USING NUMBERS IN JAVA

We now turn to working with numbers in computer programs: In this chapter, we see how a program might ask the user for a number and use that number in doing its task. We'll close with a related, important discussion of run-time exceptions.

4.1. Numeric types

We've glanced at the `double` type in Java, which we can use to represent a number. Using this type, we can, if we want, make a variable to refer to a number.

```
double scale;
```

This line creates a variable named `scale`, of type `double`. We can assign a number to the variable using an assignment statement.

```
scale = 2.4;
```

Notice that we don't have to use `new` here; we use `new` only in conjunction with constructors, and the numeric types don't have constructors.

The numeric types don't have constructors because these types are technically not objects. Instead, `double` is one of a few primitive types — that is, types that are built into the core of the Java language. There are a total of eight primitive types in Java, of which we will study three in this book: `boolean`, `double`, and `int`. (The other five are `char`, `byte`,`short`, `long`, and `float`.) You can see that the primitive types are all spelled entirely with lower-case letters. By contrast, classes are conventionally named starting with capital letters, as in `Turtle` or `Color`.

In this chapter, we'll examine both the `double` and the `int` types. The `int` type is for representing integers. (An integer is a number with no fractional part, like 0 or −3, but not 1.4.)

```
int i;
i = 5;
```

You cannot make an `int` variable reference a `double` value, even if the value has no fractional part. For example, the below fragment is illegal because it attempts to assign a`double` value to a variable whose type is `int`. Java regards any number containing a decimal point as a `double` value, including 5.0.

```
i = 5.0;       // Illegal!!
```

Java will happily allow a `double` variable to be assigned an `int` value, however, on the grounds that `int` is more restricted than `double`; any `int` value can be converted to a legitimate `double` value.

```
scale = 2;     // Ok
```

You may be wondering: Why would you ever use `int` instead of `double`? Why limit yourself? There are three reasons for this. First, computers use scientific notation with limited precision to remember `double` values. While this allows the computer to remember a wider range of numbers, it can only approximate most of the numbers in that range. An `int`value always represents the integer exactly. Second, computers compute using `double`s somewhat more slowly. (Think about how much more difficult it is to add two numbers in scientific notation than to add two simple integers.) And, finally, making the distinction rarely causes much trouble: After a bit of practice, the distinction is second nature. In practice, most programs use very few `double` values.

4.2. Return values

We've already seen that we can send a value to a method via a parameter. But, often, a method needs to provide some sort of response; for this, the method can use a return value.

The methods we have seen so far have not had return values. But the `TurtleProgram` class includes the following two instance methods with return values.

```
double readDouble()
```

> Shows the user a dialog box prompting for a number and returns the entered number.

```
int readInt()
```

> Shows the user a dialog box prompting for an integer and returns the entered integer.

The type written before the method name in the method descriptor (`double` for `readDouble` and `int` for `readInt`) specifies what type of value the method will return. Although a method can accept several values as parameters, it can have only one return value. (It happens that neither of these methods takes any parameters.)

In the methods we examined last chapter (such as `forward`), you'll recall the word `void` preceded the name of the method.

```
void forward(int dist)
```

The word **void** as a return type indicates that the method does not have a return value.

Before we use any of the `TurtleProgram` methods, we first must address the question: How can we find a `TurtleProgram` object? As it happens, we won't use a constructor, as we have with the other classes `Turtle` and `Color`. In fact, when the computer is told to run our programs, it creates a `TurtleProgram` object and then execute its `run` method:
The**extends** `TurtleProgram` and **void** `run()` bits at the top of the programs we've been writing are really saying that we are writing a `run` method for a class that is a slightly modified version of `TurtleProgram`. You needn't understand that fully, because we'll study it much more thoroughly later; the upshot, though, is that the computer has already created a `TurtleProgram` object by the time it executes our program, which is the object that we want to use. We can access this object by typing, simply, **this**. Thus, to tell our`TurtleProgram` object to execute its `readDouble` method, we will write **this**`.readDouble()`.

But we won't write that exactly, because then the return value will be lost. To be able to reference the return value later in the program, we will use an assignment statement to assign a variable to remember whatever value the method returns.

```
scale = this.readDouble();
```

Here, we've assigned the name `scale` to refer to the value returned when we tell **this** to read a **double**. Since `readDouble` returns what the user types into a dialog box, this statement assigns `scale` to be a name for whatever number the user entered.

The `DrawSquare` program of Figure 4.1 illustrates this at work.

Figure 4.1: The `DrawSquare` program.

```
1  import turtles.*;
2
3  public class DrawSquare extends TurtleProgram {
4      public void run() {
5          double sideLength;
6          sideLength = this.readDouble();
7
8          Turtle boxTurtle;
9          boxTurtle = new Turtle(10, 10);
10         boxTurtle.forward(sideLength);
11         boxTurtle.right(90);
12         boxTurtle.forward(sideLength);
13         boxTurtle.right(90);
```

```
14          boxTurtle.forward(sideLength);
15          boxTurtle.right(90);
16          boxTurtle.forward(sideLength);
17          boxTurtle.hide();
18      }
19  }
```

You can see that line 6 tells **`this`** to read a number, with `sideLength` assigned to remember the number entered by the user. We can then use the user's number as a parameter to`forward` in line 10 to indicate that the turtle should move forward as many pixels as the user indicated.

Source : http://www.toves.org/books/java/ch04-arith/index.html