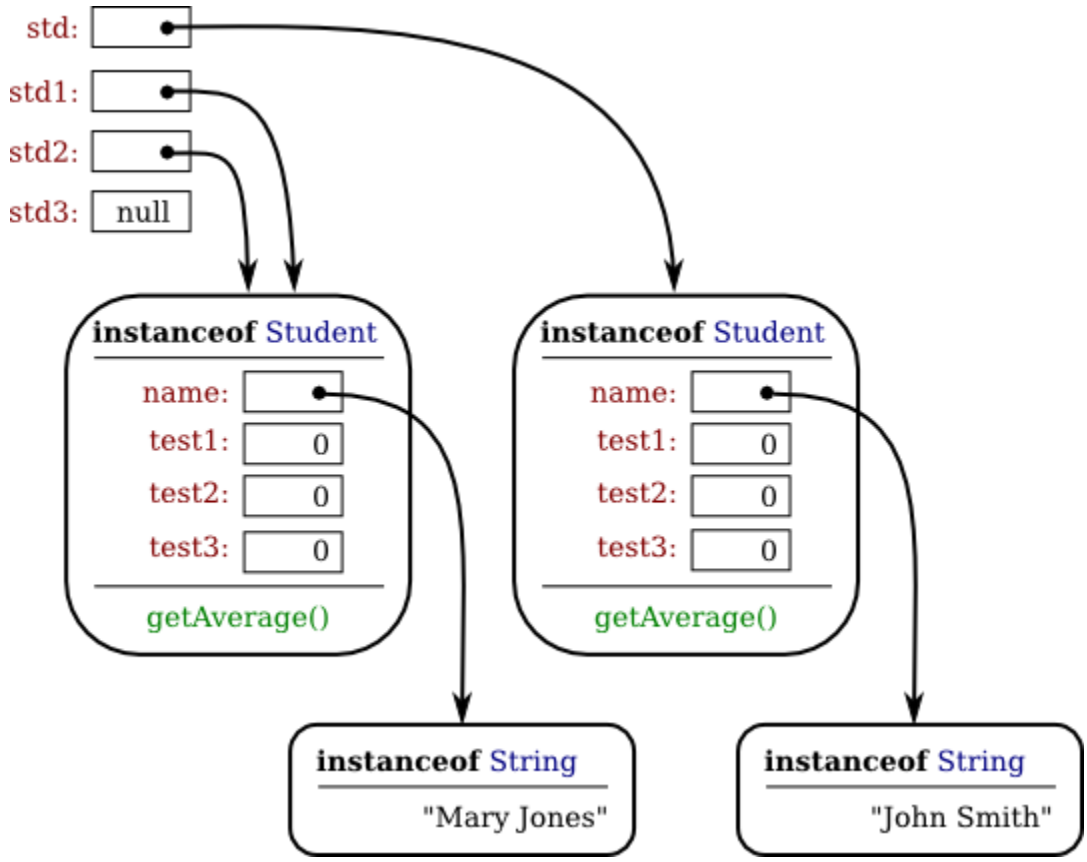
If the value of a variable is `null`, then it is, of course, illegal to refer to instance variables or instance methods through that variable -- since there **is** no object, and hence no instance variables to refer to! For example, if the value of the variable `std` is `null`, then it would be illegal to refer to `std.test1`. If your program attempts to use a null pointer illegally in this way, the result is an error called a null pointer exception. When this happens while the program is running, an exception of type *NullPointerException* is thrown.

Let's look at a sequence of statements that work with objects:

```
Student std, std1,       // Declare four variables of
          std2, std3;    //   type Student.

std = new Student();     // Create a new object belonging
                         //   to the class Student, and
                         //   store a reference to that
                         //   object in the variable std.

std1 = new Student();    // Create a second Student object
                         //   and store a reference to
                         //   it in the variable std1.

std2 = std1;             // Copy the reference value in std1
                         //   into the variable std2.

std3 = null;             // Store a null reference in the
                         //   variable std3.
```

```
std.name = "John Smith";  // Set values of some instance variables.
std1.name = "Mary Jones";

     // (Other instance variables have default
     //    initial values of zero.)
```

After the computer executes these statements, the situation in the computer's memory looks like this:



In this picture, when a variable contains a reference to an object, the value of that variable is shown as an arrow pointing to the object. Note, by the way, that the *Strings* are objects! The variable `std3`, with a value of `null`, doesn't point anywhere. The arrows from `std1` and `std2` both point to the same object. This illustrates a Very Important Point:

When one object variable is assigned
to another, only a reference is copied.
The object referred to is not copied.

When the assignment "`std2 = std1;`" was executed, no new object was created. Instead, `std2` was set to refer to the very same object that `std1` refers to. This is to be expected, since the assignment statement just copies the value that is stored in `std1` into `std2`, and that value is a pointer, not an object. But this has some consequences that might be surprising. For example,

`std1.name` and `std2.name` are two different names for the same variable, namely the instance variable in the object that both `std1` and `std2` refer to. After the string `"Mary Jones"` is assigned to the variable **`std1`**`.name`, it is also true that the value of **`std2`**`.name` is `"Mary Jones"`. There is a potential for a lot of confusion here, but you can help protect yourself from it if you keep telling yourself, "The object is not in the variable. The variable just holds a pointer to the object."

You can test objects for equality and inequality using the operators == and `!=`, but here again, the semantics are different from what you are used to. When you make a test `"if (std1 == std2)"`, you are testing whether the values stored in `std1` and `std2` are the same. But the values that you are comparing are references to objects; they are not objects. So, you are testing whether `std1` and `std2` refer to the same object, that is, whether they point to the same location in memory. This is fine, if its what you want to do. But sometimes, what you want to check is whether the instance variables in the objects have the same values. To do that, you would need to ask whether `"std1.test1 == std2.test1 && std1.test2 == std2.test2 && std1.test3 == std2.test3 && std1.name.equals(std2.name)"`.

I've remarked previously that *Strings* are objects, and I've shown the strings `"Mary Jones"` and `"John Smith"` as objects in the above illustration. (Strings are special objects, treated by Java in a special way, and I haven't attempted to show the actual internal structure of the *String* objects.) Since strings are objects, a variable of type *String* can only hold a reference to a string, not the string itself. This explains why using the == operator to test strings for equality is not a good idea. Suppose that `greeting` is a variable of type *String*, and that it refers to the string `"Hello"`. Then would the test `greeting == "Hello"` be true? Well, maybe, maybe not. The variable `greeting` and the *String* literal `"Hello"` each refer to a string that contains the characters H-e-l-l-o. But the strings could still be different objects, that just happen to contain the same characters; in that case, `greeting == "Hello"` would be false. The function `greeting.equals("Hello")` tests whether `greeting` and `"Hello"` contain the same characters, which is almost certainly the question you want to ask. The expression `greeting == "Hello"` tests whether `greeting` and `"Hello"` contain the same characters **stored in the same memory location**. (Of course, a *String* variable such as `greeting` can also contain the special value `null`, and it **would** make sense to use the == operator to test whether `"greeting == null"`.)

The fact that variables hold references to objects, not objects themselves, has a couple of other consequences that you should be aware of. They follow logically, if you just keep in mind the basic fact that the object is not stored in the variable. The object is somewhere else; the variable points to it.

Suppose that a variable that refers to an object is declared to be `final`. This means that the value stored in the variable can never be changed, once the variable has been initialized. The value stored in the variable is a reference to the object. So the variable will continue to refer to

the same object as long as the variable exists. However, this does not prevent the data **in the object** from changing. The variable is `final`, not the object. It's perfectly legal to say

```
final Student stu = new Student();

stu.name = "John Doe";  // Change data in the object;
                        // The value stored in stu is not changed!
                        // It still refers to the same object.
```

Next, suppose that `obj` is a variable that refers to an object. Let's consider what happens when `obj` is passed as an actual parameter to a subroutine. The value of `obj` is assigned to a formal parameter in the subroutine, and the subroutine is executed. The subroutine has no power to change the value stored in the variable, `obj`. It only has a copy of that value. However, the value is a reference to an object. Since the subroutine has a reference to the object, it can change the data stored **in** the object. After the subroutine ends, `obj` still points to the same object, but the data stored **in the object** might have changed. Suppose `x` is a variable of type int and `stu` is a variable of type `Student`. Compare:

```
void dontChange(int z) {
    z = 42;
}

The lines:

   x = 17;
   dontChange(x);
   System.out.println(x);
System.out.println(stu.name);

output the value 17.

The value of x is not
changed by the subroutine,
which is equivalent to

   z = x;
   z = 42;
```

```
void change(Student s) {
     s.name = "Fred";
}

The lines:

   stu.name = "Jane";
   change(stu);


output the value "Fred".

The value of stu is not
changed, but stu.name is

This is equivalent to

   s = stu;
   s.name = "Fred";
```

5.1.3 Getters and Setters

When writing new classes, it's a good idea to pay attention to the issue of access control. Recall that making a member of a class `public` makes it accessible from anywhere, including from other classes. On the other hand, a `private` member can only be used in the class where it is defined.

In the opinion of many programmers, almost all member variables should be declared `private`. This gives you complete control over what can be done with the variable. Even if the variable itself is private, you can allow other classes to find out what its value is by providing a `public`

accessor method that returns the value of the variable. For example, if your class contains a `private` member variable, `title`, of type *String*, you can provide a method

```
public String getTitle() {
    return title;
}
```

that returns the value of `title`. By convention, the name of an accessor method for a variable is obtained by capitalizing the name of variable and adding "get" in front of the name. So, for the variable `title`, we get an accessor method named "get" + "Title", or `getTitle()`. Because of this naming convention, accessor methods are more often referred to as getter methods. A getter method provides "read access" to a variable. (Sometimes for boolean variables, "is" is used in place of "get". For example, a getter for a boolean member variable named `done` might be called `isDone()`.)

You might also want to allow "write access" to a `private` variable. That is, you might want to make it possible for other classes to specify a new value for the variable. This is done with a setter method. (If you don't like simple, Anglo-Saxon words, you can use the fancier term mutator method.) The name of a setter method should consist of "set" followed by a capitalized copy of the variable's name, and it should have a parameter with the same type as the variable. A setter method for the variable `title` could be written

```
public void setTitle( String newTitle ) {
   title = newTitle;
}
```

It is actually very common to provide both a getter and a setter method for a private member variable. Since this allows other classes both to see and to change the value of the variable, you might wonder why not just make the variable `public`? The reason is that getters and setters are not restricted to simply reading and writing the variable's value. In fact, they can take any action at all. For example, a getter method might keep track of the number of times that the variable has been accessed:

```
public String getTitle() {
    titleAccessCount++;  // Increment member variable
titleAccessCount.
    return title;
}
```

and a setter method might check that the value that is being assigned to the variable is legal:

```
public void setTitle( String newTitle ) {
   if ( newTitle == null )   // Don't allow null strings as titles!
      title = "(Untitled)";  // Use an appropriate default value
instead.
   else
      title = newTitle;
}
```

Even if you can't think of any extra chores to do in a getter or setter method, you might change your mind in the future when you redesign and improve your class. If you've used a getter and setter from the beginning, you can make the modification to your class without affecting any of the classes that use your class. The `private` member variable is not part of the public interface of your class; only the `public` getter and setter methods are, and you are free to change their implementations without changing the public interface of your class. If you **haven't** used get and set from the beginning, you'll have to contact everyone who uses your class and tell them, "Sorry people, you'll have to track down every use that you've made of this variable and change your code to use my new get and set methods instead."

A couple of final notes: Some advanced aspects of Java rely on the naming convention for getter and setter methods, so it's a good idea to follow the convention rigorously. And though I've been talking about using getter and setter methods for a variable, you can define get and set methods even if there is no variable. A getter and/or setter method defines a property of the class, that might or might not correspond to a variable. For example, if a class includes a `public void` instance method with signature `setValue(double)`, then the class has a "property" named `value` of type double, and it has this property whether or not the class has a member variable named `value`.

5.1.4 Arrays and Objects

As I noted in Subsection 3.8.1, arrays are objects. Like *Strings* they are special objects, with their own unique syntax. An array type such as `int[]` or `String[]` is actually a class, and arrays are created using a special version of the `new` operator. As in the case for other object variables, an array variable can never hold an actual array -- only a reference to an array object. The array object itself exists in the heap. It is possible for an array variable to hold the value `null`, which means there is no actual array.

For example, suppose that `list` is a variable of type `int[]`. If the value of `list` is `null`, then any attempt to access `list.length` or an array element `list[i]` would be an error and would cause an exception of type *NullPointerException*. If `newlist` is another variable of type `int[]`, then the assignment statement

```
newlist = list;
```

only copies the reference value in `list` into `newlist`. If `list` is `null`, the result is that `newlist` will also be `null`. If `list` points to an array, the assignment statement does **not** make a copy of the array. It just sets `newlist` to refer to the same array as `list`. For example, the output of the following code segment

```
list = new int[3];
list[1] = 17;
newlist = list; // newlist points to the same array as list!
newlist[1] = 42;
```

```
System.out.println( list[1] );
```

would be 42, not 17, since `list[1]` and `newlist[1]` are just different names for the same element in the array. All this is very natural, once you understand that arrays are objects and array variables hold pointers to arrays.

This fact also comes into play when an array is passed as a parameter to a subroutine. The value that is copied into the subroutine is a pointer to the array. The array is not copied. Since the subroutine has a reference to the original array, any changes that it makes to elements of the array are being made to the original and will persist after the subroutine returns.

Arrays are objects. They can also hold objects. The base type of an array can be a class. We have already seen this when we used arrays of type `String[]`, but any class can be used as the base type. For example, suppose *Student* is the class defined earlier in this section. Then we can have arrays of type `Student[]`. For an array of type `Student[]`, each element of the array is a variable of type *Student*. To store information about 30 students, we could use an array

```
Student[] classlist;  // Declare a variable of type Student[].
classlist = new Student[30];  // The variable now points to an
array.
```

The array has 30 elements, `classlist[0]`, `classlist[1]`, ... `classlist[29]`. When the array is created, it is filled with the default initial value, which for an object type is `null`. So, although we have 30 array elements of type *Student*, we don't yet have any actual *Student* objects! All we have is 30 nulls. If we want student objects, we have to create them:

```
Student[] classlist;
classlist = new Student[30];
for ( int i = 0; i < 30; i++ ) {
    classlist[i] = new Student();
}
```

Once we have done this, each `classlist[i]` points to an object of type *Student*. If we want to talk about the name of student number 3, we can use `student[3].name`. The average for student number `i` can be computed by calling `classlist[i].getAverage()`. You can do anything with `classlist[i]` that you could do with any other variable of type *Student*.

Constructors and Object Initialization

OBJECT TYPES IN JAVA are very different from the primitive types. Simply declaring a variable whose type is given as a class does not automatically create an object of that class. Objects must be explicitly constructed. For the computer, the process of constructing an object

means, first, finding some unused memory in the heap that can be used to hold the object and, second, filling in the object's instance variables. As a programmer, you don't care where in memory the object is stored, but you will usually want to exercise some control over what initial values are stored in a new object's instance variables. In many cases, you will also want to do more complicated initialization or bookkeeping every time an object is created.

