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. In the next chapter, we will begin the study of Java's GUI classes, and you will encounter

other built-in classes throughout the remainder of this book. But 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 *StringBuffer* 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 *StringBuffer* class. For example:

```
StringBuffer buffer = new StringBuffer();
```

(This statement both declares the variable `buffer` and initializes it to refer to a newly created *StringBuffer* 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 *StringBuffer* contains a sequence of characters. However, it is possible to add new characters onto the end of a `StringBuffer` without making a copy of the data that it already contains. If `x` is a value of any type and `buffer` is the variable defined above, then the command `buffer.append(x)` will add `x`, converted into a string representation, onto the end of the data that was already in the buffer. This command actually modifies the buffer, rather than making a copy, and that can be done efficiently. A long string can be built up in a *StringBuffer* using a sequence of `append()` commands. When the string is complete, the function `buffer.toString()` will return a copy of the string in the buffer as an ordinary value of type *String*. The *StringBuffer* 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 encounter an example in Section 5.5, and we will study the 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.

The main point here, 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 Wrapper Classes and Autoboxing

We have already encountered the classes *Double* and *Integer* in Subsection 2.5.7. These classes contain the `static` methods `Double.parseDouble` and `Integer.parseInteger` that are used to convert strings to numerical values. We have also encountered the *Character* class in some examples, with static methods such as `Character.isLetter`, which can be used to test whether a given value of type `char` is a letter. There is a similar class for each of the other primitive types, *Long*, *Short*, *Byte*, *Float*, and *Boolean*. These classes are called wrapper classes. Although they contain useful `static` members, they have another use as well: They are used for creating objects that represent primitive type values.

Remember that the primitive types are not classes, and values of primitive type are not objects. However, sometimes it's useful to treat a primitive value as if it were an object. You can't do that literally, but you can "wrap" the primitive type value in an object belonging to one of the wrapper classes.

For example, an object of type *Double* contains a single instance variable, of type double. The object is a wrapper for the double value. For example, you can create an object that wraps the double value `6.0221415e23` with

```
Double d = new Double(6.0221415e23);
```

The value of `d` contains the same information as the value of type double, but it is an object. If you want to retrieve the double value that is wrapped in the object, you can call the function `d.doubleValue()`. Similarly, you can wrap an int in an object of type *Integer*, a boolean value in an object of type *Boolean*, and so on. (As an example of where this would be useful, the collection classes that will be studied in Chapter 10 can only hold objects. If you want to add a primitive type value to a collection, it has to be put into a wrapper object first.)

Since Java 5.0, wrapper classes have been even easier to use. Java 5.0 introduced automatic conversion between a primitive type and the corresponding wrapper class. For example, if you use a value of type int in a context that requires an object of type *Integer*, the int will automatically be wrapped in an *Integer* object. For example, you can say

```
Integer answer = 42;
```

and the computer will silently read this as if it were

```
Integer answer = new Integer(42);
```

This is called autoboxing. It works in the other direction, too. For example, if `d` refers to an object of type `Double`, you can use `d` in a numerical expression such as `2*d`. The double value inside `d` is automatically unboxed and multiplied by 2. Autoboxing and unboxing also apply to subroutine calls. For example, you can pass an actual parameter of type int to a subroutine that has a formal parameter of type *Integer*. In fact, autoboxing and unboxing make it possible in many circumstances to ignore the difference between primitive types and objects.

The wrapper classes contain a few other things that deserve to be mentioned. *Integer*, for example, contains constants `Integer.MIN_VALUE` and `Integer.MAX_VALUE`, which are equal to the largest and smallest possible values of type int, that is, to -2147483648 and 2147483647 respectively. It's certainly easier to remember the names than the numerical values. There are similar named constants in *Long*,*Short*, and *Byte*. *Double* and *Float* also have constants named `MIN_VALUE` and `MAX_VALUE`. `MAX_VALUE` still gives the largest number that can be represented in the given type, but `MIN_VALUE` represents the smallest possible **positive** value. For type double, `Double.MIN_VALUE` is 4.9 times 10^{-324}. Since double values have only a finite accuracy, they can't get arbitrarily close to zero. This is the closest they can get without actually being equal to zero.

The class *Double* deserves special mention, since doubles are so much more complicated than integers. The encoding of real numbers into values of type double has room for a few special values that are not real numbers at all in the mathematical sense. These values are given by named constants in class *Double*: `Double.POSITIVE_INFINITY`, `Double.NEGATIVE_INFINITY`, and `Double.NaN`. The infinite values can occur as the values of certain mathematical expressions. For example, dividing a positive number by zero will give the result `Double.POSITIVE_INFINITY`. (It's even more complicated than this, actually, because the double type includes a value called "negative zero", written `-0.0`. Dividing a positive number by negative zero gives `Double.NEGATIVE_INFINITY`.) You also get `Double.POSITIVE_INFINITY`whenever the mathematical value of an expression is greater than `Double.MAX_VALUE`. For example, `1e200*1e200` is considered to be infinite. The value `Double.NaN` is even more interesting. "NaN"

stands forNot a Number, and it represents an undefined value such as the square root of a negative number or the result of dividing zero by zero. Because of the existence of `Double.NaN`, no mathematical operation on real numbers will ever throw an exception; it simply gives `Double.NaN` as the result.

You can test whether a value, `x`, of type double is infinite or undefined by calling the boolean-valued static functions `Double.isInfinite(x)` and `Double.isNaN(x)`. (It's especially important to use`Double.isNaN()` to test for undefined values, because `Double.NaN` has really weird behavior when used with relational operators such as `==`. In fact, the values of `x == Double.NaN` and `x != Double.NaN`are always **both false** -- no matter what the value of `x` is -- so you can't use these expressions to test whether `x` is `Double.NaN`.)

5.3.3 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.

5.3.4 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.

Source : http://math.hws.edu/javanotes/c5/s3.html