Chapter 8

Recursive Methods

8.1 Tail Recursion

Let's take another look at the `factorial` method from the book and its stack frames in Figure 8.1.

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

The program can't calculate `factorial(3)` until it has figured out `factorial(2)`, which can't be calculated until `factorial(1)` is calculated, and so on.

This means that when the program gets to the base case, it has to return the result to the previous call, which has to return the result to *its* caller, passing results back up the call chain until the result is finally calculated. This process is indicated by the arrows in the diagram, and this is the process that tends to confuse people when they're learning about recursion.

On the other hand, look at `countdown` method and its stack frames in Figure 8.2.

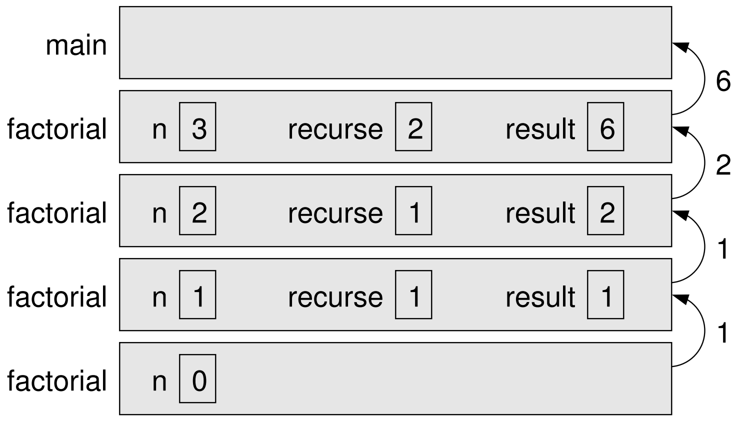

Figure 8.1: Stack diagram for the `factorial` method.

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

When the program hits the base case, (`n == 0`) it's finished. There are no arrows in the diagram because there's no result to pass back to the previous caller; it's a `void` method.

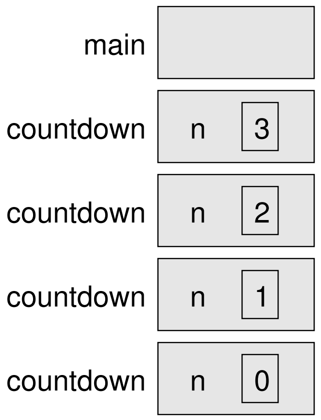

Figure 8.2: Stack diagram for the `countdown` program.

This is why the book introduced recursion with the `countdown` method; it's conceptually simpler.

There is one other important difference between these two methods. In `countdown`, the recursive call is the very last thing that happens in the non-base case. In `factorial`, the recursive call isn't the very last thing that happens. This is why the result has to be "on hold" until reaching the base case.

When the recursive call is the very last thing that happens in the non-base case, the method is called a *tail recursive* method.

Even if you were to rewrite the `factorial` method in the following way, it would still *not* be tail recursive.

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

In this version of the code, the recursive call to `factorial` still isn't the last thing that happens—the multiplication is the last operation.

Is it possible to write methods like `factorial` so that they use tail recursion? Yes, it is, by using something that I call the "accumulator trick." We're going to write the method so that the accumulated result is one of the method parameters:

```
public static int tailFactorial(int n, int result) {
    if (n == 0) {
        return result;
    } else {
        tailFactorial(n - 1, n * result);
    }
}
```

You call it like this:

```
int fac3 = tailFactorial(3, 1); // initial value of result is 1
```

Let's see what happens when we call `tailFactorial(3, 1)`.

> `n` is 3 and `result` is 1. Since 3 is not 0, we call `tailFactorial(2, 3 * 1)`
>
> > `n` is 2 and `result` is 3. Since 2 is not 0, we call `tailFactorial(1, 2 * 3)`
> >
> > > `n` is 1 and `result` is 6. Since 1 is not 0, we call `tailFactorial(0, 6 * 1)`
> > >
> > > > `n` is 0 and `result` is 6. Since 0 *is* 0, we return the value of `result`, which is the correct answer: 6.

Once we hit the base case, we have the answer we want—it's passed all the way back up the chain.

Figure 8.3 shows the call stack for `tailFactorial`. You can think of the base case returning the accumulated result directly to the original caller.