5.2.1 Initializing Instance Variables

An instance variable can be assigned an initial value in its declaration, just like any other variable. For example, consider a class named *PairOfDice*. An object of this class will represent a pair of dice. It will contain two instance variables to represent the numbers showing on the dice and an instance method for rolling the dice:

```
public class PairOfDice {

    public int die1 = 3;    // Number showing on the first die.
    public int die2 = 4;    // Number showing on the second die.

    public void roll() {
            // Roll the dice by setting each of the dice to be
            // a random number between 1 and 6.
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

The instance variables `die1` and `die2` are initialized to the values 3 and 4 respectively. These initializations are executed whenever a *PairOfDice* object is constructed. It's important to understand when and how this happens. There can be many *PairOfDice* objects. Each time one is created, it gets its own instance variables, and the assignments "`die1 = 3`" and "`die2 = 4`" are executed to fill in the values of those variables. To make this clearer, consider a variation of the *PairOfDice* class:

```
public class PairOfDice {

    public int die1 = (int)(Math.random()*6) + 1;
    public int die2 = (int)(Math.random()*6) + 1;

    public void roll() {
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

Here, every time a new *PaifOfDice* is created, the dice are initialized to random values, as if a new pair of dice were being thrown onto the gaming table. Since the initialization is executed for each new object, a set of random initial values will be computed for each new pair of dice.

Different pairs of dice can have different initial values. For initialization of **static** member variables, of course, the situation is quite different. There is only one copy of a `static` variable, and initialization of that variable is executed just once, when the class is first loaded.

If you don't provide any initial value for an instance variable, a default initial value is provided automatically. Instance variables of numerical type (int, double, etc.) are automatically initialized to zero if you provide no other values; boolean variables are initialized to `false`; and char variables, to the Unicode character with code number zero. An instance variable can also be a variable of object type. For such variables, the default initial value is `null`. (In particular, since `Strings` are objects, the default initial value for *String* variables is `null`.)

5.2.2 Constructors

Objects are created with the operator, `new`. For example, a program that wants to use a *PairOfDice* object could say:

```
PairOfDice dice;   // Declare a variable of type PairOfDice.

dice = new PairOfDice();  // Construct a new object and store a
                          //   reference to it in the variable.
```

In this example, "`new PairOfDice()`" is an expression that allocates memory for the object, initializes the object's instance variables, and then returns a reference to the object. This reference is the value of the expression, and that value is stored by the assignment statement in the variable, `dice`, so that after the assignment statement is executed, `dice` refers to the newly created object. Part of this expression, "`PairOfDice()`", looks like a subroutine call, and that is no accident. It is, in fact, a call to a special type of subroutine called a constructor. This might puzzle you, since there is no such subroutine in the class definition. However, every class has at least one constructor. If the programmer doesn't write a constructor definition in a class, then the system will provide a default constructor for that class. This default constructor does nothing beyond the basics: allocate memory and initialize instance variables. If you want more than that to happen when an object is created, you can include one or more constructors in the class definition.

The definition of a constructor looks much like the definition of any other subroutine, with three exceptions. A constructor does not have any return type (not even `void`). The name of the constructor must be the same as the name of the class in which it is defined. And the only modifiers that can be used on a constructor definition are the access modifiers `public`, `private`, and `protected`. (In particular, a constructor can't be declared `static`.)

However, a constructor does have a subroutine body of the usual form, a block of statements. There are no restrictions on what statements can be used. And a constructor can have a list of formal parameters. In fact, the ability to include parameters is one of the main reasons for using constructors. The parameters can provide data to be used in the construction of the object. For

example, a constructor for the *PairOfDice* class could provide the values that are initially showing on the dice. Here is what the class would look like in that case:

```
public class PairOfDice {

    public int die1;   // Number showing on the first die.
    public int die2;   // Number showing on the second die.

    public PairOfDice(int val1, int val2) {
            // Constructor.  Creates a pair of dice that
            // are initially showing the values val1 and val2.
         die1 = val1;  // Assign specified values
         die2 = val2;  //           to the instance variables.
    }

    public void roll() {
            // Roll the dice by setting each of the dice to be
            // a random number between 1 and 6.
         die1 = (int)(Math.random()*6) + 1;
         die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

The constructor is declared as "`public PairOfDice(int val1, int val2) ...`", with no return type and with the same name as the name of the class. This is how the Java compiler recognizes a constructor. The constructor has two parameters, and values for these parameters must be provided when the constructor is called. For example, the expression "`new PairOfDice(3,4)`" would create a *PairOfDice* object in which the values of the instance variables `die1` and `die2` are initially 3 and 4. Of course, in a program, the value returned by the constructor should be used in some way, as in

```
PairOfDice dice;            // Declare a variable of type
PairOfDice.

dice = new PairOfDice(1,1); // Let dice refer to a new PairOfDice
                            //   object that initially shows 1, 1.
```

Now that we've added a constructor to the *PairOfDice* class, we can no longer create an object by saying "`new PairOfDice()`"! The system provides a default constructor for a class **only** if the class definition does not already include a constructor. In this version of *PariOfDice*, there is only one constructor in the class, and it requires two actual parameters. However, this is not a big problem, since we can add a second constructor to the class, one that has no parameters. In fact, you can have as many different constructors as you want, as long as their signatures are different, that is, as long as they have different numbers or types of formal parameters. In the *PairOfDice* class, we might have a constructor with no parameters which produces a pair of dice showing random numbers:

```
public class PairOfDice {

    public int die1;   // Number showing on the first die.
```

```
    public int die2;   // Number showing on the second die.

    public PairOfDice() {
            // Constructor.  Rolls the dice, so that they initially
            // show some random values.
        roll();  // Call the roll() method to roll the dice.
    }

    public PairOfDice(int val1, int val2) {
            // Constructor.  Creates a pair of dice that
            // are initially showing the values val1 and val2.
        die1 = val1;  // Assign specified values
        die2 = val2;  //            to the instance variables.
    }

    public void roll() {
            // Roll the dice by setting each of the dice to be
            // a random number between 1 and 6.
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

Now we have the option of constructing a *PairOfDice* object either with "`new PairOfDice()`" or with "`new PairOfDice(x,y)`", where `x` and `y` are int-valued expressions.

This class, once it is written, can be used in any program that needs to work with one or more pairs of dice. None of those programs will ever have to use the obscure incantation "`(int)(Math.random()*6)+1`", because it's done inside the *PairOfDice* class. And the programmer, having once gotten the dice-rolling thing straight will never have to worry about it again. Here, for example, is a main program that uses the *PairOfDice* class to count how many times two pairs of dice are rolled before the two pairs come up showing the same value. This illustrates once again that you can create several instances of the same class:

```
public class RollTwoPairs {

    public static void main(String[] args) {

        PairOfDice firstDice;  // Refers to the first pair of dice.
        firstDice = new PairOfDice();

        PairOfDice secondDice; // Refers to the second pair of
dice.
        secondDice = new PairOfDice();

        int countRolls;  // Counts how many times the two pairs of
                         //    dice have been rolled.

        int total1;      // Total showing on first pair of dice.
        int total2;      // Total showing on second pair of dice.

        countRolls = 0;
```

```
        do {  // Roll the two pairs of dice until totals are the
same.

            firstDice.roll();    // Roll the first pair of dice.
            total1 = firstDice.die1 + firstDice.die2;   // Get
total.
            System.out.println("First pair comes up  " + total1);

            secondDice.roll();    // Roll the second pair of dice.
            total2 = secondDice.die1 + secondDice.die2;   // Get
total.
            System.out.println("Second pair comes up " + total2);

            countRolls++;   // Count this roll.

            System.out.println();  // Blank line.

        } while (total1 != total2);

        System.out.println("It took " + countRolls
                           + " rolls until the totals were the
same.");

    } // end main()

} // end class RollTwoPairs
```

Constructors are subroutines, but they are subroutines of a special type. They are certainly not instance methods, since they don't belong to objects. Since they are responsible for creating objects, they exist before any objects have been created. They are more like `static` member subroutines, but they are not and cannot be declared to be `static`. In fact, according to the Java language specification, they are technically not members of the class at all! In particular, constructors are not referred to as "methods."

Unlike other subroutines, a constructor can only be called using the `new` operator, in an expression that has the form

```
new class-name ( parameter-list )
```

where the **parameter-list** is possibly empty. I call this an expression because it computes and returns a value, namely a reference to the object that is constructed. Most often, you will store the returned reference in a variable, but it is also legal to use a constructor call in other ways, for example as a parameter in a subroutine call or as part of a more complex expression. Of course, if you don't save the reference in a variable, you won't have any way of referring to the object that was just created.

A constructor call is more complicated than an ordinary subroutine or function call. It is helpful to understand the exact steps that the computer goes through to execute a constructor call:

1. First, the computer gets a block of unused memory in the heap, large enough to hold an object of the specified type.
2. It initializes the instance variables of the object. If the declaration of an instance variable specifies an initial value, then that value is computed and stored in the instance variable. Otherwise, the default initial value is used.
3. The actual parameters in the constructor, if any, are evaluated, and the values are assigned to the formal parameters of the constructor.
4. The statements in the body of the constructor, if any, are executed.
5. A reference to the object is returned as the value of the constructor call.

The end result of this is that you have a reference to a newly constructed object.

For another example, let's rewrite the *Student* class that was used in Section 1. I'll add a constructor, and I'll also take the opportunity to make the instance variable, `name`, private.

```
public class Student {

   private String name;                  // Student's name.
   public double test1, test2, test3;    // Grades on three tests.

   Student(String theName) {
        // Constructor for Student objects;
        // provides a name for the Student.
        // The name can't be null.
      if ( theName == null )
          throw new IllegalArgumentException("name can't be null");
      name = theName;
   }

   public String getName() {
        // Getter method for reading the value of the private
        // instance variable, name.
      return name;
   }

   public double getAverage() {
        // Compute average test grade.
      return (test1 + test2 + test3) / 3;
   }

}  // end of class Student
```

An object of type *Student* contains information about some particular student. The constructor in this class has a parameter of type *String*, which specifies the name of that student. Objects of type *Student* can be created with statements such as:

```
std = new Student("John Smith");
std1 = new Student("Mary Jones");
```

In the original version of this class, the value of `name` had to be assigned by a program after it created the object of type *Student*. There was no guarantee that the programmer would always remember to set the `name` properly. In the new version of the class, there is no way to create a *Student* object except by calling the constructor, and that constructor automatically sets the `name`. Furthermore, the constructor makes it impossible to have a student object whose name is `null`. The programmer's life is made easier, and whole hordes of frustrating bugs are squashed before they even have a chance to be born.

Another type of guarantee is provided by the `private` modifier. Since the instance variable, `name`, is `private`, there is no way for any part of the program outside the *Student* class to get at the `name` directly. The program sets the value of `name`, indirectly, when it calls the constructor. I've provided a getter function, `getName()`, that can be used from outside the class to find out the `name` of the student. But I haven't provided any setter method or other way to change the name. Once a student object is created, it keeps the same name as long as it exists.

Note that it would be legal, and good style, to declare the variable `name` to be "`final`" in this class. An instance variable can be `final` provided it is either assigned a value in its declaration or is assigned a value in every constructor in the class. It is illegal to assign a value to a `final` instance variable, except inside a constructor.

Let's take this example a little farther to illustrate one more aspect of classes: What happens when you mix static and non-static in the same class? In that case, it's legal for an instance method in the class to use static member variables or call static member subroutines. An object knows what class it belongs to, and it can refer to static members of that class. But there it only one copy of the static member, in the class itself. Effectively, all the objects share one copy of the static member.

As an example, consider a version of the *Student* class to which I've added an `ID` for each student and a `static` member called `nextUniqueID`. Although there is an `ID` variable in each student object, there is only one `nextUniqueID` variable.

```
public class Student {

   private String name;                  // Student's name.
   public double test1, test2, test3;   // Grades on three tests.

   private int ID;  // Unique ID number for this student.

   private static int nextUniqueID = 0;
             // keep track of next available unique ID number

   Student(String theName) {
        // Constructor for Student objects; provides a name for the
Student,
        // and assigns the student a unique ID number.
      name = theName;
```

```
        nextUniqueID++;
        ID = nextUniqueID;
    }

    public String getName() {
         // Getter method for reading the value of the private
         // instance variable, name.
        return name;
    }

    public int getID() {
         // Getter method for reading the value of ID.
        return ID;
    }

    public double getAverage() {
         // Compute average test grade.
        return (test1 + test2 + test3) / 3;
    }

}  // end of class Student
```

Since `nextUniqueID` is a `static` variable, the initialization "`nextUniqueID = 0`" is done only once, when the class is first loaded. Whenever a *Student* object is constructed and the constructor says "`nextUniqueID++;`", it's always the same static member variable that is being incremented. When the very first *Student* object is created, `nextUniqueID` becomes 1. When the second object is created, `nextUniqueID` becomes 2. After the third object, it becomes 3. And so on. The constructor stores the new value of `nextUniqueID` in the `ID` variable of the object that is being created. Of course, `ID` is an instance variable, so every object has its own individual `ID` variable. The class is constructed so that each student will automatically get a different value for its `ID` variable. Furthermore, the `ID` variable is `private`, so there is no way for this variable to be tampered with after the object has been created. You are guaranteed, just by the way the class is designed, that every student object will have its own permanent, unique identification number. Which is kind of cool if you think about it.

(Unfortunately, if you think about it a bit more, it turns out that the guarantee isn't quite absolute. The guarantee is valid in programs that use a single thread. But, as a preview of the difficulties of parallel programming, I'll note that in multi-threaded programs, where several things can be going on at the same time, things can get a bit strange. In a multi-threaded program, it is possible that two threads are creating *Student* objects at exactly the same time, and it becomes possible for both objects to get the same ID number. We'll come back to this in Subsection 12.1.3, where you will learn how to fix the problem.)

5.2.3 Garbage Collection

So far, this section has been about creating objects. What about destroying them? In Java, the destruction of objects takes place automatically.

An object exists in the heap, and it can be accessed only through variables that hold references to the object. What should be done with an object if there are no variables that refer to it? Such things can happen. Consider the following two statements (though in reality, you'd never do anything like this in consecutive statements!):

```
Student std = new Student("John Smith");
std = null;
```

In the first line, a reference to a newly created *Student* object is stored in the variable `std`. But in the next line, the value of `std` is changed, and the reference to the *Student* object is gone. In fact, there are now no references whatsoever to that object, in any variable. So there is no way for the program ever to use the object again! It might as well not exist. In fact, the memory occupied by the object should be reclaimed to be used for another purpose.

Java uses a procedure called garbage collection to reclaim memory occupied by objects that are no longer accessible to a program. It is the responsibility of the system, not the programmer, to keep track of which objects are "garbage." In the above example, it was very easy to see that the *Student* object had become garbage. Usually, it's much harder. If an object has been used for a while, there might be several references to the object stored in several variables. The object doesn't become garbage until all those references have been dropped.

In many other programming languages, it's the programmer's responsibility to delete the garbage. Unfortunately, keeping track of memory usage is very error-prone, and many serious program bugs are caused by such errors. A programmer might accidently delete an object even though there are still references to that object. This is called a dangling pointer error, and it leads to problems when the program tries to access an object that is no longer there. Another type of error is a memory leak, where a programmer neglects to delete objects that are no longer in use. This can lead to filling memory with objects that are completely inaccessible, and the program might run out of memory even though, in fact, large amounts of memory are being wasted.

Because Java uses garbage collection, such errors are simply impossible. Garbage collection is an old idea and has been used in some programming languages since the 1960s. You might wonder why all languages don't use garbage collection. In the past, it was considered too slow and wasteful. However, research into garbage collection techniques combined with the incredible speed of modern computers have combined to make garbage collection feasible. Programmers should rejoice.

Programming with Objects

THERE ARE SEVERAL WAYS in which object-oriented concepts can be applied to the process of designing and writing programs. The broadest of these is object-oriented analysis and design which applies an object-oriented methodology to the earliest stages of program development, during which the overall design of a program is created. Here, the idea is to identify things in the problem domain that can be modeled as objects. On another level, object-

oriented programming encourages programmers to produce generalized software components that can be used in a wide variety of programming projects.

