RECURSION VERSUS ITERATION IN JAVA

You may object: How is this useful? I could just as easily have written the program using a loop!

```
public class Factorial extends Program {
    public void run() {
        int n = 4;
        int result = 1;
        while(n > 0) {
            result *= n;
            n--;
        }
        println(result);
    }
}
```

Indeed, you could. And indeed, most professional programmers would prefer that you did. Thus, while you might acknowledge that the `Mystery` program works, it just doesn't provide any evidence that recursion can be useful. In Chapter 18, we'll see some examples where recursion is indeed the best way to approach a problem. But before looking at those examples, we still have more to do as far as solidifying our understanding of recursion.

But this objection brings up another important point: Recursion and loops are actually related concepts. Generally, anything you can do with a loop, you can do with recursion, and vice versa. Sometimes one way is simpler to write, and sometimes the other is, but in principle they are interchangeable. In fact, some programming languages don't even have any loops (such as Haskell), and other programming languages don't permit recursion (FORTRAN 77). Nonetheless, people manage to write sophisticated programs using them. Most modern languages in wide use, though, take the position that programmers ought to be able to choose which is most appropriate for the problem.

It's useful for us to take some programs using a loop and to see how to rewrite them using recursion. I'd quickly admit that these aren't compelling examples for why recursion is useful, since the programs would be more simply written using a loop. But such examples really are the best with which to start learning about how to write recursive methods.

17.3.1. Reversing a string

We begin with our earlier program that reads a line from the user and displays it in reverse order.

Figure 17.2: The `Reverse` program.

```
public class Reverse extends Program {
    public void run() {
        String str = readLine("Type a string to reverse: ");
        int index = str.length() - 1;
        while(index >= 0) {
            print(str.substring(index, index + 1));
            index--;
        }
    }
}
```

Our goal is to remove the usage of the `while` loop and to replace it with a recursive method. To do this, we'll need to introduce a new method, which we'll call `printReverse`.

```
public class ReverseRecur extends Program {
    public void run() {
        String line = readLine("Type a string to reverse: ");
        printReverse(line);
    }

    public void printReverse(String str) {
    }
}
```

Now the question is how to write the body of this method. To do this, we'll rely on what I'll call the **magical assumption**:

Magical Assumption: Assume that our recursive method already magically works for all smaller instances of the parameter.

In our case, we're writing `printReverse` so that it prints the parameter string `str` in reverse. The magical assumption will be that `printReverse` will somehow work for all strings that are shorter than `str`. In continuing, then, we'll ask: How can we use this assumption to print all of `str` in reverse?

Of course! I hope you respond (or at least you will with some more practice). What we should do is to first print the last letter of `str`, then we apply the magical assumption to`str` with the last letter removed! For example, if we have `str` referring to the string *straw*, our method will first display the last letter *w*, then recursively invoke the method on*stra*. Since *stra* has fewer letters than *straw*, this recursive invocation (says our magical assumption) will displays its reverse *arts*, thus completing the output of *warts*.

This approach translates into the following code.

```
print(str.substring(str.length() - 1));
printReverse(str.substring(0, str.length() - 1));
```

(Incidentally, you might have responded that we should first apply the magical assumption to `str` with the first letter removed (*traw*), then finally to print the first letter (*s*). That is also a valid response, and I'm not going to get caught up arguing which is better.)

But this approach doesn't entirely work: The program is missing a base case. For this, we wonder: What's the smallest possible parameter? Of course, it would be a string with no letters in it at all. And in that case, we don't want to display anything. We use this to build our final program in Figure 17.3.

Figure 17.3: A recursive version of `Reverse`.

```
 3 public class ReverseRecur extends Program {
 4     public void run() {
 5         String line = readLine("Type a string to reverse: ");
 6         printReverse(line);
 7     }
 8 
 9     public void printReverse(String str) {
10         if(!str.equals("")) {
11             print(str.substring(str.length() - 1));
12             printReverse(str.substring(0, str.length() - 1));
13         }
14     }
15 }
```

Note that the test in line 10 tests to see whether the base case does *not* apply. I wrote it this way because, in the case that the base case *does* apply, we don't want to do anything. It would look odd to have an **`if`** statement without anything in its braces and then an **`else`** clause, so instead I inverted the condition: If the string *isn't* empty, then we do the recursion.

Of course, we wrote this thinking solely in terms of our magical assumption, which doesn't immediately convince us that the program will work. But it does.

