Programming via Java Using numbers

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 areturn 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.

```
import turtles.*;

public class DrawSquare extends TurtleProgram {
    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.

4.3. Arithmetic

Since numbers are not objects in Java, and their types are not classes, they cannot perform instance methods. Instead, to work with primitive values, we useoperators, special symbols that can be used to compute a value. Java includes many operators, including several arithmetic operators for numeric values.

`+` addition
`-` subtraction
`*` multiplication
`/` division
`%` modulus (remainder)

(The modulus operator (`%`) yields the remainder. For example, `11 % 3` would yield the integer 2, since when you divide 11 by 3, the remainder is 2.)

We can combine these operators as we like. Java will follow the normal algebraic order of operations: Multiplication and division (and modulus) have priority over addition and subtraction. And, within the same level of precedence, the computation proceeds left to right. Suppose I write something like the following.

```
start = 100 - length / 2.0;
```

The first thing that happens is that `length` is divided by 2; then this result is subtracted from 100; and finally `start` is assigned to refer to this result. (The value of `length`itself remains unchanged by this statement.) If we really wanted to do the subtraction first, then we can use parentheses.

```
start = (100 - length) / 2.0;
```

When you operate on two integers, Java will produce an integer result. But if either side is a **`double`**, Java will produce a **`double`** result. This works quite smoothly in practice, except for one thing: When you divide two integers, of course the true result is not necessarily an integer. In order to get an integer result, Java will simply ignore any remainder. Thus, `14 / 3` has the value 4. (Its true value is 4.666..., but the remainder of 2 is ignored since both `14` and `3` are **`int`** values.)

This dropping of the remainder is guaranteed to cause you headaches some day! Note that dropping the remainder will occur even if the result is assigned to a**`double`**.

```
double scale;
scale = 1 / 2;    // Sets scale to 0.0
```

It looks like this line assigns 0.5 to `scale`, but in fact it assigns 0 to it: The computer determines that 1 goes 2 zero times (with a remainder of 1, but the remainder is dropped).

The solution is simple: Convert either side of the division to a **`double`** value, as with `1.0 / 2`. Or, if neither side of the division is a constant value, as when you're dividing two **`int`** variables, then you can multiply one of **`int`** values by `1.0` to arrive at a **`double`** before performing the division.

Figure 4.2 contains a program that uses operators. It asks the user for a number (Figure 4.3(a)) and then draws a square exactly that big in the center of the window (Figure 4.3(b) and (c)).

Figure 4.2: The `DrawCenterSquare` program.

```
import turtles.*;

```

```
3  public class DrawCenterSquare extends TurtleProgram {
4      public void run() {
5          double sideLength;
6          sideLength = this.readDouble();
7
8          double start;
9          start = 100.0 - sideLength / 2.0;
10
11         Turtle boxTurtle;
12         boxTurtle = new Turtle(start, start);
13         boxTurtle.forward(sideLength);
14         boxTurtle.right(90);
15         boxTurtle.forward(sideLength);
16         boxTurtle.right(90);
17         boxTurtle.forward(sideLength);
18         boxTurtle.right(90);
19         boxTurtle.forward(sideLength);
20         boxTurtle.hide();
21     }
22 }
```

Figure 4.3: Running `DrawCenterSquare`.

(a) **(b)** **(c)**

When run, the computer begins at line 5 and works through the lines one by one.

1. **Lines 5–6:** Creates a **`double`** variable called `sideLength` for referring to the length of each side of the square to be drawn. Then it asks the user for a number and assigns `sideLength` to refer to this number. In Figure 4.3(a), the user types 60, so in the example of Figure 4.3, `sideLength` will represent the number 60.
2. **Lines 8–9:** Creates another variable called `start` that will represent the *x*-coordinate of the top left corner of the square. (Since the *x*- and *y*-coordinates of this

point are equal, this variable will do double-duty as the *y*-coordinate also.) Its value is computed in line 9 using an arithmetic expression. Since the window is 200 pixels wide, 100 is halfway between the left and right sides; we want half of the side length to extend to the left of this, so we want to subtract `sideLength` / 2 from 100. Thus, this is the expression you see in line 9, whose value is assigned to `start`. In our example, where the user had typed 60 for `sideLength`, `start` would be assigned the value 70.

3. **Lines 11–12:** Creates another variable `boxTurtle` for referring to a `Turtle` object. The turtle's initial *x*- and *y*-coordinates being `start`. Thus, this turtle would be placed at (70, 70) facing east.
4. **Lines 13–20:** Tells the turtle to trace a square with each side being `sideLength` pixels long, and then to hide, leaving the square behind.

In line 12, we passed a variable's value as a parameter, instead of simply a number. More generally, we could write any expression for a parameter, and the computer will compute its value before calling the method. When we create the turtle, we could write the following instead.