Of course, for the most part, you will experience "generalized software components" by using the standard classes that come along with Java. We begin this section by looking at some built-in classes that are used for creating objects. At the end of the section, we will get back to generalities.

5.3.1 Some Built-in Classes

Although the focus of object-oriented programming is generally on the design and implementation of new classes, it's important not to forget that the designers of Java have already provided a large number of reusable classes. Some of these classes are meant to be extended to produce new classes, while others can be used directly to create useful objects. A true mastery of Java requires familiarity with a large number of built-in classes -- something that takes a lot of time and experience to develop. Let's take a moment to look at a few built-in classes that you might find useful.

A string can be built up from smaller pieces using the + operator, but this is not always efficient. If `str` is a *String* and `ch` is a character, then executing the command "`str = str + ch;`" involves creating a whole new string that is a copy of `str`, with the value of `ch` appended onto the end. Copying the string takes some time. Building up a long string letter by letter would require a surprising amount of processing. The class *StringBuilder* makes it possible to be efficient about building up a long string from a number of smaller pieces. To do this, you must make an object belonging to the *StringBuilder* class. For example:

```
StringBuilder builder = new StringBuilder();
```

(This statement both declares the variable `builder` and initializes it to refer to a newly created *StringBuilder* object. Combining declaration with initialization was covered in Subsection 4.7.1 and works for objects just as it does for primitive types.)

Like a *String*, a *StringBuilder* contains a sequence of characters. However, it is possible to add new characters onto the end of a `StringBuilder` without continually making copies of the data that it already contains. If `x` is a value of any type and `builder` is the variable defined above, then the command `builder.append(x)` will add `x`, converted into a string representation, onto the end of the data that was already in the builder. This can be done more efficiently than copying the data every time something is appended. A long string can be built up in a *StringBuilder* using a sequence of `append()` commands. When the string is complete, the function `builder.toString()` will return a copy of the string in the builder as an ordinary value of type *String*. The *StringBuilder* class is in the standard package `java.lang`, so you can use its simple name without importing it.

A number of useful classes are collected in the package `java.util`. For example, this package contains classes for working with collections of objects. We will study such collection classes extensively in Chapter 10. Another class in this package, `java.util.Date`, is used to represent times. When a `Date` object is constructed without parameters, the result represents the current date and time, so an easy way to display this information is:

```
System.out.println( new Date() );
```

Of course, since it is in the package `java.util`, in order to use the `Date` class in your program, you must make it available by importing it with one of the statements "`import java.util.Date;`" or "`import java.util.*;`" at the beginning of your program. (See Subsection 4.5.3 for a discussion of packages and `import`.)

I will also mention the class `java.util.Random`. An object belonging to this class is a *source* of random numbers (or, more precisely pseudorandom numbers). The standard function `Math.random()` uses one of these objects behind the scenes to generate its random numbers. An object of type `Random` can generate random integers, as well as random real numbers. If `randGen` is created with the command:

```
Random randGen = new Random();
```

and if `N` is a positive integer, then `randGen.nextInt(N)` generates a random integer in the range from `0` to `N-1`. For example, this makes it a little easier to roll a pair of dice. Instead of saying "`die1 = (int)(6*Math.random())+1;`", one can say "`die1 = randGen.nextInt(6)+1;`". (Since you also have to import the class `java.util.Random` and create the `Random` object, you might not agree that it is actually easier.) An object of type `Random` can also be used to generate so-called Gaussian distributed random real numbers.

Many of Java's standard classes are used in GUI programming. You will encounter a number of them in the Chapter 6. Here, I will mention only the class *Color*, from the package `java.awt`, so that I can use it in the next example. A *Color* object represents a color that can be used for drawing. In Section 3.9, you encountered color constants such as `Color.RED`. These constants are final, static member variables in the *Color* class, and their values are objects of type *Color*. It is also possible to create new color objects. Class *Color* has several constructors. One of them, which is called as `new Color(r,g,b)`, takes three integer parameters to specify the red, green, and blue components of the color. The parameters must be in the range 0 to 255. Another constructor, `new Color(r,g,b,t)`, adds a fourth integer parameter, which must also be in the range 0 to 255. The fourth parameter determines how transparent or opaque the color is. When you draw with a transparent color, the background shows through the color to some extent. A larger value of the parameter `t` gives a color that is less transparent and more opaque. (If you know hexadecimal color codes, there is a constructor for you. It takes one parameter of type int that encodes all the color components. For example, `new Color(0x8B5413)` creates a brown color.)

A *Color* object has only a few instance methods that you are likely to use. Mainly, there are functions like `getRed()` to get the individual color components of the color. There are no "setter" methods to change the color components. In fact, a *Color* is an immutable object, meaning that all of its instance variables are `final` and cannot be changed after the object is created. *Strings* are another example of immutable objects, and we will make some of our own later in this section.

The main point of all this, again, is that many problems have already been solved, and the solutions are available in Java's standard classes. If you are faced with a task that looks like it should be fairly common, it might be worth looking through a Java reference to see whether someone has already written a class that you can use.

5.3.2 The class "Object"

We have already seen that one of the major features of object-oriented programming is the ability to create subclasses of a class. The subclass inherits all the properties or behaviors of the class, but can modify and add to what it inherits. In Section 5.5, you'll learn how to create subclasses. What you don't know yet is that **every** class in Java (with just one exception) is a subclass of some other class. If you create a class and don't explicitly make it a subclass of some other class, then it automatically becomes a subclass of the special class named *Object*. (*Object* is the one class that is not a subclass of any other class.)

Class *Object* defines several instance methods that are inherited by every other class. These methods can be used with any object whatsoever. I will mention just one of them here. You will encounter more of them later in the book.

The instance method `toString()` in class *Object* returns a value of type *String* that is supposed to be a string representation of the object. You've already used this method implicitly, any time you've printed out an object or concatenated an object onto a string. When you use an object in a context that requires a string, the object is automatically converted to type *String* by calling its `toString()` method.

The version of `toString` that is defined in *Object* just returns the name of the class that the object belongs to, concatenated with a code number called the hash code of the object; this is not very useful. When you create a class, you can write a new `toString()` method for it, which will replace the inherited version. For example, we might add the following method to any of the *PairOfDice* classes from the previous section:

```
/**
 * Return a String representation of a pair of dice, where die1
 * and die2 are instance variables containing the numbers that are
 * showing on the two dice.
 */
public String toString() {
   if (die1 == die2)
```

```
            return "double " + die1;
        else
            return die1 + " and " + die2;
    }
```

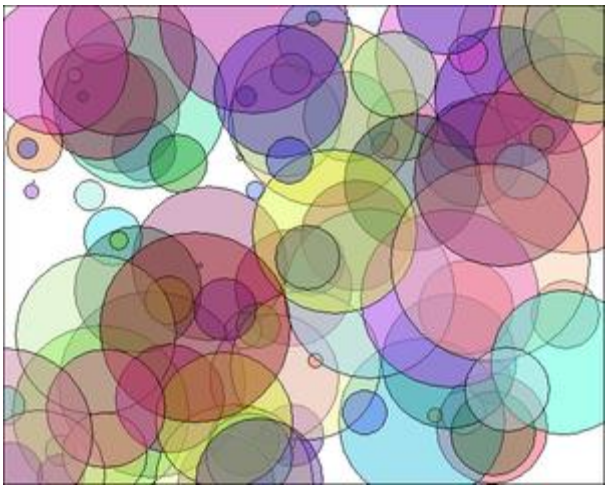
If `dice` refers to a *PairOfDice* object, then `dice.toString()` will return strings such as "3 and 6", "5 and 1", and "double 2", depending on the numbers showing on the dice. This method would be used automatically to convert `dice` to type *String* in a statement such as

```
System.out.println( "The dice came up " + dice );
```

so this statement might output, "The dice came up 5 and 1" or "The dice came up double 2". You'll see another example of a `toString()` method in the next section.

5.3.3 Writing and Using a Class

As an example, we will write an animation program, based on the same animation framework that was used in Subsection 3.9.3. The animation shows a number of semi-transparent disk that grow in size as the animation plays. The disks have random colors and locations. When a disk gets too big, or sometimes just at random, the disk disappears and is replaced with a new disk at a random location. Here is a screenshot from the program:



A disk in this program can be represented as an object. A disk has properties -- color, location, and size -- that can be instance variables in the object. As for instance methods, we need to think about what we might want to do with a disk. An obvious candidate is that we need to be able to draw it, so we can include an instance method `draw(g)`, where `g` is a graphics context that will be used to do the drawing. The class can also include one or more constructors. A constructor initializes the object. It's not always clear what data should be provided as parameters to the constructors. In this case, as an example, the constructor's parameters specify the location and size for the circle, but the constructor makes up a color using random values for the red, green, and blue components. Here's the complete class:

```
import java.awt.Color;  // import some standard GUI classes
import java.awt.Graphics;

/**
 * A simple class that holds the size, color, and location of a
colored disk,
 * with a method for drawing the filled circle in a graphics
context.  The
 * circle is drawn as a filled oval, with a black outline.
 */
public class CircleInfo {

    public int radius;    // The radius of the circle.
    public int x,y;       // The location of the center of the
circle.
    public Color color;   // The color of the circle.

    /**
     * Create a CircleInfo with a given location and radius and
with a
     * randomly selected, semi-transparent color.
     * @param centerX   The x coordinate of the center.
     * @param centerY   The y coordinate of the center.
     * @param rad       The radius of the circle.
     */
    public CircleInfo( int centerX, int centerY, int rad ) {
        x = centerX;
        y = centerY;
        radius = rad;
        int red = (int)(255*Math.random());
        int green = (int)(255*Math.random());
        int blue = (int)(255*Math.random());
        color = new Color(red,green,blue, 100);
    }

    /**
     * Draw the disk in graphics context g, with a black outline.
     */
    public void draw( Graphics g ) {
        g.setColor( color );
        g.fillOval( x - radius, y - radius, 2*radius, 2*radius );
        g.setColor( Color.BLACK );
        g.drawOval( x - radius, y - radius, 2*radius, 2*radius );
    }
}
```

It would probably be better style to write getters and setters for the instance variables, but to keep things simple, I made them public.

The main program for my animation is a class *GrowingCircleAnimation*. The program uses 100 disks, each one represented by an object of type *CircleInfo*. To make that manageable, the program uses an array of objects. The array variable is a member variable in the class:

```
private CircleInfo[] circleData; // holds the data for all 100
circles
```

Note that it is not `static`. GUI programming generally uses objects rather than static variables and methods. Basically, this is because we can imagine having several *GrowingCircleAnimations* going on at the same time, each with its own array of disks. Each animation would be represented by an object, and each object will need to have its own `circleData` instance variable. If `circleData` were static, there would only be one array and all the animations would show exactly the same thing.

The array must be created and filled with data. The array is created using `new CircleInfo[100]`, and then 100 objects of type *CircleInfo* are created to fill the array. The new objects are created with random locations and sizes. In the program, this is done before drawing the first frame of the animation. Here is the code, where `width` and `height` are the size of the drawing area:

```
circleData = new CircleInfo[100];  // create the array

for (int i = 0; i < circleData.length; i++) { // create the objects
    circleData[i] = new CircleInfo(
                              (int)(width*Math.random()),
                              (int)(height*Math.random()),
                              (int)(100*Math.random()) );
}
```

In each frame, the radius of the disk is increased and the disk is drawn using the code

```
circleData[i].radius++;
circleData[i].draw(g);
```

These statements look complicated, so let's unpack them. Now, `circleData[i]` is an element of the array `circleData`. That means that it is a variable of type *CircleInfo*. This variable refers to an object of type *CircleInfo*, which contains a public instance variable named `radius`. This means that `circleData[i].radius` is the full name for that variable. Since it is a variable of type int, we can use the `++` operator to increment its value. So the effect of `circleData[i].radius++` is to increase the radius of the circle by one. The second line of code is similar, but in that statement, `circleData[i].draw` is an instance method in the *CircleInfo* object. The statement `circleData[i].draw(g)` calls that instance method with parameter `g`.

The source code example *GrowingCircleAnimation.java* contains the full source code for the program, if you are interested. Since the program uses class *CircleInfo*, you will also need a copy of *CircleInfo.java* in order to compile and run the program.

5.3.4 Object-oriented Analysis and Design

Every programmer builds up a stock of techniques and expertise expressed as snippets of code that can be reused in new programs using the tried-and-true method of cut-and-paste: The old

code is physically copied into the new program and then edited to customize it as necessary. The problem is that the editing is error-prone and time-consuming, and the whole enterprise is dependent on the programmer's ability to pull out that particular piece of code from last year's project that looks like it might be made to fit. (On the level of a corporation that wants to save money by not reinventing the wheel for each new project, just keeping track of all the old wheels becomes a major task.)

Well-designed classes are software components that can be reused without editing. A well-designed class is not carefully crafted to do a particular job in a particular program. Instead, it is crafted to model some particular type of object or a single coherent concept. Since objects and concepts can recur in many problems, a well-designed class is likely to be reusable without modification in a variety of projects.

Furthermore, in an object-oriented programming language, it is possible to make subclasses of an existing class. This makes classes even more reusable. If a class needs to be customized, a subclass can be created, and additions or modifications can be made in the subclass without making any changes to the original class. This can be done even if the programmer doesn't have access to the source code of the class and doesn't know any details of its internal, hidden implementation.

The *PairOfDice* class in the previous section is already an example of a generalized software component, although one that could certainly be improved. The class represents a single, coherent concept, "a pair of dice." The instance variables hold the data relevant to the state of the dice, that is, the number showing on each of the dice. The instance method represents the behavior of a pair of dice, that is, the ability to be rolled. This class would be reusable in many different programming projects.

On the other hand, the *Student* class from the previous section is not very reusable. It seems to be crafted to represent students in a particular course where the grade will be based on three tests. If there are more tests or quizzes or papers, it's useless. If there are two people in the class who have the same name, we are in trouble (one reason why numerical student ID's are often used). Admittedly, it's much more difficult to develop a general-purpose student class than a general-purpose pair-of-dice class. But this particular *Student* class is good mostly as an example in a programming textbook.

A large programming project goes through a number of stages, starting with specification of the problem to be solved, followed by analysis of the problem and design of a program to solve it. Then comes coding, in which the program's design is expressed in some actual programming language. This is followed by testing and debugging of the program. After that comes a long period of maintenance, which means fixing any new problems that are found in the program and modifying it to adapt it to changing requirements. Together, these stages form what is called the software life cycle. (In the real world, the ideal of consecutive stages is seldom if ever achieved.

During the analysis stage, it might turn out that the specifications are incomplete or inconsistent. A problem found during testing requires at least a brief return to the coding stage. If the problem is serious enough, it might even require a new design. Maintenance usually involves redoing some of the work from previous stages....)

Large, complex programming projects are only likely to succeed if a careful, systematic approach is adopted during all stages of the software life cycle. The systematic approach to programming, using accepted principles of good design, is called software engineering. The software engineer tries to efficiently construct programs that verifiably meet their specifications and that are easy to modify if necessary. There is a wide range of "methodologies" that can be applied to help in the systematic design of programs. (Most of these methodologies seem to involve drawing little boxes to represent program components, with labeled arrows to represent relationships among the boxes.)

