Chapter 7

Loops

Computers are often used to automate repetitive tasks. Repeating tasks without making errors is something that computers do well and people do poorly.

Running the same code multiple times is called **iteration**. We have seen methods, like `countdown` and `factorial`, that use recursion to iterate. Although recursion is elegant and powerful, it takes some getting used to. Java provides language features that make iteration much easier: the `while` and `for` statements.

7.1 The while statement

Using a `while` statement, we can rewrite `countdown` like this:

```
public static void countdown(int n) {
    while (n > 0) {
        System.out.println(n);
        n = n - 1;
    }
    System.out.println("Blastoff!");
}
```

You can almost read the `while` statement like English: "While `n` is greater than zero, print the value of `n` and then reduce the value of `n` by 1. When you get to zero, print Blastoff!"

The expression in parentheses is called the condition. The statements in braces are called the **body**. The flow of execution for a `while` statement is:

1. Evaluate the condition, yielding `true` or `false`.
2. If the condition is `false`, skip the body and go to the next statement.
3. If the condition is `true`, execute the body and go back to step 1.

This type of flow is called a **loop**, because the last step loops back around to the first.

The body of the loop should change the value of one or more variables so that, eventually, the condition becomes `false` and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**. An endless source of amusement for computer scientists is the observation that the directions on shampoo, "Lather, rinse, repeat," are an infinite loop.

In the case of `countdown`, we can prove that the loop terminates when `n` is positive. But in general, it is not so easy to tell whether a loop terminates. For example, this loop continues until `n` is 1 (which makes the condition `false`):

```
public static void sequence(int n) {
    while (n != 1) {
        System.out.println(n);
        if (n % 2 == 0) {          // n is even
            n = n / 2;
        } else {                   // n is odd
            n = n * 3 + 1;
        }
    }
}
```

Each time through the loop, the program displays the value of `n` and then checks whether it is even or odd. If it is even, the value of `n` is divided by two. If it is odd, the value is replaced by $3n + 1$. For example, if the starting value (the argument passed to `sequence`) is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1.

Since `n` sometimes increases and sometimes decreases, there is no obvious proof that `n` will ever reach 1 and that the program will ever terminate. For some

values of `n`, we can prove that it terminates. For example, if the starting value is a power of two, then the value of `n` will be even every time through the loop until we get to 1. The previous example ends with such a sequence, starting when `n` is 16.

The hard question is whether this program terminates for *all* values of n. So far, no one has been able to prove it *or* disprove it! For more information, see https://en.wikipedia.org/wiki/Collatz_conjecture.

7.2 Generating tables

Loops are good for generating and displaying tabular data. Before computers were readily available, people had to calculate logarithms, sines and cosines, and other common mathematical functions by hand. To make that easier, there were books of tables where you could look up values of various functions. Creating these tables by hand was slow and boring, and the results were often full of errors.

When computers appeared on the scene, one of the initial reactions was: "This is great! We can use a computer to generate the tables, so there will be no errors." That turned out to be true (mostly), but shortsighted. Not much later, computers were so pervasive that printed tables became obsolete.

Even so, for some operations, computers use tables of values to get an approximate answer, and then perform computations to improve the approximation. In some cases, there have been errors in the underlying tables, most famously in the table the original Intel Pentium used to perform floating-point division (see https://en.wikipedia.org/wiki/Pentium_FDIV_bug).

