

Chapter 5

Recursion

Now that we have fruitful methods and condition statements, you might be interested to know that we have a **complete** programming language, by which I mean that anything that can be computed can be expressed in this language. Any program ever written could be rewritten using only the language features we have used so far (actually, we would need a few commands to control devices like the keyboard, mouse, disks, etc., but that's all).

Proving that claim is a non-trivial exercise first accomplished by Alan Turing, one of the first computer scientists (well, some would argue that he was a mathematician, but a lot of the early computer scientists started as mathematicians). Accordingly, it is known as the Turing thesis. If you take a course on the Theory of Computation, you will have a chance to see the proof.

5.1 Recursion

I mentioned in the last chapter that it is legal for one method to invoke another, and we have seen several examples of that. I neglected to mention that it is also legal for a method to invoke itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical and interesting things a program can do.

For example, look at the following method:

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

The name of the method is `countdown` and it takes a single integer as a parameter. If the parameter is zero, it prints the word “Blastoff.” Otherwise, it prints the number and then invokes a method named `countdown`—itself—passing `n-1` as an argument.

What happens if we invoke the method like this:

```
countdown (3);
```

The execution of `countdown` begins with `n=3`, and since `n` is not zero, it prints the value 3, and then invokes itself..

The execution of `countdown` begins with `n=2`, and since `n` is not zero, it prints the value 2, and then invokes itself..

The execution of `countdown` begins with `n=1`, and since `n` is not zero, it prints the value 1, and then invokes itself..

The execution of `countdown` begins with `n=0`, and since `n` is zero, it prints the word “Blastoff!” and then returns.

The countdown that got `n=1` returns.

The countdown that got `n=2` returns.

The countdown that got `n=3` returns.

And then you’re back to the original invocation of `countdown`. So the total output looks like:

```
3
2
1
Blastoff!
```

The process of a method invoking itself is called **recursion**, and such methods are said to be **recursive**.

5.2 Recursive definition

A recursive definition is similar to a circular definition, in the sense that the definition contains a reference to the thing being defined. A truly circular definition is typically not very useful:

frabjuous: an adjective used to describe something that is frabjuous.

If you saw that definition in the dictionary, you might be annoyed. On the other hand, if you looked up the definition of the mathematical function **factorial**, you might get something like:

$$0! = 1$$

$$n! = n \cdot (n - 1)!$$

(Factorial is usually denoted with the mathematical symbol $!$, which is not to be confused with the Java logical operator $!$ which means NOT.) This definition says that the factorial of 0 is 1, and the factorial of any other value, n , is n multiplied by the factorial of $n - 1$. So $3!$ is 3 times $2!$, which is 2 times $1!$, which is 1 times $0!$. Putting it all together, we get $3!$ equal to 3 times 2 times 1 times 1, which is 6.

5.3 Fruitful recursive methods

If you can write a recursive definition of something, you can usually write a Java program to evaluate it. The first step is to decide what the parameters are for the function, and what the return type is. With a little thought, you should conclude that factorial takes an integer as a parameter and returns an integer:

```
public int factorial (int n) {
}
```

You can add this method to the Buggle class to try it out. Buggles enjoy doing arithmetic in their spare time.

If the argument happens to be zero, all we have to do is return 1:

```
public int factorial (int n) {
    if (n == 0) {
        return 1;
    }
}
```

Otherwise, and this is the interesting part, we have to make a recursive invocation to find the factorial of $n - 1$, and then multiply it by n .

```
public int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        int recurse = factorial (n-1);
        int result = n * recurse;
        return result;
    }
}
```

If we invoke `factorial` with the value 3:

Since 3 is not zero, we take the second branch and calculate the factorial of $n - 1$...

Since 2 is not zero, we take the second branch and calculate the factorial of $n - 1$...

Since 1 is not zero, we take the second branch and calculate the factorial of $n - 1$...

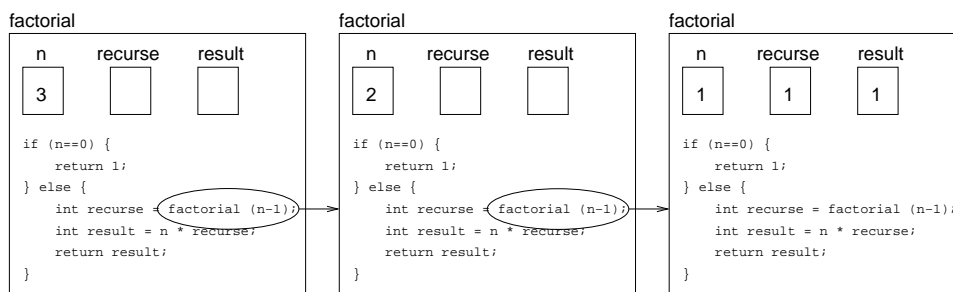
Since 0 is zero, we take the first branch and return the value 1 immediately without making any more recursive invocations.