We have been discussing object orientation in programming languages, which is relevant to the coding stage of program development. But there are also object-oriented methodologies for analysis and design. The question in this stage of the software life cycle is, How can one discover or invent the overall structure of a program? As an example of a rather simple object-oriented approach to analysis and design, consider this advice: Write down a description of the problem. Underline all the nouns in that description. The nouns should be considered as candidates for becoming classes or objects in the program design. Similarly, underline all the verbs. These are candidates for methods. This is your starting point. Further analysis might uncover the need for more classes and methods, and it might reveal that subclassing can be used to take advantage of similarities among classes.

This is perhaps a bit simple-minded, but the idea is clear and the general approach can be effective: Analyze the problem to discover the concepts that are involved, and create classes to represent those concepts. The design should arise from the problem itself, and you should end up with a program whose structure reflects the structure of the problem in a natural way.

Programming Example: Card, Hand, Deck

In this section, we look at some specific examples of object-oriented design in a domain that is simple enough that we have a chance of coming up with something reasonably reusable. Consider card games that are played with a standard deck of playing cards (a so-called "poker" deck, since it is used in the game of poker).

5.4.1 Designing the classes

In a typical card game, each player gets a hand of cards. The deck is shuffled and cards are dealt one at a time from the deck and added to the players' hands. In some games, cards can be removed from a hand, and new cards can be added. The game is won or lost depending on the value (ace, 2, ..., king) and suit (spades, diamonds, clubs, hearts) of the cards that a player receives. If we look for nouns in this description, there are several candidates for objects: game, player, hand, card, deck, value, and suit. Of these, the value and the suit of a card are simple values, and they might just be represented as instance variables in a *Card* object. In a complete program, the other five nouns might be represented by classes. But let's work on the ones that are most obviously reusable: *card*, *hand*, and *deck*.

If we look for verbs in the description of a card game, we see that we can *shuffle* a deck and *deal* a card from a deck. This gives use us two candidates for instance methods in a *Deck* class: `shuffle()` and `dealCard()`. Cards can be added to and removed from hands. This gives two candidates for instance methods in a *Hand* class: `addCard()` and `removeCard()`. Cards are relatively passive things, but we at least need to be able to determine their suits and values. We will discover more instance methods as we go along.

First, we'll design the deck class in detail. When a deck of cards is first created, it contains 52 cards in some standard order. The *Deck* class will need a constructor to create a new deck. The constructor needs no parameters because any new deck is the same as any other. There will be an instance method called `shuffle()` that will rearrange the 52 cards into a random order. The `dealCard()` instance method will get the next card from the deck. This will be a function with a return type of `Card`, since the caller needs to know what card is being dealt. It has no parameters -- when you deal the next card from the deck, you don't provide any information to the deck; you just get the next card, whatever it is. What will happen if there are no more cards in the deck when its `dealCard()` method is called? It should probably be considered an error to try to deal a card from an empty deck, so the deck can throw an exception in that case. But this raises another question: How will the rest of the program know whether the deck is empty? Of course, the program could keep track of how many cards it has used. But the deck itself should know how many cards it has left, so the program should just be able to ask the deck object. We can make this possible by specifying another instance method, `cardsLeft()`, that returns the number of cards remaining in the deck. This leads to a full specification of all the subroutines in the *Deck* class:

Constructor and instance methods in class Deck:

```
/**
 * Constructor.  Create an unshuffled deck of cards.
 */
public Deck()

/**
 * Put all the used cards back into the deck,
 * and shuffle it into a random order.
 */
public void shuffle()
```

```
/**
 * As cards are dealt from the deck, the number of
 * cards left decreases.  This function returns the
 * number of cards that are still left in the deck.
 */
public int cardsLeft()

/**
 * Deals one card from the deck and returns it.
 * @throws IllegalStateException if no more cards are left.
 */
public Card dealCard()
```

This is everything you need to know in order to **use** the `Deck` class. Of course, it doesn't tell us how to write the class. This has been an exercise in design, not in coding. You can look at the source code, *Deck.java*, if you want. It should not be a surprise that the class includes an array of *Cards* as an instance variable, but there are a few things you might not understand at this point. Of course, you can use the class in your programs as a black box, without understanding the implementation.

We can do a similar analysis for the *Hand* class. When a hand object is first created, it has no cards in it. An `addCard()` instance method will add a card to the hand. This method needs a parameter of type *Card* to specify which card is being added. For the `removeCard()` method, a parameter is needed to specify which card to remove. But should we specify the card itself ("Remove the ace of spades"), or should we specify the card by its position in the hand ("Remove the third card in the hand")? Actually, we don't have to decide, since we can allow for both options. We'll have two `removeCard()` instance methods, one with a parameter of type *Card* specifying the card to be removed and one with a parameter of type int specifying the position of the card in the hand. (Remember that you can have two methods in a class with the same name, provided they have different numbers or types of parameters.) Since a hand can contain a variable number of cards, it's convenient to be able to ask a hand object how many cards it contains. So, we need an instance method `getCardCount()` that returns the number of cards in the hand. When I play cards, I like to arrange the cards in my hand so that cards of the same value are next to each other. Since this is a generally useful thing to be able to do, we can provide instance methods for sorting the cards in the hand. Here is a full specification for a reusable *Hand* class:

Constructor and instance methods in class Hand:

```
/**
 * Constructor. Create a Hand object that is initially empty.
 */
public Hand()

/**
 * Discard all cards from the hand, making the hand empty.
 */
public void clear()
```

```
/**
 * Add the card c to the hand.  c should be non-null.
 * @throws NullPointerException if c is null.
 */
public void addCard(Card c)

/**
 * If the specified card is in the hand, it is removed.
 */
public void removeCard(Card c)

/**
 * Remove the card in the specified position from the
 * hand.  Cards are numbered counting from zero.
 * @throws IllegalArgumentException if the specified
 *    position does not exist in the hand.
 */
public void removeCard(int position)

/**
 * Return the number of cards in the hand.
 */
public int getCardCount()

/**
 * Get the card from the hand in given position, where
 * positions are numbered starting from 0.
 * @throws IllegalArgumentException if the specified
 *    position does not exist in the hand.
 */
public Card getCard(int position)

/**
 * Sorts the cards in the hand so that cards of the same
 * suit are grouped together, and within a suit the cards
 * are sorted by value.
 */
public void sortBySuit()

/**
 * Sorts the cards in the hand so that cards are sorted into
 * order of increasing value.  Cards with the same value
 * are sorted by suit. Note that aces are considered
 * to have the lowest value.
 */
public void sortByValue()
```

Again, there are a few things in the implementation of the class that you won't understand at this point, but that doesn't stop you from using the class in your projects. The source code can be found in the file *Hand.java*

5.4.2 The Card Class

We will look at the design and implementation of a *Card* class in full detail. The class will have a constructor that specifies the value and suit of the card that is being created. There are four suits, which can be represented by the integers 0, 1, 2, and 3. It would be tough to remember which number represents which suit, so I've defined named constants in the *Card* class to represent the four possibilities. For example, `Card.SPADES` is a constant that represents the suit, "spades". (These constants are declared to be `public final static ints`. It might be better to use an enumerated type, but I will stick here to integer-valued constants.) The possible values of a card are the numbers 1, 2, ..., 13, with 1 standing for an ace, 11 for a jack, 12 for a queen, and 13 for a king. Again, I've defined some named constants to represent the values of aces and face cards. (When you read the *Card* class, you'll see that I've also added support for Jokers.)

A *Card* object can be constructed knowing the value and the suit of the card. For example, we can call the constructor with statements such as:

```
card1 = new Card( Card.ACE, Card.SPADES );  // Construct ace of
spades.
card2 = new Card( 10, Card.DIAMONDS );   // Construct 10 of
diamonds.
card3 = new Card( v, s );  // This is OK, as long as v and s
                           //               are integer
expressions.
```

A *Card* object needs instance variables to represent its value and suit. I've made these `private` so that they cannot be changed from outside the class, and I've provided getter methods `getSuit()` and `getValue()` so that it will be possible to discover the suit and value from outside the class. The instance variables are initialized in the constructor, and are never changed after that. In fact, I've declared the instance variables `suit` and `value` to be `final`, since they are never changed after they are initialized. An instance variable can be declared `final` provided it is either given an initial value in its declaration or is initialized in every constructor in the class. Since all its instance variables are `final`, a *Card* is an immutable object.

Finally, I've added a few convenience methods to the class to make it easier to print out cards in a human-readable form. For example, I want to be able to print out the suit of a card as the word "Diamonds", rather than as the meaningless code number 2, which is used in the class to represent diamonds. Since this is something that I'll probably have to do in many programs, it makes sense to include support for it in the class. So, I've provided instance methods `getSuitAsString()` and `getValueAsString()` to return string representations of the suit and value of a card. Finally, I've defined the instance method `toString()` to return a string with both the value and suit, such as "Queen of Hearts". Recall that this method will be used automatically whenever a *Card* needs to be converted into a *String*, such as when the card is concatenated onto a string with the + operator. Thus, the statement

```
System.out.println( "Your card is the " + card );
```

is equivalent to

```
System.out.println( "Your card is the " + card.toString() );
```

If the card is the queen of hearts, either of these will print out "Your card is the Queen of Hearts".

Here is the complete *Card* class. It is general enough to be highly reusable, so the work that went into designing, writing, and testing it pays off handsomely in the long run.

```
/**
 * An object of type Card represents a playing card from a
 * standard Poker deck, including Jokers.  The card has a suit,
which
 * can be spades, hearts, diamonds, clubs, or joker.  A spade,
heart,
 * diamond, or club has one of the 13 values: ace, 2, 3, 4, 5, 6,
7,
 * 8, 9, 10, jack, queen, or king.  Note that "ace" is considered
to be
 * the smallest value.  A joker can also have an associated value;
 * this value can be anything and can be used to keep track of
several
 * different jokers.
 */
public class Card {

   public final static int SPADES = 0;   // Codes for the 4 suits,
plus Joker.
   public final static int HEARTS = 1;
   public final static int DIAMONDS = 2;
   public final static int CLUBS = 3;
   public final static int JOKER = 4;

   public final static int ACE = 1;      // Codes for the non-
numeric cards.
   public final static int JACK = 11;    //   Cards 2 through 10
have their
   public final static int QUEEN = 12;   //   numerical values for
their codes.
   public final static int KING = 13;

   /**
    * This card's suit, one of the constants SPADES, HEARTS,
DIAMONDS,
    * CLUBS, or JOKER.  The suit cannot be changed after the card
is
    * constructed.
    */
   private final int suit;

   /**
    * The card's value.  For a normal card, this is one of the
values
```

```
    * 1 through 13, with 1 representing ACE.  For a JOKER, the
value
    * can be anything.  The value cannot be changed after the card
    * is constructed.
    */
   private final int value;

   /**
    * Creates a Joker, with 1 as the associated value.  (Note that
    * "new Card()" is equivalent to "new Card(1,Card.JOKER)".)
    */
   public Card() {
      suit = JOKER;
      value = 1;
   }

   /**
    * Creates a card with a specified suit and value.
    * @param theValue the value of the new card.  For a regular
card (non-joker),
    * the value must be in the range 1 through 13, with 1
representing an Ace.
    * You can use the constants Card.ACE, Card.JACK, Card.QUEEN,
and Card.KING.
    * For a Joker, the value can be anything.
    * @param theSuit the suit of the new card.  This must be one of
the values
    * Card.SPADES, Card.HEARTS, Card.DIAMONDS, Card.CLUBS, or
Card.JOKER.
    * @throws IllegalArgumentException if the parameter values are
not in the
    * permissible ranges
    */
   public Card(int theValue, int theSuit) {
      if (theSuit != SPADES && theSuit != HEARTS && theSuit !=
DIAMONDS &&
            theSuit != CLUBS && theSuit != JOKER)
         throw new IllegalArgumentException("Illegal playing card
suit");
      if (theSuit != JOKER && (theValue < 1 || theValue > 13))
         throw new IllegalArgumentException("Illegal playing card
value");
      value = theValue;
      suit = theSuit;
   }

   /**
    * Returns the suit of this card.
    * @returns the suit, which is one of the constants Card.SPADES,
    * Card.HEARTS, Card.DIAMONDS, Card.CLUBS, or Card.JOKER
    */
   public int getSuit() {
      return suit;
   }

   /**
    * Returns the value of this card.
```

```
    * @return the value, which is one of the numbers 1 through 13,
inclusive for
    * a regular card, and which can be any value for a Joker.
    */
   public int getValue() {
      return value;
   }

   /**
    * Returns a String representation of the card's suit.
    * @return one of the strings "Spades", "Hearts", "Diamonds",
"Clubs"
    * or "Joker".
    */
   public String getSuitAsString() {
      switch ( suit ) {
      case SPADES:   return "Spades";
      case HEARTS:   return "Hearts";
      case DIAMONDS: return "Diamonds";
      case CLUBS:    return "Clubs";
      default:       return "Joker";
      }
   }

   /**
    * Returns a String representation of the card's value.
    * @return for a regular card, one of the strings "Ace", "2",
    * "3", ..., "10", "Jack", "Queen", or "King".  For a Joker, the
    * string is always numerical.
    */
   public String getValueAsString() {
      if (suit == JOKER)
         return "" + value;
      else {
         switch ( value ) {
         case 1:   return "Ace";
         case 2:   return "2";
         case 3:   return "3";
         case 4:   return "4";
         case 5:   return "5";
         case 6:   return "6";
         case 7:   return "7";
         case 8:   return "8";
         case 9:   return "9";
         case 10:  return "10";
         case 11:  return "Jack";
         case 12:  return "Queen";
         default:  return "King";
         }
      }
   }

   /**
    * Returns a string representation of this card, including both
    * its suit and its value (except that for a Joker with value 1,
    * the return value is just "Joker").  Sample return values
    * are: "Queen of Hearts", "10 of Diamonds", "Ace of Spades",
```

```
     * "Joker", "Joker #2"
     */
    public String toString() {
       if (suit == JOKER) {
          if (value == 1)
             return "Joker";
          else
             return "Joker #" + value;
       }
       else
          return getValueAsString() + " of " + getSuitAsString();
    }


} // end class Card
```

5.4.3 Example: A Simple Card Game

I will finish this section by presenting a complete program that uses the *Card* and *Deck* classes. The program lets the user play a very simple card game called HighLow. A deck of cards is shuffled, and one card is dealt from the deck and shown to the user. The user predicts whether the next card from the deck will be higher or lower than the current card. If the user predicts correctly, then the next card from the deck becomes the current card, and the user makes another prediction. This continues until the user makes an incorrect prediction. The number of correct predictions is the user's score.

My program has a static method that plays one game of HighLow. This method has a return value that represents the user's score in the game. The `main()` routine lets the user play several games of HighLow. At the end, it reports the user's average score.