Although a "log table" is not as useful as it once was, it still makes a good example of iteration. The following loop displays a table with a sequence of values in the left column and their logarithms in the right column:

```
int i = 1;
while (i < 10) {
    double x = (double) i;
    System.out.println(x + "    " + Math.log(x));
    i = i + 1;
}
```

The output of this program is:

```
1.0   0.0
2.0   0.6931471805599453
3.0   1.0986122886681098
4.0   1.3862943611198906
5.0   1.6094379124341003
6.0   1.791759469228055
7.0   1.9459101490553132
8.0   2.0794415416798357
9.0   2.1972245773362196
```

`Math.log` computes natural logarithms, that is, logarithms base e. For computer science applications, we often want logarithms with respect to base 2. To compute them, we can apply this equation:

$$\log_2 x = \frac{log_e x}{log_e 2}$$

We can modify the loop as follows:

```
int i = 1;
while (i < 10) {
    double x = (double) i;
    System.out.println(x + "   " + Math.log(x) / Math.log(2));
    i = i + 1;
}
```

And here are the results:

```
1.0   0.0
2.0   1.0
3.0   1.5849625007211563
4.0   2.0
5.0   2.321928094887362
6.0   2.584962500721156
7.0   2.807354922057604
8.0   3.0
9.0   3.1699250014423126
```

Each time through the loop, we add one to `x`, so the result is an arithmetic sequence. If we multiply `x` by something instead, we get a geometric sequence:

```
final double LOG2 = Math.log(2);
int i = 1;
while (i < 100) {
    double x = (double) i;
    System.out.println(x + "   " + Math.log(x) / LOG2);
    i = i * 2;
}
```

The first line stores `Math.log(2)` in a `final` variable to avoid computing that value over and over again. The last line multiplies `x` by 2. The result is:

```
1.0   0.0
2.0   1.0
4.0   2.0
8.0   3.0
16.0   4.0
32.0   5.0
64.0   6.0
```

This table shows the powers of two and their logarithms, base 2. Log tables may not be useful anymore, but for computer scientists, knowing the powers of two helps a lot!

7.3 Encapsulation and generalization

In Section 6.2, we presented a way of writing programs called incremental development. In this section we present another **program development** process called "encapsulation and generalization". The steps are:

1. Write a few lines of code in `main` or another method, and test them.
2. When they are working, wrap them in a new method, and test again.
3. If it's appropriate, replace literal values with variables and parameters.

The second step is called **encapsulation**; the third step is **generalization**.

To demonstrate this process, we'll develop methods that display multiplication tables. Here is a loop that displays the multiples of two, all on one line:

```
int i = 1;
while (i <= 6) {
    System.out.printf("%4d", 2 * i);
    i = i + 1;
}
System.out.println();
```

The first line initializes a variable named `i`, which is going to act as a **loop variable**: as the loop executes, the value of `i` increases from 1 to 6; when `i` is 7, the loop terminates.

Each time through the loop, we display the value `2 * i` padded with spaces so it's four characters wide. Since we use `System.out.printf`, the output appears on a single line.

After the loop, we call `println` to print a newline and complete the line. Remember that in some environments, none of the output is displayed until the line is complete.

The output of the code so far is:

```
   2   4   6   8  10  12
```

The next step is to "encapsulate" this code in a new method. Here's what it looks like:

```
public static void printRow() {
    int i = 1;
    while (i <= 6) {
        System.out.printf("%4d", 2 * i);
        i = i + 1;
    }
    System.out.println();
}
```

Next we replace the constant value, 2, with a parameter, `n`. This step is called "generalization" because it makes the method more general (less specific).

```
public static void printRow(int n) {
    int i = 1;
    while (i <= 6) {
        System.out.printf("%4d", n * i);
        i = i + 1;
    }
    System.out.println();
}
```

Invoking this method with the argument 2 yields the same output as before. With the argument 3, the output is:

```
   3   6   9  12  15  18
```

And with argument 4, the output is:

```
   4   8  12  16  20  24
```

By now you can probably guess how we are going to display a multiplication table: we'll invoke `printRow` repeatedly with different arguments. In fact, we'll use another loop to iterate through the rows.

```
int i = 1;
while (i <= 6) {
    printRow(i);
    i = i + 1;
}
```

And the output looks like this:

```
   1   2   3   4   5   6
   2   4   6   8  10  12
   3   6   9  12  15  18
   4   8  12  16  20  24
   5  10  15  20  25  30
   6  12  18  24  30  36
```

The format specifier `%4d` in `printRow` causes the output to align vertically, regardless of whether the numbers are one or two digits.

