MORE ON CLASSES IN JAVA

Several of our programs have used classes such as `Turtle`, `GOval`, and `Color`. Classes are quite important to Java programming; it is time to examine some of their features more closely.

10.1. Packages

The full Java system comes with *many* classes — in fact, at the time of this writing, the latest version includes about 3,000 classes. Moreover, there are many organizations who distribute additional libraries of classes, to make it easier for others to develop software. The `acm` library that we have been using is one such third-party library, intended for beginning students to be able to work with graphics.

To allow organizing all of these classes, and to address the inevitable problem of having some of the classes having identical names, Java uses the notion of a package to group classes together. An example is the `acm.graphics` package, which includes classes like `GOval` and `GRect`.

The packages distributed with Java have names beginning with `java` or `javax`. One such package is the `java.awt` package, which includes the `Color` class that we have been using. There are many more packages, several of which we will see in the remainder of this book.

When a program refers to a class, the compiler must be able to determine which package contains that class. To specify the packages, we begin each program with a sequence of`import` lines.

```
import acm.graphics.*;
import acm.program.*;
import java.awt.*;
```

The asterisk ('*') indicates that our program will freely use any combination of the classes from that package. Thus, when the compiler sees a reference to the `GOval` class, it will examine each of the three packages named to determine which package contains it.

If you omit the `import java.awt.*;` line but still attempt to access a class in that package, such as the `Color` class, then the compiler will point to the occurrence of `Color` in the program and give an error message such as cannot find symbol.

One can also import classes individually rather than use the asterisk. For example, if we know that the only class we will use from the `acm.program` package is `GraphicsProgram`, and the only classes we will use from `java.awt` are `Color` and `Font`, then we might write the following.

```
import acm.graphics.*;
import acm.program.GraphicsProgram;
import java.awt.Color;
import java.awt.Font;
```

Many professional Java programmers maintain that good Java programs should always import classes individually rather than use '*', mostly because it makes it easy for a reader to quickly see what classes the program depends upon. Importing individual classes is particularly useful when using two packages that contain classes with identical names.

There is one special package: All classes in `java.lang` are automatically imported, with or without an `import` line. The package includes several classes, most with some special meaning within the Java base language. Examples of classes in `java.lang` include `String` (Chapter 8), `Math` (later in this chapter), and `Object` (Chapter 11).

Java does allow a program to reference classes without first importing them, even when the class is outside `java.lang`. To do this, you incorporate the package name into the class name every time it appears in the program. Thus, you can omit the `import java.awt.Color;` line, but if you do this, you must write `java.awt.Color` in your program every time you would otherwise have written just `Color`. For example, instead of writing `new Color(255, 0, 0)`, you would write`new java.awt.Color(255, 0, 0)`.

I strongly encourage avoiding this in all cases. In rare cases, though, it is unavoidable. This happens when you are writing a program that uses two identically named types occurring in different packages, such as `List`, which is in both `java.awt` and `java.util`. This can normally be avoided by steering clear of the '`*`' notation in `import`lines. But if somehow you have a program that actually uses both types of the same name, there is no other way around it.

10.2. Static methods

There are three categories of methods that can be in a class. We've seen two: *instance methods*, which are applied to individual objects of the class, and *constructors*, which are used to manufacture new instances of the class. (Technically, constructors aren't methods, but they have enough similarities that we won't let this technicality bother us.) The third category of method is the static method, which is a method that is applied to the class as a whole, rather than to individual objects of that class. Static methods are sometimes also called class methods.

In the documentation for a class, a static methods is identified by the keyword `static`.
The `Math` class (in `java.lang`) is an example of a class with several `static` methods, including the following.

`static int abs(int x)`

Returns the absolute value of the parameter `x`. (Note `x` is an `int` here.)

`static double abs(double x)`

Returns the absolute value of the parameter `x`. (Note `x` is a `double` here.)

`static double pow(double base, double exponent)`

Returns the result of raising `base` to the power `exponent` (i.e., $base^{exponent}$).

`static double random()`

Returns a pseudorandom number between 0.0 and 1.0.

`static double sqrt(double x)`

Returns the square root of `x`.