I won't go through the development of the algorithms used in this program, but I encourage you to read it carefully and make sure that you understand how it works. Note in particular that the subroutine that plays one game of HighLow returns the user's score in the game as its return value. This gets the score back to the main program, where it is needed. Here is the program:

```
/**
 * This program lets the user play HighLow, a simple card game
 * that is described in the output statements at the beginning of
 * the main() routine.  After the user plays several games,
 * the user's average score is reported.
 */
public class HighLow {


   public static void main(String[] args) {

      System.out.println("This program lets you play the simple
card game,");
      System.out.println("HighLow.  A card is dealt from a deck of
cards.");
```

```
      System.out.println("You have to predict whether the next card
will be");
      System.out.println("higher or lower.  Your score in the game
is the");
      System.out.println("number of correct predictions you make
before");
      System.out.println("you guess wrong.");
      System.out.println();

      int gamesPlayed = 0;     // Number of games user has played.
      int sumOfScores = 0;     // The sum of all the scores from
                               //      all the games played.
      double averageScore;     // Average score, computed by
dividing
                               //      sumOfScores by gamesPlayed.
      boolean playAgain;       // Record user's response when user
is
                               //   asked whether he wants to play
                               //   another game.

      do {
         int scoreThisGame;        // Score for one game.
         scoreThisGame = play();   // Play the game and get the
score.
         sumOfScores += scoreThisGame;
         gamesPlayed++;
         System.out.print("Play again? ");
         playAgain = TextIO.getlnBoolean();
      } while (playAgain);

      averageScore = ((double)sumOfScores) / gamesPlayed;

      System.out.println();
      System.out.println("You played " + gamesPlayed + " games.");
      System.out.printf("Your average score was %1.3f.\n",
averageScore);

   }  // end main()


   /**
    * Lets the user play one game of HighLow, and returns the
    * user's score in that game.  The score is the number of
    * correct guesses that the user makes.
    */
   private static int play() {

      Deck deck = new Deck();  // Get a new deck of cards, and
                               //   store a reference to it in
                               //   the variable, deck.

      Card currentCard;  // The current card, which the user sees.

      Card nextCard;   // The next card in the deck.  The user
tries
                       //    to predict whether this is higher or
lower
```

```
                        //    than the current card.

      int correctGuesses ;  // The number of correct predictions
the
                            //   user has made.  At the end of the
game,
                            //   this will be the user's score.

      char guess;   // The user's guess.  'H' if the user predicts
that
                    //   the next card will be higher, 'L' if the
user
                    //   predicts that it will be lower.

      deck.shuffle();  // Shuffle the deck into a random order
before
                        //    starting the game.

      correctGuesses = 0;
      currentCard = deck.dealCard();
      System.out.println("The first card is the " + currentCard);

      while (true) {  // Loop ends when user's prediction is wrong.

         /* Get the user's prediction, 'H' or 'L' (or 'h' or 'l').
*/

         System.out.print("Will the next card be higher (H) or
lower (L)?  ");
         do {
             guess = TextIO.getlnChar();
             guess = Character.toUpperCase(guess);
             if (guess != 'H' && guess != 'L')
                System.out.print("Please respond with H or L:  ");
         } while (guess != 'H' && guess != 'L');

         /* Get the next card and show it to the user. */

         nextCard = deck.dealCard();
         System.out.println("The next card is " + nextCard);

         /* Check the user's prediction. */

         if (nextCard.getValue() == currentCard.getValue()) {
            System.out.println("The value is the same as the
previous card.");
            System.out.println("You lose on ties.  Sorry!");
            break;  // End the game.
         }
         else if (nextCard.getValue() > currentCard.getValue()) {
            if (guess == 'H') {
                System.out.println("Your prediction was correct.");
                correctGuesses++;
            }
            else {
                System.out.println("Your prediction was
incorrect.");
```

```
                break;  // End the game.
            }
         }
         else {  // nextCard is lower
            if (guess == 'L') {
                System.out.println("Your prediction was correct.");
                correctGuesses++;
            }
            else {
                System.out.println("Your prediction was
incorrect.");
                break;  // End the game.
            }
         }

         /* To set up for the next iteration of the loop, the
nextCard
            becomes the currentCard, since the currentCard has to
be
            the card that the user sees, and the nextCard will be
            set to the next card in the deck after the user makes
            his prediction.  */

         currentCard = nextCard;
         System.out.println();
         System.out.println("The card is " + currentCard);

      } // end of while loop

      System.out.println();
      System.out.println("The game is over.");
      System.out.println("You made " + correctGuesses
                                          + " correct
predictions.");
      System.out.println();

      return correctGuesses;

   }  // end play()


} // end class HighLow
```

Inheritance, Polymorphism, and Abstract Classes

A CLASS REPRESENTS A SET OF OBJECTS which share the same structure and behaviors. The class determines the structure of objects by specifying variables that are contained in each instance of the class, and it determines behavior by providing the instance methods that express the behavior of the objects. This is a powerful idea. However, something like this can be done in most programming languages. The central new idea in object-oriented programming -- the idea that really distinguishes it from traditional programming -- is to allow classes to express the

similarities among objects that share **some**, but not all, of their structure and behavior. Such similarities can be expressed using inheritance and polymorphism.

5.5.1 Extending Existing Classes

The topics covered in later subsections of this section are relatively advanced aspects of object-oriented programming. Any programmer should know what is meant by subclass, inheritance, and polymorphism. However, it will probably be a while before you actually do anything with inheritance except for extending classes that already exist. In the first part of this section, we look at how that is done.

In day-to-day programming, especially for programmers who are just beginning to work with objects, subclassing is used mainly in one situation: There is an existing class that can be adapted with a few changes or additions. This is much more common than designing groups of classes and subclasses from scratch. The existing class can be extended to make a subclass. The syntax for this is

```
public class subclass-name extends existing-class-name {
    .
    .   // Changes and additions.
    .
}
```

As an example, suppose you want to write a program that plays the card game, Blackjack. You can use the *Card*, *Hand*, and *Deck* classes developed in Section 5.4. However, a hand in the game of Blackjack is a little different from a hand of cards in general, since it must be possible to compute the "value" of a Blackjack hand according to the rules of the game. The rules are as follows: The value of a hand is obtained by adding up the values of the cards in the hand. The value of a numeric card such as a three or a ten is its numerical value. The value of a Jack, Queen, or King is 10. The value of an Ace can be either 1 or 11. An Ace should be counted as 11 unless doing so would put the total value of the hand over 21. Note that this means that the second, third, or fourth Ace in the hand will always be counted as 1.

One way to handle this is to extend the existing *Hand* class by adding a method that computes the Blackjack value of the hand. Here's the definition of such a class:

```
public class BlackjackHand extends Hand {

    /**
     * Computes and returns the value of this hand in the game
     * of Blackjack.
     */
    public int getBlackjackValue() {

        int val;      // The value computed for the hand.
        boolean ace;  // This will be set to true if the
                      //   hand contains an ace.
```

```
            int cards;    // Number of cards in the hand.

            val = 0;
            ace = false;
            cards = getCardCount();  // (method defined in class Hand.)

            for ( int i = 0;  i < cards;  i++ ) {
                    // Add the value of the i-th card in the hand.
                Card card;    // The i-th card;
                int cardVal;  // The blackjack value of the i-th card.
                card = getCard(i);
                cardVal = card.getValue();  // The normal value, 1 to
13.
                if (cardVal > 10) {
                    cardVal = 10;   // For a Jack, Queen, or King.
                }
                if (cardVal == 1) {
                    ace = true;     // There is at least one ace.
                }
                val = val + cardVal;
             }

             // Now, val is the value of the hand, counting any ace as
1.
             // If there is an ace, and if changing its value from 1 to
             // 11 would leave the score less than or equal to 21,
             // then do so by adding the extra 10 points to val.

             if ( ace == true  &&  val + 10 <= 21 )
                 val = val + 10;

             return val;

        }  // end getBlackjackValue()

    } // end class BlackjackHand
```

Since *BlackjackHand* is a subclass of *Hand*, an object of type *BlackjackHand* contains all the instance variables and instance methods defined in *Hand*, plus the new instance method named `getBlackjackValue()`. For example, if `bjh` is a variable of type `BlackjackHand`, then the following are all legal: `bjh.getCardCount()`, `bjh.removeCard(0)`, and `bjh.getBlackjackValue()`. The first two methods are defined in *Hand*, but are inherited by *BlackjackHand*.

Variables and methods from the *Hand* class are inherited by *BlackjackHand*, and they can be used in the definition of *BlackjackHand* just as if they were actually defined in that class -- except for any that are declared to be `private`, which prevents access even by subclasses. The statement "`cards = getCardCount();`" in the above definition of `getBlackjackValue()` calls the instance method `getCardCount()`, which was defined in *Hand*.

Extending existing classes is an easy way to build on previous work. We'll see that many standard classes have been written specifically to be used as the basis for making subclasses.

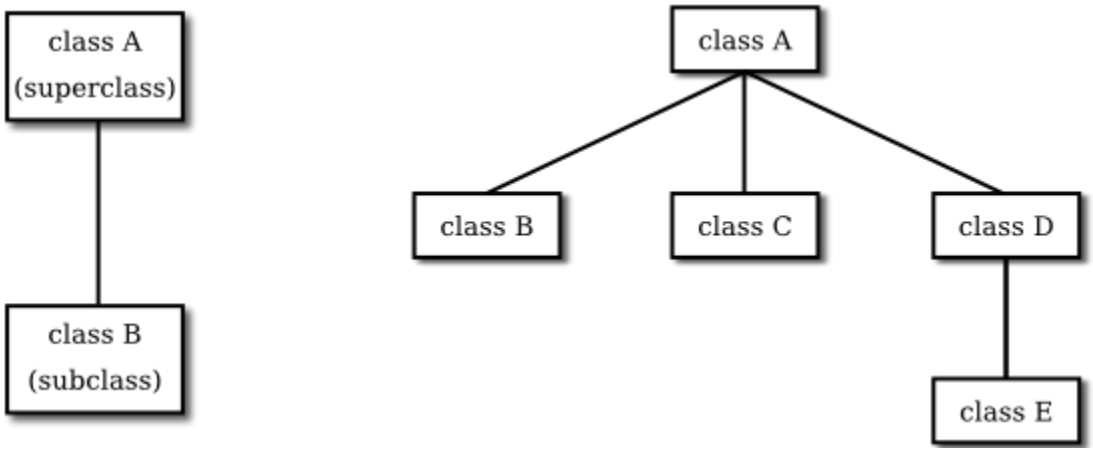
Access modifiers such as `public` and `private` are used to control access to members of a class. There is one more access modifier, protected, that comes into the picture when subclasses are taken into consideration. When `protected` is applied as an access modifier to a method or member variable in a class, that member can be used in subclasses -- direct or indirect -- of the class in which it is defined, but it cannot be used in non-subclasses. (There is an exception: A `protected` member can also be accessed by any class in the same package as the class that contains the protected member. Recall that using no access modifier makes a member accessible to classes in the same package, and nowhere else. Using the `protected` modifier is strictly more liberal than using no modifier at all: It allows access from classes in the same package and from **subclasses** that are not in the same package.)

When you declare a method or member variable to be `protected`, you are saying that it is part of the implementation of the class, rather than part of the public interface of the class. However, you are allowing subclasses to use and modify that part of the implementation.

For example, consider a *PairOfDice* class that has instance variables `die1` and `die2` to represent the numbers appearing on the two dice. We could make those variables `private` to make it impossible to change their values from outside the class, while still allowing read access through getter methods. However, if we think it possible that *PairOfDice* will be used to create subclasses, we might want to make it possible for subclasses to change the numbers on the dice. For example, a *GraphicalDice* subclass that draws the dice might want to change the numbers at other times besides when the dice are rolled. In that case, we could make `die1` and `die2` `protected`, which would allow the subclass to change their values without making them public to the rest of the world. (An even better idea would be to define `protected` setter methods for the variables. A setter method could, for example, ensure that the value that is being assigned to the variable is in the legal range 1 through 6.)

5.5.2 Inheritance and Class Hierarchy

The term inheritance refers to the fact that one class can inherit part or all of its structure and behavior from another class. The class that does the inheriting is said to be a subclass of the class from which it inherits. If class B is a subclass of class A, we also say that class A is a superclass of class B. (Sometimes the terms derived class and base class are used instead of subclass and superclass; this is the common terminology in C++.) A subclass can add to the structure and behavior that it inherits. It can also replace or modify inherited behavior (though not inherited structure). The relationship between subclass and superclass is sometimes shown by a diagram in which the subclass is shown below, and connected to, its superclass, as shown on the left below:



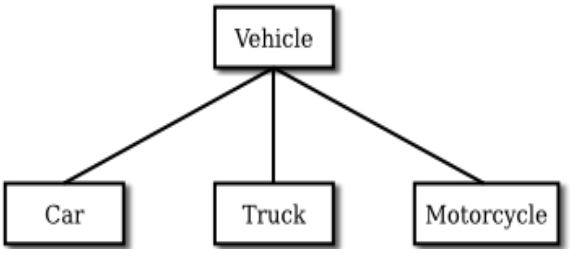
In Java, to create a class named "B" as a subclass of a class named "A", you would write

```
class B extends A {
    .
    .  // additions to, and modifications of,
    .  // stuff inherited from class A
    .
}
```

Several classes can be declared as subclasses of the same superclass. The subclasses, which might be referred to as "sibling classes," share some structures and behaviors -- namely, the ones they inherit from their common superclass. The superclass expresses these shared structures and behaviors. In the diagram shown on the right above, classes B, C, and D are sibling classes. Inheritance can also extend over several "generations" of classes. This is shown in the diagram, where class E is a subclass of class D which is itself a subclass of class A. In this case, class E is considered to be a subclass of class A, even though it is not a direct subclass. This whole set of classes forms a small class hierarchy.

5.5.3 Example: Vehicles

Let's look at an example. Suppose that a program has to deal with motor vehicles, including cars, trucks, and motorcycles. (This might be a program used by a Department of Motor Vehicles to keep track of registrations.) The program could use a class named *Vehicle* to represent all types of vehicles. Since cars, trucks, and motorcycles are types of vehicles, they would be represented by subclasses of the *Vehicle* class, as shown in this class hierarchy diagram:



The *Vehicle* class would include instance variables such as `registrationNumber` and `owner` and instance methods such as `transferOwnership()`. These are variables and methods common to all vehicles. The three subclasses of *Vehicle* -- *Car*, *Truck*, and *Motorcycle* -- could then be used to hold variables and methods specific to particular types of vehicles. The *Car* class might add an instance variable `numberOfDoors`, the *Truck* class might have `numberOfAxles`, and the *Motorcycle* class could have a boolean variable `hasSidecar`. (Well, it could in theory at least, even if it might give a chuckle to the people at the Department of Motor Vehicles.) The declarations of these classes in a Java program would look, in outline, like this (although they are likely to be defined in separate files and declared as `public` classes):

```
class Vehicle {
   int registrationNumber;
   Person owner;  // (Assuming that a Person class has been
defined!)
   void transferOwnership(Person newOwner) {
       . . .
   }
   . . .
}

class Car extends Vehicle {
   int numberOfDoors;
   . . .
}

class Truck extends Vehicle {
   int numberOfAxles;
   . . .
}

class Motorcycle extends Vehicle {
   boolean hasSidecar;
   . . .
}
```

Suppose that `myCar` is a variable of type *Car* that has been declared and initialized with the statement

```
Car myCar = new Car();
```

Given this declaration, a program could refer to `myCar.numberOfDoors`, since `numberOfDoors` is an instance variable in the class `Car`. But since class *Car* extends class

Vehicle, a car also has all the structure and behavior of a vehicle. This means that `myCar.registrationNumber`, `myCar.owner`, and `myCar.transferOwnership()` also exist.

Now, in the real world, cars, trucks, and motorcycles are in fact vehicles. The same is true in a program. That is, an object of type *Car* or *Truck* or *Motorcycle* is automatically an object of type *Vehicle* too. This brings us to the following Important Fact:

A variable that can hold a reference
to an object of class A can also hold a reference
to an object belonging to any subclass of A.

The practical effect of this in our example is that an object of type *Car* can be assigned to a variable of type *Vehicle*. That is, it would be legal to say

```
Vehicle myVehicle = myCar;
```

or even

```
Vehicle myVehicle = new Car();
```

After either of these statements, the variable `myVehicle` holds a reference to a *Vehicle* object that happens to be an instance of the subclass, *Car*. The object "remembers" that it is in fact a *Car*, and not **just** a *Vehicle*. Information about the actual class of an object is stored as part of that object. It is even possible to test whether a given object belongs to a given class, using the `instanceof` operator. The test:

```
if (myVehicle instanceof Car) ...
```

determines whether the object referred to by `myVehicle` is in fact a car.