Finally, we encapsulate the second loop in a method:

```
public static void printTable() {
    int i = 1;
    while (i <= 6) {
        printRow(i);
        i = i + 1;
    }
}
```

One of the challenges of programming, especially for beginners, is figuring out how to divide up a program into methods. The process of encapsulation and generalization allows you to design as you go along.

7.4 More generalization

The previous version of `printTable` always displays six rows. We can generalize it by replacing the literal `6` with a parameter:

```
public static void printTable(int rows) {
    int i = 1;
    while (i <= rows) {
        printRow(i);
        i = i + 1;
    }
}
```

Here is the output with the argument 7:

```
 1  2  3  4  5  6
 2  4  6  8 10 12
 3  6  9 12 15 18
 4  8 12 16 20 24
 5 10 15 20 25 30
 6 12 18 24 30 36
 7 14 21 28 35 42
```

That's better, but it still has a problem: it always displays the same number of columns. We can generalize more by adding a parameter to `printRow`:

```
public static void printRow(int n, int cols) {
    int i = 1;
    while (i <= cols) {
        System.out.printf("%4d", n * i);
        i = i + 1;
    }
    System.out.println();
}
```

Now `printRow` takes two parameters: `n` is the value whose multiples should be displayed, and `cols` is the number of columns. Since we added a parameter to `printRow`, we also have to change the line in `printTable` where it is invoked:

```
public static void printTable(int rows) {
    int i = 1;
    while (i <= rows) {
        printRow(i, rows);
        i = i + 1;
    }
}
```

When this line executes, it evaluates `rows` and passes the value, which is 7 in this example, as an argument. In `printRow`, this value is assigned to `cols`. As a result, the number of columns equals the number of rows, so we get a square 7x7 table:

```
   1   2   3   4   5   6   7
   2   4   6   8  10  12  14
   3   6   9  12  15  18  21
   4   8  12  16  20  24  28
   5  10  15  20  25  30  35
   6  12  18  24  30  36  42
   7  14  21  28  35  42  49
```

When you generalize a method appropriately, you often find that it has capabilities you did not plan. For example, you might notice that the multiplication table is symmetric; since $ab = ba$, all the entries in the table appear twice. You could save ink by printing half of the table, and you would only have to change one line of `printTable`:

```
printRow(i, i);
```

In words, the length of each row is the same as its row number. The result is a triangular multiplication table.

```
1
2   4
3   6   9
4   8  12  16
5  10  15  20  25
6  12  18  24  30  36
7  14  21  28  35  42  49
```

Generalization makes code more versatile, more likely to be reused, and sometimes easier to write.

7.5 The for statement

The loops we have written so far have several elements in common. They start by initializing a variable, they have a condition that depends on that variable, and inside the loop they do something to update that variable. This type of loop is so common that there is another statement, the `for` loop, that expresses it more concisely.

For example, we could rewrite `printTable` like this:

```
public static void printTable(int rows) {
    for (int i = 1; i <= rows; i = i + 1) {
        printRow(i, rows);
    }
}
```

`for` loops have three components in parentheses, separated by semicolons: the initializer, the condition, and the update.

1. The *initializer* runs once at the very beginning of the loop.
2. The *condition* is checked each time through the loop. If it is `false`, the loop ends. Otherwise, the body of the loop is executed (again).
3. At the end of each iteration, the *update* runs, and we go back to step 2.

The `for` loop is often easier to read because it puts all the loop-related statements at the top of the loop.