```
boxTurtle = new Turtle(100 - sideLength / 2, 100 - sideLength / 2);
```

This would remove the need for the `start` variable, so lines 8 and 9 could be removed.

4.4. Exceptions

Hopefully, you've been writing your own programs as you've been going along. Those programs have surely sometimes had problems. There are three categories of problems that a program might have. (We alluded to these before in Section 1.2, but we're now prepared to review them specifically with reference to Java.)

- **Compile errors:** The compiler identifies a problem within your program, and it refuses to compile the program, so that the computer is unable to attempt executing the program. For beginning programmers, one of the most common compile errors is omitting a semicolon at the end of a statement.
- **Logic errors:** The computer successfully compiles and executes the program, but it doesn't quite do what you expected. An example would be omitting the `hide`invocation on a turtle, so that the turtle remains on the screen even though you wanted the turtle to go away once it completed its task.
- **Exceptions:** The computer identifies a fatal problem while running the program, and it terminates the program with a message telling about the problem. One way

this could occur is if the program attempts somehow to perform integer division where the divisor is zero.

You may well not have encountered any exceptions yet, but as we go on you will encounter them more frequently, and now is a good time to learn how to recognize them. Consider the `DrawFraction` program of Figure 4.4. It reads a number *n* from the user, and it draws a line that extends an *n*th of the way across the window.

Figure 4.4: The `DrawFraction` program.

```
1  import turtles.*;
2
3  public class DrawFraction extends TurtleProgram {
4      public void run() {
5          int divisor;
6          divisor = this.readInt();
7
8          Turtle lineTurtle;
9          lineTurtle = new Turtle(0, 100);
10         lineTurtle.forward(200 / divisor);
11         lineTurtle.hide();
12     }
13 }
```

Now suppose that the user enters 0 as the number. Then when the computer reaches line 10, the computer will have to divide the two integers 200 and 0. At this point, the computer will give up and display a message like the following. (You may have to look to find the message; it won't appear in the turtles' window, and it probably appear in a dialog box. You will probably have to look somewhere else to get the information.)

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DrawFraction.run(DrawFraction.java:10)
        at acm.program.Program.runHook(Program.java)
        at acm.program.Program.startRun(Program.java)
        at acm.program.Program.start(Program.java)
        at acm.program.Program.start(Program.java)
        at acm.program.Program.main(Program.java)
```

This message is rather cryptic, but in fact it provides quite a bit of information that's helpful for identifying what went wrong with the program. The first thing it tells you is the sort of thing that went wrong. To find this out, you examne the first line and look for a long mixedcase word, probably ending in Exception. Here, it saysArithmeticException. This word indicates the category of problem that occurred. Sometimes, there will be a colon after this long word with additional helpful information; here, the information, / by zero, describes exactly the problem.

The next lines are also very useful, as they describe where the computer was in the program it encountered the problem. Notice the parentheses where it saysDrawFraction.java:10. This identifies exactly which line of the program contained the error.

Sometimes the first line beginning with at will indicate a line that is not in your own code, but instead in some method that you've invoked. You should keep looking down through the at lines until you find a line that identifies a line in your code. In this case, we would look for the first line mentioning `DrawFraction.java`, since our program was named `DrawFraction` in line 3.

Exercise 4.1

Modify the `DrawTriangle` program of Figure 3.1 so that it reads two numbers from the user, x and y, and places the triangle with its lower left corner at (x, y).

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