On the other hand, the assignment statement

```
myCar = myVehicle;
```

would be illegal because `myVehicle` could potentially refer to other types of vehicles that are not cars. This is similar to a problem we saw previously in Subsection 2.5.6: The computer will not allow you to assign an int value to a variable of type short, because not every int is a short. Similarly, it will not allow you to assign a value of type *Vehicle* to a variable of type *Car* because not every vehicle is a car. As in the case of ints and shorts, the solution here is to use type-casting. If, for some reason, you happen to know that `myVehicle` does in fact refer to a *Car*, you can use the type cast `(Car)myVehicle` to tell the computer to treat `myVehicle` as if it were actually of type *Car*. So, you could say

```
myCar = (Car)myVehicle;
```

and you could even refer to `((Car)myVehicle).numberOfDoors`. (The parentheses are necessary because of precedence. The "`.`" has higher precedence than the type-cast, so `(Car)myVehicle.numberOfDoors` would be read as `(Car)(myVehicle.numberOfDoors)`, an attempt to type-cast the int `myVehicle.numberOfDoors` into a *Vehicle*, which is impossible.)

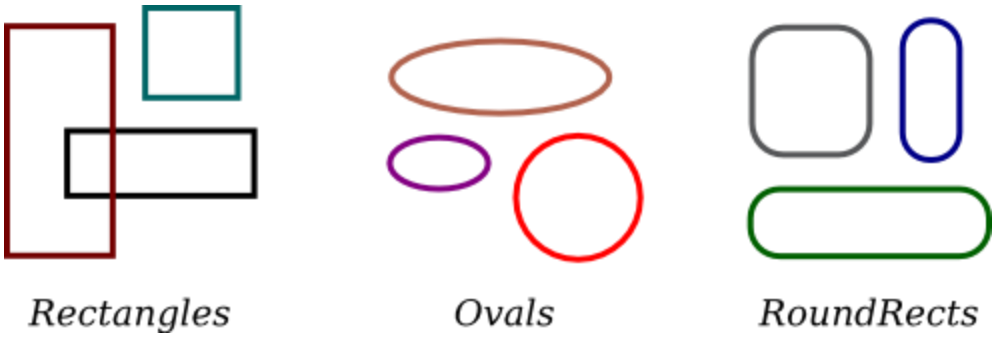
As an example of how this could be used in a program, suppose that you want to print out relevant data about the *Vehicle* referred to by `myVehicle`. If it's a *car*, you will want to print out the car's `numberOfDoors`, but you can't say `myVehicle.numberOfDoors`, since there is no `numberOfDoors` in the *Vehicle* class. But you could say:

```
System.out.println("Vehicle Data:");
System.out.println("Registration number:  "
                              + myVehicle.registrationNumber);
if (myVehicle instanceof Car) {
   System.out.println("Type of vehicle:  Car");
   Car c;
   c = (Car)myVehicle;  // Type-cast to get access to
numberOfDoors!
   System.out.println("Number of doors:  " + c.numberOfDoors);
}
else if (myVehicle instanceof Truck) {
   System.out.println("Type of vehicle:  Truck");
   Truck t;
   t = (Truck)myVehicle;  // Type-cast to get access to
numberOfAxles!
   System.out.println("Number of axles:  " + t.numberOfAxles);
}
else if (myVehicle instanceof Motorcycle) {
   System.out.println("Type of vehicle:  Motorcycle");
   Motorcycle m;
   m = (Motorcycle)myVehicle;  // Type-cast to get access to
hasSidecar!
   System.out.println("Has a sidecar:    " + m.hasSidecar);
}
```

Note that for object types, when the computer executes a program, it checks whether type-casts are valid. So, for example, if `myVehicle` refers to an object of type *Truck*, then the type cast `(Car)myVehicle` would be an error. When this happens, an exception of type *ClassCastException* is thrown. This check is done at run time, not compile time, because the actual type of the object referred to by `myVehicle` is not known when the program is compiled.

5.5.4 Polymorphism

As another example, consider a program that deals with shapes drawn on the screen. Let's say that the shapes include rectangles, ovals, and roundrects of various colors. (A "roundrect" is just a rectangle with rounded corners.)



Three classes, *Rectangle*, *Oval*, and *RoundRect*, could be used to represent the three types of shapes. These three classes would have a common superclass, *Shape*, to represent features that all three shapes have in common. The *Shape* class could include instance variables to represent the color, position, and size of a shape, and it could include instance methods for changing the values of those properties. Changing the color, for example, might involve changing the value of an instance variable, and then redrawing the shape in its new color:

```
class Shape {

    Color color; // (must be imported from package java.awt)

    void setColor(Color newColor) {
          // Method to change the color of the shape.
       color = newColor; // change value of instance variable
       redraw(); // redraw shape, which will appear in new color
    }

    void redraw() {
          // method for drawing the shape
       ? ? ?  // what commands should go here?
    }

    . . .          // more instance variables and methods

} // end of class Shape
```

Now, you might see a problem here with the method `redraw()`. The problem is that each different type of shape is drawn differently. The method `setColor()` can be called for any type of shape. How does the computer know which shape to draw when it executes the `redraw()`? Informally, we can answer the question like this: The computer executes `redraw()` by asking the shape to redraw **itself**. Every shape object knows what it has to do to redraw itself.

In practice, this means that each of the specific shape classes has its own `redraw()` method:

```
class Rectangle extends Shape {
   void redraw() {
      . . .  // commands for drawing a rectangle
   }
   . . . // possibly, more methods and variables
```

```
}

class Oval extends Shape {
   void redraw() {
      . . .  // commands for drawing an oval
   }
   . . . // possibly, more methods and variables
}

class RoundRect extends Shape {
   void redraw() {
      . . .  // commands for drawing a rounded rectangle
   }
   . . . // possibly, more methods and variables
}
```

Suppose that `someShape` is a variable of type *Shape*. Then it could refer to an object of any of the types *Rectangle*, *Oval*, or *RoundRect*. As a program executes, and the value of `someShape` changes, it could even refer to objects of different types at different times! Whenever the statement

```
someShape.redraw();
```

is executed, the redraw method that is actually called is the one appropriate for the type of object to which `someShape` actually refers. There may be no way of telling, from looking at the text of the program, what shape this statement will draw, since it depends on the value that `someShape` happens to have when the program is executed. Even more is true. Suppose the statement is in a loop and gets executed many times. If the value of `someShape` changes as the loop is executed, it is possible that the very same statement "`someShape.redraw();`" will call different methods and draw different shapes as it is executed over and over. We say that the `redraw()` method is polymorphic. A method is polymorphic if the action performed by the method depends on the actual type of the object to which the method is applied. Polymorphism is one of the major distinguishing features of object-oriented programming. This can be seen most vividly, perhaps, if we have an array of shapes. Suppose that `shapelist` is an a variable of type `Shape[]`, and that the array has already been created and filled with data. Some of the elements in the array might be *Rectangles*, some might be *Ovals*, and some might be *RoundRects*. We can draw all the shapes in the array by saying

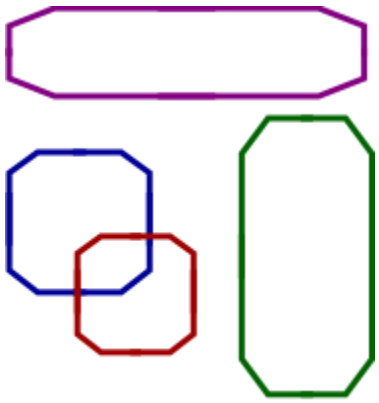
```
for (int i = 0; i < shapelist.length; i++ ) {
    Shape shape = shapelist[i];
    shape.redraw();
}
```

As the computer goes through this loop, the statement `shape.redraw()` will sometimes draw a rectangle, sometimes an oval, and sometimes a roundrect, depending on the type of object to which array element number `i` refers.

Perhaps this becomes more understandable if we change our terminology a bit: In object-oriented programming, calling a method is often referred to as sending a message to an object. The object

responds to the message by executing the appropriate method. The statement "`someShape.redraw();`" is a message to the object referred to by `someShape`. Since that object knows what type of object it is, it knows how it should respond to the message. From this point of view, the computer always executes "`someShape.redraw();`" in the same way: by sending a message. The response to the message depends, naturally, on who receives it. From this point of view, objects are active entities that send and receive messages, and polymorphism is a natural, even necessary, part of this view. Polymorphism just means that different objects can respond to the same message in different ways.

One of the most beautiful things about polymorphism is that it lets code that you write do things that you didn't even conceive of, at the time you wrote it. Suppose that I decide to add beveled rectangles to the types of shapes my program can deal with. A beveled rectangle has a triangle cut off each corner:



BeveledRects

To implement beveled rectangles, I can write a new subclass, *BeveledRect*, of class *Shape* and give it its own `redraw()` method. Automatically, code that I wrote previously -- such as the statement `someShape.redraw()` -- can now suddenly start drawing beveled rectangles, even though the beveled rectangle class didn't exist when I wrote the statement!

In the statement "`someShape.redraw();`", the `redraw` message is sent to the object `someShape`. Look back at the method in the *Shape* class for changing the color of a shape:

```
void setColor(Color newColor) {
   color = newColor; // change value of instance variable
   redraw(); // redraw shape, which will appear in new color
}
```

A `redraw` message is sent here, but which object is it sent to? Well, the `setColor` method is itself a message that was sent to some object. The answer is that the `redraw` message is sent to that **same object**, the one that received the `setColor` message. If that object is a rectangle, then it contains a `redraw()` method for drawing rectangles, and that is the one that is executed.

If the object is an oval, then it is the `redraw()` method from the *Oval* class. This is what you should expect, but it means that the "`redraw();`" statement in the `setColor()` method does **not** necessarily call the `redraw()` method in the *Shape* class! The `redraw()` method that is executed could be in any subclass of *Shape*. This is just another case of polymorphism.

5.5.5 Abstract Classes

Whenever a *Rectangle*, *Oval*, or *RoundRect* object has to draw itself, it is the `redraw()` method in the appropriate class that is executed. This leaves open the question, What does the `redraw()` method in the *Shape* class do? How should it be defined?

The answer may be surprising: We should leave it blank! The fact is that the class *Shape* represents the abstract idea of a shape, and there is no way to draw such a thing. Only particular, concrete shapes like rectangles and ovals can be drawn. So, why should there even be a `redraw()` method in the *Shape* class? Well, it has to be there, or it would be illegal to call it in the `setColor()` method of the *Shape* class, and it would be illegal to write "`someShape.redraw();`". The compiler would complain that `someShape` is a variable of type *Shape* and there's no `redraw()` method in the *Shape* class.

Nevertheless the version of `redraw()` in the *Shape* class itself will never actually be called. In fact, if you think about it, there can never be any reason to construct an actual object of type *Shape*! You can have **variables** of type *Shape*, but the objects they refer to will always belong to one of the subclasses of *Shape*. We say that *Shape* is an abstract class. An abstract class is one that is not used to construct objects, but only as a basis for making subclasses. An abstract class exists **only** to express the common properties of all its subclasses. A class that is not abstract is said to be concrete. You can create objects belonging to a concrete class, but not to an abstract class. A variable whose type is given by an abstract class can only refer to objects that belong to concrete subclasses of the abstract class.

Similarly, we say that the `redraw()` method in class *Shape* is an abstract method, since it is never meant to be called. In fact, there is nothing for it to do -- any actual redrawing is done by `redraw()` methods in the subclasses of *Shape*. The `redraw()` method in *Shape* has to be there. But it is there only to tell the computer that **all** `Shapes` understand the `redraw` message. As an abstract method, it exists merely to specify the common interface of all the actual, concrete versions of `redraw()` in the subclasses. There is no reason for the abstract `redraw()` in class *Shape* to contain any code at all.

Shape and its `redraw()` method are semantically abstract. You can also tell the computer, syntactically, that they are abstract by adding the modifier "`abstract`" to their definitions. For an abstract method, the block of code that gives the implementation of an ordinary method is replaced by a semicolon. An implementation must then be provided for the abstract method in any concrete subclass of the abstract class. Here's what the *Shape* class would look like as an abstract class:

```
public abstract class Shape {

    Color color;   // color of shape.

    void setColor(Color newColor) {
          // method to change the color of the shape
       color = newColor; // change value of instance variable
       redraw(); // redraw shape, which will appear in new color
    }

    abstract void redraw();
          // abstract method -- must be defined in
          // concrete subclasses

    . . .  // more instance variables and methods

} // end of class Shape
```

Once you have declared the class to be `abstract`, it becomes illegal to try to create actual objects of type *Shape*, and the computer will report a syntax error if you try to do so.

Note, by the way, that the *Vehicle* class discussed above would probably also be an abstract class. There is no way to own a vehicle as such -- the actual vehicle has to be a car or a truck or a motorcycle, or some other "concrete" type of vehicle.

Recall from Subsection 5.3.2 that a class that is not explicitly declared to be a subclass of some other class is automatically made a subclass of the standard class *Object*. That is, a class declaration with no "`extends`" part such as

```
public class myClass { . . .
```

is exactly equivalent to

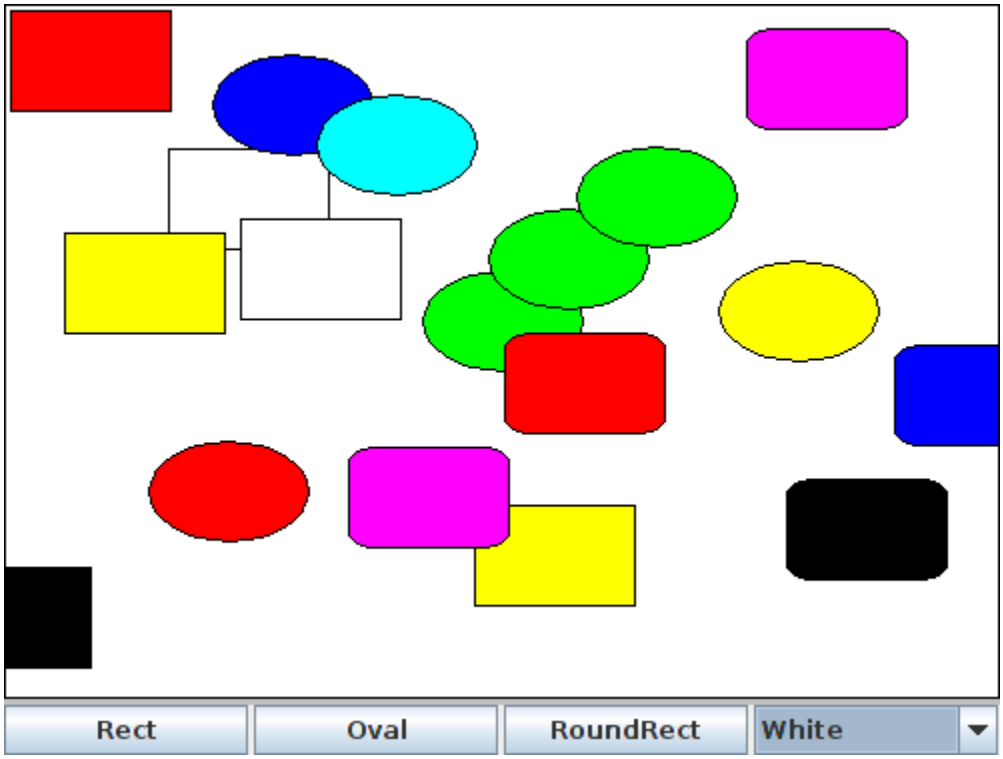
```
public class myClass extends Object { . . .
```

This means that class *Object* is at the top of a huge class hierarchy that includes every other class. (Semantically, *Object* is an abstract class, in fact the most abstract class of all. Curiously, however, it is not declared to be `abstract` syntactically, which means that you can create objects of type *Object*. However, there is not much that you can do with them.)