There is one difference between `for` loops and `while` loops: if you declare a variable in the initializer, it only exists inside the `for` loop. For example, here is a version of `printRow` that uses a `for` loop:

```
public static void printRow(int n, int cols) {
    for (int i = 1; i <= cols; i = i + 1) {
        System.out.printf("%4d", n * i);
    }
    System.out.println(i);  // compiler error
}
```

The last line tries to display `i` (for no reason other than demonstration) but it won't work. If you need to use a loop variable outside the loop, you have to declare it outside the loop, like this:

```
public static void printRow(int n, int cols) {
    int i;
    for (i = 1; i <= cols; i = i + 1) {
        System.out.printf("%4d", n * i);
    }
    System.out.println(i);
}
```

Assignments like `i = i + 1` don't often appear in `for` loops, because Java provides a more concise way to add and subtract by one. Specifically, `++` is the **increment** operator; it has the same effect as `i = i + 1`. And `--` is the **decrement** operator; it has the same effect as `i = i - 1`.

If you want to increment or decrement a variable by an amount other than `1`, you can use `+=` and `-=`. For example, `i += 2` increments `i` by `2`.

7.6 The do-while loop

The `while` and `for` statements are **pretest loops**; that is, they test the condition first and at the beginning of each pass through the loop.

Java also provides a **posttest loop**: the `do-while` statement. This type of loop is useful when you need to run the body of the loop at least once.

For example, in Section 5.7 we used the `return` statement to avoid reading invalid input from the user. We can use a `do-while` loop to keep reading input until it's valid:

```
Scanner in = new Scanner(System.in);
boolean okay;
do {
    System.out.print("Enter a number: ");
    if (in.hasNextDouble()) {
        okay = true;
    } else {
        okay = false;
        String word = in.next();
        System.err.println(word + " is not a number");
    }
} while (!okay);
double x = in.nextDouble();
```

Although this code looks complicated, it is essentially only three steps:

1. Display a prompt.
2. Check the input; if invalid, display an error and start over.
3. Read the input.

The code uses a flag variable, `okay`, to indicate whether we need to repeat the loop body. If `hasNextDouble()` returns `false`, we consume the invalid input by calling `next()`. We then display an error message via `System.err`. The loop terminates when `hasNextDouble()` return `true`.

7.7 Break and continue

Sometimes neither a pretest nor a posttest loop will provide exactly what you need. In the previous example, the "test" needed to happen in the middle of the loop. As a result, we used a flag variable and a nested `if-else` statement.

A simpler way to solve this problem is to use a `break` statement. When a program reaches a `break` statement, it exits the current loop.

```
Scanner in = new Scanner(System.in);
while (true) {
    System.out.print("Enter a number: ");
    if (in.hasNextDouble()) {
        break;
    }
    String word = in.next();
    System.err.println(word + " is not a number");
}
double x = in.nextDouble();
```

Using `true` as a conditional in a `while` loop is an idiom that means "loop forever", or in this case "loop until you get to a `break` statement."

In addition to the `break` statement, which exits the loop, Java provides a `continue` statement that moves on to the next iteration. For example, the following code reads integers from the keyboard and computes a running total. The `continue` statement causes the program to skip over any negative values.

```
Scanner in = new Scanner(System.in);
int x = -1;
int sum = 0;
while (x != 0) {
    x = in.nextInt();
    if (x <= 0) {
        continue;
    }
    System.out.println("Adding " + x);
    sum += x;
}
```

Although `break` and `continue` statements give you more control of the loop execution, they can make code difficult to understand and debug. Use them sparingly.

7.8 Vocabulary

iteration: Executing a sequence of statements repeatedly.

loop: A statement that executes a sequence of statements repeatedly.

loop body: The statements inside the loop.

infinite loop: A loop whose condition is always true.

program development: A process for writing programs. So far we have seen "incremental development" and "encapsulation and generalization".

encapsulate: To wrap a sequence of statements in a method.

generalize: To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter).

loop variable: A variable that is initialized, tested, and updated in order to control a loop.

increment: Increase the value of a variable.

decrement: Decrease the value of a variable.

pretest loop: A loop that tests the condition before each iteration.

posttest loop: A loop that tests the condition after each iteration.