The return value (1) gets multiplied by n , which is 1, and the result is returned.

The return value (1) gets multiplied by n , which is 2, and the result is returned.

The return value (2) gets multiplied by n , which is 3, and the result, 6, is returned to whoever invoked `factorial` (3).

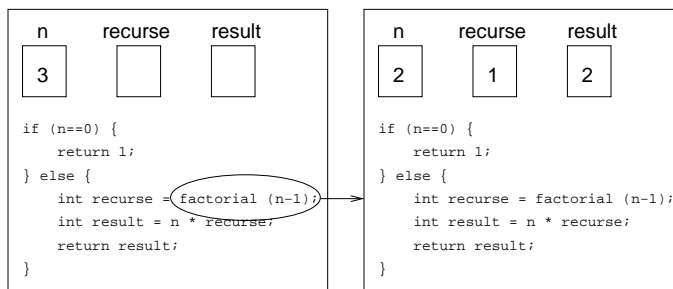
Here is what Execution Land looks like while this recursion is taking place.



This is a snapshot of the point in the execution where the last invocation of `factorial`, the one that got 0 as an argument, has just returned the value 1.

The previous invocation of `factorial` takes that result and multiplies it by n , which is 1, yielding the `result` 1. That's the value it is about to return.

If we come back a little later, we will see this state:



The execution frame that got $n=1$ has completed and vanished. But before it left, it returned the value 1, which was assigned to the variable `recurse`. The execution frame that got $n=2$ can now compute `result=2` and return 2.

As an exercise, draw a JEM for the program state just before the last execution frame disappears.

5.4 Leap of faith

Following the flow of execution is one way to read programs, but as you saw in the previous section, it can quickly become labyrinthine. An alternative is what I call the “leap of faith.” When you come to a method invocation, instead of following the flow of execution, you *assume* that the method works correctly and returns the appropriate value.

In fact, you are already practicing this leap of faith when you use built-in methods. When you invoke a built-in method, you don’t examine its implementation. You just assume that it works, because the people who wrote the Java classes were good programmers.

Well, the same is true when you invoke one of your own methods. For example, as an exercise, write a method called `isSingleDigit` that determines whether a number is between 0 and 9. Once you have convinced ourselves that this method is correct—by testing and examination of the code—you can use the method without ever looking at the code again.

The same is true of recursive methods. When you get to the recursive invocation, instead of following the flow of execution, you should *assume* that the recursive invocation works (yields the correct result), and then ask yourself, “Assuming that I can find the factorial of $n - 1$, can I compute the factorial of n ?” In this case, it is clear that you can, by multiplying by n .

Of course, it is a bit strange to assume that the method works correctly when you have not even finished writing it, but that’s why it’s called a leap of faith!

5.5 Another example

In the previous example I used temporary variables to spell out the steps, and to make the code easier to debug, but I could have saved a few lines:

```
public int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial (n-1);
    }
}
```

From now on I will tend to use the more concise version, but I recommend that you use the more explicit version while you are developing code. When you have it working, you can tighten it up, if you are feeling inspired.

After `factorial`, the classic example of a recursively-defined mathematical function is `fibonacci`, which has the following definition:

$$\begin{aligned} \text{fibonacci}(0) &= 1 \\ \text{fibonacci}(1) &= 1 \\ \text{fibonacci}(n) &= \text{fibonacci}(n-1) + \text{fibonacci}(n-2); \end{aligned}$$

Translated into Java, this is

```
public int fibonacci (int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return fibonacci (n-1) + fibonacci (n-2);
    }
}
```

If you try to follow the flow of execution here, even for fairly small values of `n`, your head explodes. But according to the leap of faith, if we assume that the two recursive invocations (yes, you can make two recursive invocations) work correctly, then it is clear that we get the right result by adding them together.

5.6 Recursion in BuggleWorld

Let's apply what we've learned about recursion to a Buggle problem. What if we would like to tell a Buggle to go forward until it hits a wall? As long as Buggles know when they reach the wall, we can do this with recursion. Fortunately, Buggles provide a method called `isFacingWall` that returns `true` if the Buggle is next to a wall and facing it.