Since every class is a subclass of *Object*, a variable of type *Object* can refer to any object whatsoever, of any type. Similarly, an array of type `Object[]` can hold objects of any type.

The sample source code file *ShapeDraw.java* uses an abstract *Shape* class and an array of type `Shape[]` to hold a list of shapes. You might want to look at this file, even though you won't be

able to understand all of it at this time. Even the definitions of the shape classes are somewhat different from those that I have described in this section. (For example, the `draw()` method has a parameter of type *Graphics*. This parameter is required because drawing in Java requires a graphics context.) I'll return to similar examples in later chapters when you know more about GUI programming. However, it would still be worthwhile to look at the definition of the *Shape* class and its subclasses in the source code. You might also check how an array is used to hold the list of shapes. Here is a screenshot from the program:



If you run the `ShapeDraw` program, you can click one of the buttons along the bottom to add a shape to the picture. The new shape will appear in the upper left corner of the drawing area. The color of the shape is given by the "pop-up menu" in the lower right. Once a shape is on the screen, you can drag it around with the mouse. A shape will maintain the same front-to-back order with respect to other shapes on the screen, even while you are dragging it. However, you can move a shape out in front of all the other shapes if you hold down the shift key as you click on it.

In the program, the only time when the actual class of a shape is used is when that shape is added to the screen. Once the shape has been created, it is manipulated entirely as an abstract shape. The routine that implements dragging, for example, works with variables of type *Shape* and makes no reference to any of its subclasses. As the shape is being dragged, the dragging routine just calls the shape's draw method each time the shape has to be drawn, so it doesn't have to know how to draw the shape or even what type of shape it is. The object is responsible for drawing itself. If I wanted to add a new type of shape to the program, I would define a new subclass of *Shape*, add another button, and program the button to add the correct type of shape to the screen. No other changes in the programming would be necessary.

this and super

ALTHOUGH THE BASIC IDEAS of object-oriented programming are reasonably simple and clear, they are subtle, and they take time to get used to. And unfortunately, beyond the basic ideas there are a lot of details. The rest of this chapter covers more of those annoying details. Remember that you don't need to master everything in this chapter the first time through. In this section, we'll look at two variables, `this` and `super`, that are automatically defined in any instance method.

5.6.1 The Special Variable this

What does it mean when you use a simple identifier such as `amount` or `process()` to refer to a variable or method? The answer depends on scope rules that tell where and how each declared variable and method can be accessed in a program. Inside the definition of a method, a simple variable name might refer to a local variable or parameter, if there is one "in scope," that is, one whose declaration is in effect at the point in the source code where the reference occurs. If not, it must refer to a member variable of the class in which the reference occurs. Similarly, a simple method name must refer to a method in the same class.

A **static** member of a class has a simple name that can only be used inside the class definition; for use outside the class, it has a full name of the form **class-name.simple-name**. For example, "`Math.PI`" is a static member variable with simple name "`PI`" in the class "*Math*". It's always legal to use the full name of a static member, even within the class where it's defined. Sometimes it's even necessary, as when the simple name of a static member variable is hidden by a local variable or parameter of the same name.

Instance variables and instance methods also have simple names. The simple name of such an instance member can be used in instance methods in the class where the instance member is defined (but not in static methods). Instance members also have full names -- but remember that an instance variable or instance method is actually contained in an object rather than in a class, and each object has its own version. A full name of an instance member starts with a reference to the object that contains the instance member. For example, if `std` is a variable that refers to an object of type *Student*, then `std.test1` could be a full name for an instance variable named `test1` that is contained in that object.

But when we are working inside a class and use a simple name to refer to an instance variable like `test1`, where is the object that contains the variable? The solution to this riddle is simple: Suppose that a reference to "`test1`" occurs in the definition of some instance method. The actual method that gets executed is part of some particular object of type *Student*. When that method gets executed, the occurrence of the name "`test1`" refers to the `test1` variable **in that same object**. (This is why simple names of instance members cannot be used in static methods --

when a static method is executed, it is not part of an object, and hence there are no instance members in sight!)

This leaves open the question of full names for instance members inside the same class where they are defined. We need a way to refer to "the object that contains this method." Java defines a special variable named this for just this purpose. The variable `this` can be used in the source code of an instance method to refer to the object that contains the method. This intent of the name, "`this`," is to refer to "this object," the one right here that this very method is in. If `var` is an instance variable in the same object as the method, then "`this.var`" is a full name for that variable. If `otherMethod()` is an instance method in the same object, then `this.otherMethod()` could be used to call that method. Whenever the computer executes an instance method, it automatically sets the variable `this` to refer to the object that contains the method.

(Some object oriented languages use the name "self" instead of "this." Here, an object is seen as an entity that receives messages and responds by performing some action. From the point of view of that entity, an instance variable such as `self.name` refers to the entity's own `name`, something that is part of the entity itself. Calling an instance method such as `self.redraw()` is like saying "message to self: redraw!")

One common use of `this` is in constructors. For example:

```
public class Student {

    private String name;  // Name of the student.

    public Student(String name) {
         // Constructor.  Create a student with specified name.
       this.name = name;
    }
      .
      .   // More variables and methods.
      .
}
```

In the constructor, the instance variable called `name` is hidden by a formal parameter that is also called "name." However, the instance variable can still be referred to by its full name, which is `this.name`. In the assignment statement "`this.name = name`", the value of the formal parameter, `name`, is assigned to the instance variable, `this.name`. This is considered to be acceptable style: There is no need to dream up cute new names for formal parameters that are just used to initialize instance variables. You can use the same name for the parameter as for the instance variable.

There are other uses for `this`. Sometimes, when you are writing an instance method, you need to pass the object that contains the method to a subroutine, as an actual parameter. In that case, you can use `this` as the actual parameter. For example, if you wanted to print out a string representation of the object, you could say "`System.out.println(this);`". Or you could

assign the value of `this` to another variable in an assignment statement. You can store it in an array. In fact, you can do anything with `this` that you could do with any other variable, except change its value. (Consider it to be a `final` variable.)

5.6.2 The Special Variable super

Java also defines another special variable, named "`super`", for use in the definitions of instance methods. The variable `super` is for use in a subclass. Like `this`, `super` refers to the object that contains the method. But it's forgetful. It forgets that the object belongs to the class you are writing, and it remembers only that it belongs to the superclass of that class. The point is that the class can contain additions and modifications to the superclass. `super` doesn't know about any of those additions and modifications; it can only be used to refer to methods and variables in the superclass.

Let's say that the class that you are writing contains an instance method named `doSomething()`. Consider the subroutine call statement `super.doSomething()`. Now, `super` doesn't know anything about the `doSomething()` method in the subclass. It only knows about things in the superclass, so it tries to execute a method named `doSomething()` from the superclass. If there is none -- if the `doSomething()` method was an addition rather than a modification -- you'll get a syntax error.

The reason `super` exists is so you can get access to things in the superclass that are **hidden** by things in the subclass. For example, `super.var` always refers to an instance variable named `var` in the superclass. This can be useful for the following reason: If a class contains an instance variable with the same name as an instance variable in its superclass, then an object of that class will actually contain two variables with the same name: one defined as part of the class itself and one defined as part of the superclass. The variable in the subclass does not **replace** the variable of the same name in the superclass; it merely **hides** it. The variable from the superclass can still be accessed, using `super`.

When a subclass contains an instance method that has the same signature as a method in its superclass, the method from the superclass is hidden in the same way. We say that the method in the subclass overrides the method from the superclass. Again, however, `super` can be used to access the method from the superclass.

The major use of `super` is to override a method with a new method that **extends** the behavior of the inherited method, instead of **replacing** that behavior entirely. The new method can use `super` to call the method from the superclass, and then it can add additional code to provide additional behavior. As an example, suppose you have a *PairOfDice* class that includes a `roll()` method. Suppose that you want a subclass, *GraphicalDice*, to represent a pair of dice drawn on the computer screen. The `roll()` method in the *GraphicalDice* class should do everything that the `roll()` method in the *PairOfDice* class does. We can express this with a call to `super.roll()`, which calls the method in the superclass. But in addition to that, the

`roll()` method for a *GraphicalDice* object has to redraw the dice to show the new values. The *GraphicalDice* class might look something like this:

```
public class GraphicalDice extends PairOfDice {

    public void roll() {
            // Roll the dice, and redraw them.
         super.roll();  // Call the roll method from PairOfDice.
         redraw();      // Call a method to draw the dice.
    }
       .
       .  // More stuff, including definition of redraw().
       .
}
```

Note that this allows you to extend the behavior of the `roll()` method even if you don't know how the method is implemented in the superclass!

5.6.3 super and this As Constructors

Constructors are not inherited. That is, if you extend an existing class to make a subclass, the constructors in the superclass do `not` become part of the subclass. If you want constructors in the subclass, you have to define new ones from scratch. If you don't define any constructors in the subclass, then the computer will make up a default constructor, with no parameters, for you.

This could be a problem, if there is a constructor in the superclass that does a lot of necessary work. It looks like you might have to repeat all that work in the subclass! This could be a **real** problem if you don't have the source code to the superclass, and don't even know how it is implemented. It might look like an impossible problem, if the constructor in the superclass uses `private` member variables that you don't even have access to in the subclass!

Obviously, there has to be some fix for this, and there is. It involves the special variable, `super`. As the very first statement in a constructor, you can use `super` to call a constructor from the superclass. The notation for this is a bit ugly and misleading, and it can only be used in this one particular circumstance: It looks like you are calling `super` as a subroutine (even though `super` is not a subroutine and you can't call constructors the same way you call other subroutines anyway). As an example, assume that the *PairOfDice* class has a constructor that takes two integers as parameters. Consider a subclass:

```
public class GraphicalDice extends PairOfDice {

     public GraphicalDice() {  // Constructor for this class.

         super(3,4);  // Call the constructor from the
                      //   PairOfDice class, with parameters 3, 4.

         initializeGraphics();  // Do some initialization specific
```

```
                                        //   to the GraphicalDice class.
    }
        .
        .  // More constructors, methods, variables...
        .
}
```

The statement "`super(3,4);`" calls the constructor from the superclass. This call must be the first line of the constructor in the subclass. Note that if you don't explicitly call a constructor from the superclass in this way, then the default constructor from the superclass, the one with no parameters, will be called automatically. (And if no such constructor exists in the superclass, the compiler will consider it to be a syntax error.)

You can use the special variable `this` in exactly the same way to call another constructor in the same class. That is, the very first line of a constructor can look like a subroutine call with "this" as the name of the subroutine. The result is that the body of another constructor in the same class is executed. This can be very useful since it can save you from repeating the same code in several different constructors. As an example, consider *MosaicPanel.java*, which was used indirectly in Section 4.6. A *MosaicPanel* represents a grid of colored rectangles. It has a constructor with many parameters:

```
public MosaicPanel(int rows, int columns,
                int preferredBlockWidth, int preferredBlockHeight,
                Color borderColor, int borderWidth)
```

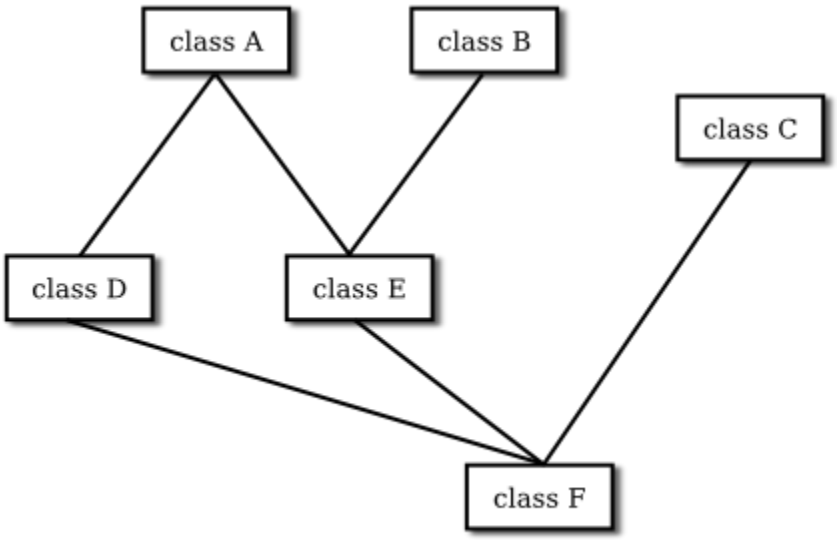
This constructor provides a lot of options and does a lot of initialization. I wanted to provide easier-to-use constructors with fewer options, but all the initialization still has to be done. The class also contains these constructors:

```
public MosaicPanel() {
    this(42,42,16,16);
}

public MosaicPanel(int rows, int columns) {
    this(rows,columns,16,16);
}

public MosaicPanel(int rows, int columns,
                   int preferredBlockWidth, int
preferredBlockHeight) {
    this(rows, columns, preferredBlockWidth, preferredBlockHeight,
null, 0);
}
```

Each of these constructors exists just to call another constructor, while providing constant values for some of the parameters. For example, `this(42,42,16,16)` calls the last constructor listed here, while that constructor in turn calls the main, six-parameter constructor. That main constructor is eventually called in all cases, so that all the essential initialization gets done in every case.

Interfaces

Some object-oriented programming languages, such as C++, allow a class to extend two or more superclasses. This is called multiple inheritance. In the illustration below, for example, class E is shown as having both class A and class B as direct superclasses, while class F has three direct superclasses.



Multiple inheritance (**NOT** allowed in Java)

Such multiple inheritance is **not** allowed in Java. The designers of Java wanted to keep the language reasonably simple, and felt that the benefits of multiple inheritance were not worth the cost in increased complexity. However, Java does have a feature that can be used to accomplish many of the same goals as multiple inheritance: interfaces.

5.7.1 Defining and Implementing Interfaces

We've encountered the term "interface" before, in connection with black boxes in general and subroutines in particular. The interface of a subroutine consists of the name of the subroutine, its return type, and the number and types of its parameters. This is the information you need to know if you want to call the subroutine. A subroutine also has an implementation: the block of code which defines it and which is executed when the subroutine is called.

In Java, `interface` is a reserved word with an additional, technical meaning. An "`interface`" in this sense consists of a set of instance method interfaces, without any associated implementations. (Actually, a Java interface can contain other things as well, as we'll

see later.) A class can implement an `interface` by providing an implementation for each of the methods specified by the interface. Here is an example of a very simple Java `interface`:

```
public interface Drawable {
   public void draw(Graphics g);
}
```

This looks much like a class definition, except that the implementation of the `draw()` method is omitted. A class that implements the `interface` *Drawable* must provide an implementation for this method. Of course, the class can also include other methods and variables. For example,

```
public class Line implements Drawable {
    public void draw(Graphics g) {
        . . . // do something -- presumably, draw a line
    }
    . . . // other methods and variables
}
```

Note that to implement an interface, a class must do more than simply provide an implementation for each method in the interface; it must also **state** that it implements the interface, using the reserved word `implements` as in this example: "`public class Line` **`implements`** `Drawable`". Any concrete class that implements the *Drawable* interface must defines a `draw()` instance method. Any object created from such a class includes a `draw()` method. We say that an **object** implements an `interface` if it belongs to a class that implements the interface. For example, any object of type *Line* implements the *Drawable* interface.