7.9 Exercises

The code for this chapter is in the `ch07` directory of `ThinkJavaCode`. See page xv for instructions on how to download the repository. Before you start the exercises, we recommend that you compile and run the examples.

If you have not already read Appendix A.5, now might be a good time. It describes Checkstyle, a tool that analyzes many aspects of your source code.

Exercise 7.1 Consider the following methods:

```
public static void main(String[] args) {
    loop(10);
}

public static void loop(int n) {
    int i = n;
    while (i > 1) {
        System.out.println(i);
        if (i % 2 == 0) {
            i = i / 2;
        } else {
            i = i + 1;
        }
    }
}
```

1. Draw a table that shows the value of the variables `i` and `n` during the execution of `loop`. The table should contain one column for each variable and one line for each iteration.

2. What is the output of this program?

3. Can you prove that this loop terminates for any positive value of `n`?

Exercise 7.2 Let's say you are given a number, a, and you want to find its square root. One way to do that is to start with a rough guess about the answer, x_0, and then improve the guess using this formula:

$$x_1 = (x_0 + a/x_0)/2$$

For example, if we want to find the square root of 9, and we start with $x_0 = 6$, then $x_1 = (6 + 9/6)/2 = 3.75$, which is closer. We can repeat the procedure, using x_1 to calculate x_2, and so on. In this case, $x_2 = 3.075$ and $x_3 = 3.00091$. So it converges quickly on the correct answer.

Write a method called `squareRoot` that takes a `double` and returns an approximation of the square root of the parameter, using this technique. You should not use `Math.sqrt`.

As your initial guess, you should use $a/2$. Your method should iterate until it gets two consecutive estimates that differ by less than 0.0001. You can use `Math.abs` to calculate the absolute value of the difference.

Exercise 7.3 In Exercise 6.9 we wrote a recursive version of `power`, which takes a double `x` and an integer `n` and returns x^n. Now write an iterative method to perform the same calculation.

Exercise 7.4 Section 6.7 presents a recursive method that computes the factorial function. Write an iterative version of `factorial`.

Exercise 7.5 One way to calculate e^x is to use the infinite series expansion:

$$e^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + \ldots$$

The ith term in the series is $x^i/i!$.

1. Write a method called `myexp` that takes `x` and `n` as parameters and estimates e^x by adding the first `n` terms of this series. You can use the `factorial` method from Section 6.7 or your iterative version from the previous exercise.

2. You can make this method more efficient if you realize that the numerator of each term is the same as its predecessor multiplied by `x`, and the denominator is the same as its predecessor multiplied by `i`. Use this observation to eliminate the use of `Math.pow` and `factorial`, and check that you get the same result.

3. Write a method called `check` that takes a parameter, `x`, and displays `x`, `myexp(x)`, and `Math.exp(x)`. The output should look something like:

   ```
   1.0     2.708333333333333     2.718281828459045
   ```

   You can use the escape sequence `"\t"` to put a tab character between columns of a table.

4. Vary the number of terms in the series (the second argument that `check` sends to `myexp`) and see the effect on the accuracy of the result. Adjust this value until the estimated value agrees with the correct answer when `x` is 1.

5. Write a loop in `main` that invokes `check` with the values 0.1, 1.0, 10.0, and 100.0. How does the accuracy of the result vary as `x` varies? Compare the number of digits of agreement rather than the difference between the actual and estimated values.

6. Add a loop in `main` that checks `myexp` with the values -0.1, -1.0, -10.0, and -100.0. Comment on the accuracy.

Exercise 7.6 One way to evaluate $\exp(-x^2)$ is to use the infinite series expansion:

$$\exp(-x^2) = 1 - x^2 + x^4/2 - x^6/6 + \ldots$$

The ith term in this series is $(-1)^i x^{2i}/i!$. Write a method named `gauss` that takes `x` and `n` as arguments and returns the sum of the first `n` terms of the series. You should not use `factorial` or `pow`.