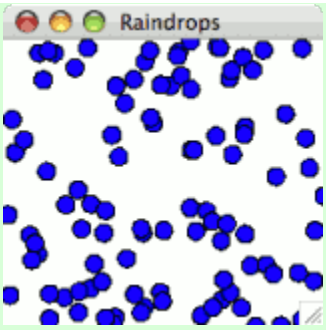
To invoke a static method, write the class name and method name, separated by a period, followed by parentheses containing any parameter values: `Math.sqrt(4)` is an example (which gives the value 2.0). Figure 10.1 includes a sample program that splatters blue circles randomly on the screen, as demonstrated in Figure 10.2.

Figure 10.1: The `Raindrops` program.

```
1  import acm.program.*;
2  import acm.graphics.*;
3  import java.awt.*;
4
5  public class Raindrops extends GraphicsProgram {
6      public void run() {
7          while(true) {
8              pause(40);
9              double x = Math.random() * getWidth();
10             double y = Math.random() * getHeight();
11             GOval drop = new GOval(x - 5, y - 5, 10, 10);
12             drop.setFilled(true);
13             drop.setFillColor(new Color(0, 0, 255));
14             add(drop);
```

```
15            }
16        }
17    }
```

Figure 10.2: Running the `Raindrops` program of Figure 10.1.



Notice that though this program must create a `GOval` object to invoke its `setFilled` method, there is no need to create a `Math` object to invoke its `random` method. Thus, a program could never invoke `setFilled` as `GOval.setFilled`, since `setFilled` requires the program to specify before the period the instance to which the method is applied; however, the program can legally have `Math.random`, without ever creating any `Math` instances. This is because `setFilled` is an instance method, while `random` is a static method. You can think of the static method as something that the class as a whole does, whereas an instance method is something that an individual object of that class does.

(Since `Math` contains only static methods, in fact there is never any need to create a `Math` object. It is possible, though, to have a class that mixes both instance methods and static methods. The `String` class is an example.)

You may ask: Why are they called *static*? The word *static* is being used in this context with its meaning of *fixed*. Since is just one thing (the class) that can perform the method, the method's destination is fixed; by contrast, the thing (object) that performs an instance method changes depending on what object is specified preceding the period.

In fact, the choice of the word *static* is based just as much on historical quirks. The word *static* derives from Java's predecessor language, C, where the word was used for very different purposes. Over the years it has been adapted for this different purpose, since it seemed vaguely suitable and didn't require adding new keywords into the language. Language designers prefer to avoid adding new keywords because adding a new keyword means that any preexisting program that used that word as a variable name will become broken and thus require modification before being adapted into the new language version.

10.3. Constants

Many classes define constants, which are listed in the documentation as being both *static* (i.e., associated with the class) and *final* (i.e., the value doesn't change). They are essentially variables whose value cannot change. An example of a constant can be found in `java.lang`'s `Math` class.

`static final double PI`

> The value of the mathematical constant π (i.e., 3.14159…).

This gives an easy way to refer to the value of π in a program without having to recite all of its digits. Like static methods, we identify the constant by writing first the name of the class containing the constant, followed by a period and then the name of the constant. Thus, we reference π by writing `Math.PI`.

```
double radius = readDouble();
double area = Math.PI * radius * radius;
```

The `Color` class defines many more useful constants for referring to individual colors, including black and red. (It also includes the constants `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`,`LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `WHITE`, and `YELLOW`.)

`static final Color BLACK`

> A `Color` object approximating the color black.

`static final Color RED`

> A `Color` object approximating the color red.

Thus, we can modify the body of our `BasicBall` program of Figure 6.1 to the following.

```
GOval ball = new GOval(75, 75, 50, 50);
ball.setFilled(true);
ball.setFillColor(Color.RED);
add(ball);
```

Experienced programmers use constants wherever possible, since they make a program easier to understand. Thus, an expert would consider the above to be preferable to constructing a new `Color` object to represent red, as we have been doing heretofore.

By convention, constants are usually named using all capital letters, like `PI`. If there are multiple words in the name, the constant will include an underscore to separate the words. (There are a few constants in the built-in Java library that don't follow this convention. Generally, these predate the firm establishment of this naming convention. An example of such a constant is `red` in the `Color` class. The `red` constant was defined in the earliest version of Java, but this was soon repaired in a later version by also introducing`RED` to mean the same thing. They had to keep `red` so that old programs using that name would still compile under newer versions of Java.)

Source : http://www.toves.org/books/java/ch10-moreclass/index.html