Assume that the Buggle is three steps from the wall, as in the figure:



The solution starts by checking if we are already facing a wall. If so, then the problem is trivial—we can return immediately without doing anything. Otherwise, we should take a single step forward. Here's what the program looks like so far:

```
public void goToWall () {
    if (isFacingWall ()) {
        return;
    } else {
        forward ();
    }
}
```

```

        // now what?
    }
}

```

And then what? Well, we are right back where we started. We want the Buggle to go forward until it reaches a wall. Fortunately, we know how to do that. All we have to do is invoke `goToWall`:

```

public void goToWall () {
    if (isFacingWall ()) {
        return;
    } else {
        forward ();
        goToWall ();
    }
}

```

Strangely, that's all there is to it. As an exercise, draw a JEM for this method when the last execution frame is about to complete.

5.7 Recursion with parameters

By now you have seen `TurtleWorld` and `Turtles` in lecture and laboratory. We can extend the `Turtle` class and add the following method:

```

public void spiral (int steps, int angle, int length, int increment) {
    if (steps==0) {
        return;
    } else {
        fd (length);
        lt (angle);
        spiral (steps-1, angle, length+increment, increment);
    }
}

```

`spiral` takes four integer parameters. The first parameter, `steps` determines how many line segments there are in the spiral. When `steps` is zero, we are done—the method does nothing and returns without making a recursive invocation. This is called the **base case**, because it is the end of the recursion. The other branch of the conditional is called the **recursive step**, since it takes one step toward the base case.

In this case the recursive step draws a line segment with length `length`, turns to the left by angle `angle` and then makes a recursive invocation to draw the rest of the spiral.

The arguments of the recursive invocation are based on the parameters we just received. Since we just took one step, the number of steps remaining is `steps-1`. The angle between segments doesn't change, so we just pass it along.

The value we pass as the new `length` is the old length plus whatever the value of `increment` is. Changing the length from step to step is what makes a spiral. Otherwise, we get a ... well, you can figure it out, or try it by setting `increment` to zero.

It might seem wasteful to pass `angle` and `increment` from one execution frame to the next, since they don't change. But that's the price we pay for modularity.

5.8 Recursion with return values

Suppose we want to create a Buggle that walks to the wall eating bagels as she goes along and returns the number of bagels that she has eaten. How can we do this? Well, if we only had to walk to the wall eating bagels this would be easy. All we would have to do is make the Buggle go to the wall (as we just learned in the previous section) and along the way, if she is over any bagels, have her eat them. So, the new piece of this problem is to have her count the number of bagels as she eats them and return the result.

Here is a method that solves this problem:

```
public int eatBagels () {
    if (isFacingWall ()) {
        if (isOverBagel ()) {
            pickUpBagel ();
            return 1;
        } else {
            return 0;
        }
    }

    if (isOverBagel ()) {
        pickUpBagel ();
        forward ();
        return eatBagels () + 1;
    } else {
        forward ();
        return eatBagels ();
    }
}
}
```

The return type of the method is `int`. It returns the number of bagels that got eaten. As an exercise, draw the JEM for this example.

5.9 Infinite recursion

In all the examples we have looked at, there is a branch in each recursive method that returns without making another recursive call. This branch is the base case, and it is required to make the program finish in a finite amount of time. Without it, the program will recurse forever.

```
public void countdown (int n) {
    System.out.println (n);
    countdown (n-1);
}
```

This version of `countdown` is recursive, but it has no base case. If you run it, you see something like:

```
3
2
1
0
-1
-2
-3
...

```

And eventually you will see a `StackOverflowException`. The **stack** is the structure in the Java run-time system that contains all the execution frames for running methods. In a recursive method, there is one execution frame for every time the method invokes itself.

If a method recurses forever, eventually the stack fills with execution frames and overflows. At that point the program stops running. This situation is called an **infinite recursion**.

In general, to show that a recursive method is correct, you have to show three things:

- There is a base case (and it is correct).
- The recursive step is correct, assuming that the recursive invocation works.
- For all parameters the method might receive, the program will reach the base case in a finite number of steps.

5.10 Glossary

complete language: A programming language that can implement any computable function.

recursion: A way of programming that involves methods that invoke themselves.

recursive: A program that contains one or more recursive methods.

base case: The branch in a recursive function that does not make a recursive invocation, thereby terminating the recursion.

recursive step: The branch in a recursive function that makes one or more recursive invocations.

stack: The structure in the Java run-time system that contains execution frames.

infinite recursion: If a recursion lacks a base case, or the base case is never reached, the program will continue to recurse forever, or until a run-time error occurs.