While a class can **extend** only one other class, it can **implement** any number of interfaces. In fact, a class can both extend one other class and implement one or more interfaces. So, we can have things like

```
class FilledCircle extends Circle
                        implements Drawable, Fillable {
   . . .
}
```

The point of all this is that, although interfaces are not classes, they are something very similar. An interface is very much like an abstract class, that is, a class that can never be used for constructing objects, but can be used as a basis for making subclasses. The subroutines in an interface are abstract methods, which must be implemented in any concrete class that implements the interface. You can compare the *Drawable* interface with the abstract class

```
public abstract class AbstractDrawable {
   public abstract void draw(Graphics g);
}
```

The main difference is that a class that extends *AbstractDrawable* cannot extend any other class, while a class that implements *Drawable* can also extend some class, as well as implement other

interfaces. Of course, an abstract class can contain non-abstract methods as well as abstract methods. An interface is like a "pure" abstract class, which contains only abstract methods.

Note that the methods declared in an interface must be `public`. In fact, since that is the only option, it is not necessary to specify the access modifier in the declaration.

In addition to method declarations, an interface can also include variable declarations. The variables must be "`public static final`" and effectively become public static final variables in every class that implements the interface. In fact, since the variables can only be public and static and final, specifying the modifiers is optional. For example,

```
public interface ConversionFactors {
    int INCHES_PER_FOOT = 12;
    int FEET_PER_YARD = 3;
    int YARDS_PER_MILE = 1760;
}
```

This is a convenient way to define named constants that can be used in several classes. A class that implements *ConversionFactors* can use the constants defined in the interface as if they were defined in the class.

You are not likely to need to write your own interfaces until you get to the point of writing fairly complex programs. However, there are several interfaces that are used in important ways in Java's standard packages. You'll learn about some of these standard interfaces in the next few chapters, and you will write classes that implement them.

5.7.2 Interfaces as Types

As with abstract classes, even though you can't construct an object from an interface, you can declare a variable whose type is given by the interface. For example, if *Drawable* is the interface given above, and if *Line* and *FilledCircle* are classes that implement *Drawable*, as above, then you could say:

```
Drawable figure;  // Declare a variable of type Drawable.  It can
                  //    refer to any object that implements the
                  //    Drawable interface.

figure = new Line();  // figure now refers to an object of class
Line
figure.draw(g);   // calls draw() method from class Line

figure = new FilledCircle();   // Now, figure refers to an object
                               //   of class FilledCircle.
figure.draw(g);   // calls draw() method from class FilledCircle
```

A variable of type *Drawable* can refer to any object of any class that implements the *Drawable* interface. A statement like `figure.draw(g)`, above, is legal because `figure` is of type

Drawable, and **any** *Drawable* object has a `draw()` method. So, whatever object `figure` refers to, that object must have a `draw()` method.

Note that a type is something that can be used to declare variables. A type can also be used to specify the type of a parameter in a subroutine, or the return type of a function. In Java, a type can be either a class, an interface, or one of the eight built-in primitive types. These are the only possibilities. Of these, however, only classes can be used to construct new objects.

An interface can also be the base type of an array. For example, we can use an array type `Drawable[]` to declare variables and create arrays. The elements of the array can refer to any objects that implement the *Drawable* interface:

```
Drawable[] listOfFigures;
listOfFigures = new Drawable[10];
listOfFigures[0] = new Line();
listOfFigures[1] = new FilledCircle();
listOfFigures[2] = new Line();
  .
  .
  .
```

Every element of the array will then have a `draw()` method, so that we can say things like `listOfFigures[i].draw(g)`.

5.7.3 Interfaces in Java 8

The newest version of Java, Java 8, makes a few useful additions to interfaces. The one that I will discuss here is default methods. Unlike the usual abstract methods in interfaces, a default method has an implementation. When a class implements the interface, it does not have to provide an implementation for the default method -- although it can do so if it wants to provide a different implementation. Essentially, default methods are inherited from interfaces in much the same way that ordinary methods are inherited from classes. This moves Java partway towards supporting multiple inheritance. It's not true multiple inheritance, however, since interfaces still cannot define instance variables.

A default method in an interface must be marked with the modifier `default`. It can optionally be marked `public` but, as for everything else in interfaces, default methods are automatically public and the `public` modifier can be omitted. Here is an example.:

```
public interface Readable { // represents a source of input

    public char readChar();  // read the next character from the
input

    default public String readLine() { // read up to the next line
feed
        StringBuilder line = new StringBuilder();
```

```
        char ch = readChar();
        while (ch != '\n') {
            line.append(ch);
            ch = readChar();
        }
        return line.toString();
    }

}
```

A concrete class that implements this interface must provide an implementation for `readChar()`. It will inherit a definition for `readLine()` from the interface, but can provide a new definition if necessary. Note that the default `readLine()` calls the abstract method `readChar()`, whose definition will only be provided in the implementing class. The reference to `readChar()` in the definition is polymorphic. The default implementation of `readLine()` is one that would make sense in almost any class that implements *Readable*. Here's a rather silly example of a class that implements *Readable*, including a `main()` routine that tests the class. Can you figure out what it does?

```
public class Stars implements Readable {

    public char readChar() {
        if (Math.random() > 0.02)
           return '*';
        else
           return '\n';
    }

    public static void main(String[] args) {
        Stars stars = new Stars();
        for (int i = 0 ; i < 10; i++ ) {
            String line = stars.readLine();
            System.out.println( line );
        }
    }

}
```

Default methods provide Java with a capability similar to something called a "mixin" in other programming languages, namely the ability to mix functionality from another source into a class. Since a class can implement several interfaces, it is possible to mix in functionality from several different sources.

Nested Classes

A CLASS SEEMS LIKE IT SHOULD BE a pretty important thing. A class is a high-level building block of a program, representing a potentially complex idea and its associated data and behaviors. I've always felt a bit silly writing tiny little classes that exist only to group a few

scraps of data together. However, such trivial classes are often useful and even essential. Fortunately, in Java, I can ease the embarrassment, because one class can be nested inside another class. My trivial little class doesn't have to stand on its own. It becomes part of a larger more respectable class. This is particularly useful when you want to create a little class specifically to support the work of a larger class. And, more seriously, there are other good reasons for nesting the definition of one class inside another class.

In Java, a nested class is any class whose definition is inside the definition of another class. (In fact, a class can even be nested inside a subroutine, which must, of course, itself be inside a class). Nested classes can be either named or anonymous. I will come back to the topic of anonymous classes later in this section. A named nested class, like most other things that occur in classes, can be either static or non-static.

5.8.1 Static Nested Classes

The definition of a static nested class looks just like the definition of any other class, except that it is nested inside another class and it has the modifier `static` as part of its declaration. A static nested class is part of the static structure of the containing class. It can be used inside that class to create objects in the usual way. If it is used outside the containing class, its name must indicate its membership in the containing class. That is, the full name of the static nested class consists of the name of the class in which it is nested, followed by a period, followed by the name of the nested class. This is similar to other static components of a class: A static nested class is part of the class itself in the same way that static member variables are parts of the class itself.

For example, suppose a class named *WireFrameModel* represents a set of lines in three-dimensional space. (Such models are used to represent three-dimensional objects in graphics programs.) Suppose that the *WireFrameModel* class contains a static nested class, *Line*, that represents a single line. Then, outside of the class *WireFrameModel*, the *Line* class would be referred to as `WireFrameModel.Line`. Of course, this just follows the normal naming convention for static members of a class. The definition of the *WireFrameModel* class with its nested *Line* class would look, in outline, like this:

```
public class WireFrameModel {

   . . . // other members of the WireFrameModel class

   static public class Line {
         // Represents a line from the point (x1,y1,z1)
         // to the point (x2,y2,z2) in 3-dimensional space.
      double x1, y1, z1;
      double x2, y2, z2;
   } // end class Line

   . . . // other members of the WireFrameModel class

} // end WireFrameModel
```

The full name of the nested class is *WireFrameModel.Line*. That name can be used, for examples, to declare variables. Inside the *WireFrameModel* class, a *Line* object would be created with the constructor "`new Line()`". Outside the class, "`new WireFrameModel.Line()`" would be used.

A static nested class has full access to the static members of the containing class, even to the `private` members. Similarly, the containing class has full access to the members of the nested class, even if they are marked `private`. This can be another motivation for declaring a nested class, since it lets you give one class access to the private members of another class without making those members generally available to other classes. Note also that a nested class can itself be private, meaning that it can only be used inside the class in which it is nested.

When you compile the above class definition, two class files will be created. Even though the definition of *Line* is nested inside *WireFrameModel*, the compiled *Line* class is stored in a separate file. The name of the class file for *Line* will be `WireFrameModel$Line.class`.

5.8.2 Inner Classes

Non-static nested classes are referred to as inner classes. Inner classes are not, in practice, very different from static nested classes, but a non-static nested class is actually associated with an object rather than with the class in which its definition is nested. This can take some getting used to.

Any non-static member of a class is not really part of the class itself (although its source code is contained in the class definition). This is true for inner classes, just as it is for any other non-static part of a class. The non-static members of a class specify what will be contained in objects that are created from that class. The same is true -- at least logically -- for inner classes. It's as if each object that belongs to the containing class has its **own copy** of the nested class (although it does not literally contain a copy of the compiled code for the nested class). This copy has access to all the instance methods and instance variables of the object, even to those that are declared `private`. The two copies of the inner class in two different objects differ because the instance variables and methods they refer to are in different objects. In fact, the rule for deciding whether a nested class should be static or non-static is simple: If the nested class needs to use any instance variable or instance method from the containing class, make the nested class non-static. Otherwise, it might as well be static.

In most cases, an inner class is used only within the class where it is defined. When that is true, using the inner class is really not much different from using any other class. You can create variables and declare objects using the simple name of the inner class in the usual way.

From outside the containing class, however, an inner class has to be referred to using a name of the form **variableName.NestedClassName**, where **variableName** is a variable that refers to the object that contains the inner class. In order to create an object that belongs to an inner class, you

must first have an object that belongs to the containing class. (When working inside the class, the object "`this`" is used implicitly.)

Looking at an example will help, and will hopefully convince you that inner classes are really very natural. Consider a class that represents poker games. This class might include a nested class to represent the players of the game. The structure of the *PokerGame* class could be:

```
public class PokerGame {  // Represents a game of poker.

    class Player {  // Represents one of the players in this game.
        .
        .
        .
    } // end class Player

    private Deck deck;      // A deck of cards for playing the
game.
    private int pot;        // The amount of money that has been
bet.

    .
    .
    .

} // end class PokerGame
```

If `game` is a variable of type *PokerGame*, then, conceptually, `game` contains its own copy of the *Player* class. In an instance method of a *PokerGame* object, a new *Player* object would be created by saying "`new Player()`", just as for any other class. (A *Player* object could be created outside the *PokerGame* class with an expression such as "`game.new Player()`". Again, however, this is rare.) The *Player* object will have access to the `deck` and `pot` instance variables in the *PokerGame* object. Each `PokerGame` object has its own `deck` and `pot` and `Players`. Players of that poker game use the deck and pot for that game; players of another poker game use the other game's deck and pot. That's the effect of making the *Player* class non-static. This is the most natural way for players to behave. A *Player* object represents a player of one particular poker game. If *Player* were an independent class or a **static** nested class, on the other hand, it would represent the general idea of a poker player, independent of a particular poker game.

5.8.3 Anonymous Inner Classes

In some cases, you might find yourself writing an inner class and then using that class in just a single line of your program. Is it worth creating such a class? Indeed, it can be, but for cases like this you have the option of using an anonymous inner class. An anonymous class is created with a variation of the `new` operator that has the form

```
new  superclass-or-interface ( parameter-list ) {
```

```
                    methods-and-variables
                }
```

This constructor defines a new class, without giving it a name, and it simultaneously creates an object that belongs to that class. This form of the `new` operator can be used in any statement where a regular "`new`" could be used. The intention of this expression is to create: "a new object belonging to a class that is the same as **superclass-or-interface** but with these **methods-and-variables** added." The effect is to create a uniquely customized object, just at the point in the program where you need it. Note that it is possible to base an anonymous class on an interface, rather than a class. In this case, the anonymous class must implement the interface by defining all the methods that are declared in the interface. If an interface is used as a base, the **parameter-list** must be empty. Otherwise, it can contain parameters for a constructor in the **superclass**.

Anonymous classes are often used for handling events in graphical user interfaces, and we will encounter them several times in the chapters on GUI programming. For now, we will look at one not-very-plausible example. Consider the *Drawable* interface, which is defined earlier in this section. Suppose that we want a *Drawable* object that draws a filled, red, 100-pixel square. Rather than defining a new, separate class and then using that class to create the object, we can use an anonymous class to create the object in one statement:

```
Drawable redSquare = new Drawable() {
       void draw(Graphics g) {
          g.setColor(Color.RED);
          g.fillRect(10,10,100,100);
       }
   };
```

Then `redSquare` refers to an object that implements *Drawable* and that draws a red square when its `draw()` method is called. By the way, the semicolon at the end of the statement is not part of the class definition; it's the semicolon that is required at the end of every declaration statement.

Anonymous classes are often used for actual parameters. For example, consider the following simple method, which draws a *Drawable* in two different graphics contexts:

```
void drawTwice( Graphics g1, Graphics g2, Drawable figure ) {
    figure.draw(g1);
    figure.draw(g2);
}
```

When calling this method, the third parameter can be created using an anonymous inner class. For example:

```
drawTwice( firstG, secondG, new Drawable() {
          void draw(Graphics g) {
             g.drawOval(10,10,100,100);
          }
     } );
```

When a Java class is compiled, each anonymous nested class will produce a separate class file. If the name of the main class is `MainClass`, for example, then the names of the class files for the anonymous nested classes will be `MainClass$1.class`, `MainClass$2.class`, `MainClass$3.class`, and so on.

5.8.4 Java 8 Lambda Expressions

The syntax for anonymous classes is cumbersome. In many cases, an anonymous class implements an interface that defines just one method. Java 8 introduces a new syntax that can be used in place of the anonymous class in that circumstance: the lambda expression. Here is what the previous subroutine call looks like using a lambda expression:

```
drawTwice( firstG, secondG, g -> g.drawOval(10,10,100,100) )
```

The lambda expression is `g -> g.drawOval(10,10,100,100)`. Its meaning is, "the method that has a parameter `g` and executes the code `g.drawOval(10,10,100,100)`." The computer knows that `g` is of type *Graphics* because it is expecting a *Drawable* as the actual parameter, and the only method in the *Drawable* interface has a parameter of type *Graphics*. Lambda expressions can only be used in places where this kind of type inference can be made. The general syntax of a lambda expression is

```
formal-parameter-list  ->  method-body
```

where the **method body** can be a single expression, a single subroutine call, or a block of statements enclosed between `{` and `}`. When the body is a single expression or function call, the value of the expression is automatically used as the return value of the method that is being defined. The parameter list in the lambda expression does not have to specify the types of the parameters, although it can. Parentheses around the parameter list are optional if there is exactly one parameter and no type is specified for the parameter; this is the form seen in the example above. For a method with no parameters, the parameter list is just an empty set of parentheses. Here are a few more examples of lambda expressions:

```
() -> System.out.println("Hello World")

g -> { g.setColor(Color.RED); g.drawRect(10,10,100,100); }

(a, b) -> a + b

(int n) -> {
          while (n > 0) {
              System.out.println(n);
              n = n/2;
           }
      } // lambda expressions ends here
```

As you can see, the syntax can still get pretty complicated. There is quite a lot more to say about lambda expressions, but my intention here is only to briefly introduce one of the most interesting new features in Java 8.