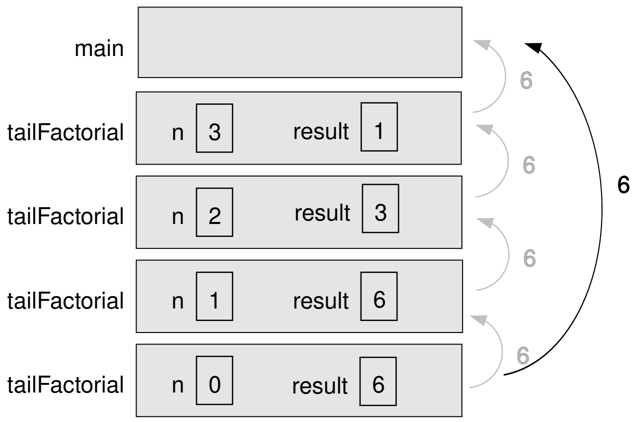

Figure 8.3: Stack diagram for the `tailFactorial` program.

In fact, some languages, but *not* Java, do *tail call optimization.* Since all the necessary information is carried from stack frame to stack frame, these languages can optimize the code so that it re-uses the same stack frame over and over again. With this optimization, stack overflow cannot occur.

Even though Java doesn't do tail call optimization, you should still learn the "accumulator trick." First, it's conceptually simpler: once you hit the base case, you've finished. Second, if you know how to write tail recursive methods in other languages that *do* optimize, you'll be in a position to take advantage of it.

8.2 Overloaded Methods

In the `tailFactorial` example, the user has to provide the number whose factorial they want as well as the starting value for the accumulator. We could make life easier for our users by providing a one-parameter `factorial` method for the convenience of our users:

```java
public int factorial(int n) {
    // Provide the result value
    // for the user's convenience
    return tailFactorial(n, 1);
}
```

They would then be able to write code like this:

```java
int answer = factorial(7);
```

In this example, we have two different method names: `tailFactorial` and `factorial`. Many programming languages *require* you to have different names for all your methods. Java, however, is one of those languages which allow you to have multiple methods with the same name—as long as they have a different number and/or type of their parameters. This is called an *overloaded method.*

While overloading methods indiscriminately can lead to code that is difficult to read and maintain, overloading does have its place. For example, if you look at the documentation for the `Math` class at https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/Math.html, you will see that the `Math.abs` method is overloaded to accept different data types.

We could write the preceding example to use overloading.

```
1 public static int factorial(int n, int result) {
2     if (n == 0) {
3         return result;
4     } else {
5         factorial(n - 1, n * result);
6     }
7 }
8
9 public static int factorial(int n) {
10     factorial(n, 1);
11 }
12
13 public static void main(String[] args) {
14     int answer = factorial(7);
15 }
```

When Java encounters the call on line 14, it sees there is only one parameter and calls the corresponding `factorial` method on line 9. When it gets to the call on line 10, it sees there are two parameters and calls the corresponding `factorial` method on line 1. The call on line 5 has two parameters, which means it is a recursive call to the method on line 1.

8.3 Exercises

Exercise 8.1 Find whether a `String` is a palindrome (the same forward and backwards, such as “radar” or “racecar”). Write an `isPalindrome` method with this header:

```
public static boolean isPalindrome(String s, int start, int end)
```

where `s` is the string you are testing, `start` is the starting index in the string, and `end` is the ending string.

If the characters at the `start` and `end` position are not identical, return `false`; it’s not a palindrome. Otherwise, `return` the result of a recursive call to `isPalindrome`, adding one to the `start` position and subtracting one from the `end` position.

The base case occurs when the `start` and `end` position are the same, which happens for words like “radar“, or when `start` becomes greater than `end`, which happens for words like “anna”. If you reach the base case, return `true`.

As described here, this is a naturally tail recursive process. Either it fails on a mismatch, or you hit the base case and you’ve finished.

Exercise 8.2 Write a tail recursive method to find the sum of an array of integers. To use the accumulator trick in this method, you’ll need to keep track of *two* things: one for the current index into the array and another for the accumulated sum, which starts at 0. The base case occurs when the index equals the array length; at that point you have the result in the accumulated sum. The method header might look like this:

```
public static int sum(int[] arr, int index, int sum)
```

You would call it like this:

```
int[] data = {10, 47, 66, 11};
int total = sum(data, 0, 0);
```

Exercise 8.3 Write a tail recursive method to find the nth Fibonacci number. To use the accumulator trick in this method, you’ll need to keep track of which Fibonacci number you’re working on (*hint*: it counts down), the current result, and the previous result. The method header might look like this:

```
public static int fibonacci(int n, int result, int previous)
```

The base case occurs when `n` is 1 or 2, in which case you return 1.

You would call it like this:

```
// Find the twelfth Fibonacci number
int answer = fibonacci(10, 1, 1); // should be 55
```

Exercise 8.4 In order to make your methods more friendly to users, use method overloading to implement these methods for the preceding exercises:

- `public static boolean isPalindrome(String s)`
- `public static int sum(int[] arr)`
- `public static int fibonacci(int n)`