1. Given the parameter *straw*, the method displays an *w* and invokes itself recursively with the parameter *stra*.
2. That recursive invocation (with *stra* as a parameter) displays an *a* and invokes itself recursively with the parameter *str*.
3. That recursive invocation (with *str*) displays an *r* and invokes itself recursively with the parameter *st*.
4. That recursive invocation (with *st*) displays a *t* and invokes itself recursively with the parameter *s*.
5. That recursive invocation (with *s*) displays an *s* and invokes itself recursively with an empty string as a parameter.
6. That recursive invocation (with an empty string) does nothing.
7. As each of the recursive invocations picks up where it left off, they have nothing more to do.

The overall result is that the program has displayed *warts* as required.

17.3.2. Counting letters

Let's do another example. Suppose I want to count the number of *r*'s in a string typed by the user. (Remember, the useful examples are coming later….) We can do this using iteration easily enough.

Figure 17.4: The `CountRs` program.

```
public class CountRs extends Program {
    public void run() {
        String str = readLine("Type a string to analyze: ");
        int index = 0;
        int count = 0;
        while(index < str.length()) {
            if(str.substring(index, index + 1).equals("r")) {
                count++;
            }
            index--;
        }
        println("There are " + count + " r's.");
    }
}
```

This time, when we convert it to a recursive method taking a string as a parameter, it will be a method that returns an integer. This is so that the `run` method will be able to receive an integer that it can then display.

```
public class CountRs extends Program {
    public void run() {
        String line = readLine("Type a string to analyze: ");
        int count = countRs(line);
        println("There are " + count + " r's.");
    }

    public void countRs(String str) {
    }
}
```

To write the recursive `countRs` method, we again apply the magical assumption: We have a parameter named `str`, but we suppose that any invocation of `countRs` on a string shorter than `str` somehow manages to return the numbers of *r*'s in that shorter string. This leads to an implementation where we examine the first letter of the string to see if it is an *r*, and then use a recursive invocation to count the *r*'s in the remainder of the string.

```
int k = 0;
if(str.substring(0, 1).equals("r")) {
    k++;
}
k += countRs(str.substring(1));
return k;
```

Once again, though, we're missing the base case, which is when the string is empty. In that case, we want to return 0. We conclude with the full, working implementation.

Figure 17.5: A recursive version of `CountRs`.

```
public class CountRs extends Program {
    public void run() {
        String line = readLine("Type a string to analyze: ");
        int count = countRs(line);
        println("There are " + count + " r's.");
    }

    public void countRs(String str) {
        if(str.equals("")) {
            return 0;
        } else {
            int k = 0;
            if(str.substring(0, 1).equals("r")) {
                k++;
            }
            k += countRs(str.substring(1));
            return k;
        }
    }
}
```

17.3.3. Perfect numbers

A positive integer is said to be **perfect** if the sum of its factors (excluding the integer itself) is that integer. For example, 6 is perfect, since the numbers that divide into it exactly are 1, 2, 3, and 6, and the sum of 1, 2, and 3 is itself 6. So also is 28 perfect: Its factors are 1, 2, 4, 7, 14, and 28, and 1 + 2 + 4 + 7 + 14 = 28.

Suppose we want a program to determine whether a number is perfect. We could do it easily enough using a loop.

Figure 17.6: The `Perfect` program.

```
public class Perfect extends Program {
    public void run() {
        int query = readInt("Type an integer: ");
        int index = 1;
        int sum = 0;
        while(index < query) {
            if(query % index == 0) {
                sum += index;
            }
            index++;
        }
        if(sum == query) {
            println(query + " is perfect");
        } else {
            println(query + " isn't perfect: The sum is " + sum);
        }
    }
}
```

But of course, for the sake of practice, we want to write this using recursion instead. We start by writing our recursive method.

```
public class PerfectRecur extends Program {
    public void run() {
        int query = readInt("Type an integer: ");
        int sum = sumFactors(query);
        if(sum == query) {
            println(query + " is perfect");
        } else {
            println(query + " isn't perfect: The sum is " + sum);
        }
    }

    public int sumFactors(int num) {
    }
}
```

But now we hit a brick wall: Try as we might, the magical assumption just doesn't help us. Knowing the sum of the factors up to 1, 2, 3, 4, and 5 just doesn't help with determining the sum of the factors up to 6.

The way over this brick wall is to introduce an additional parameter for our recursive method. This additional parameter will correspond to the `index` variable in our initial loop-based solution.

```
public class PerfectRecur extends Program {
    public void run() {
        int query = readInt("Type an integer: ");
        int sum = sumFactorsTo(query, query - 1);
        if(sum == query) {
            println(query + " is perfect");
        } else {
            println(query + " isn't perfect: The sum is " + sum);
        }
    }

    public int sumFactorsTo(int num, int max) {
    }
}
```

This helps to put us back on track: Given a query of 6, this code will invoke `sumFactorsTo` with two parameters, 6 and 5, with the intent of summing all the factors of 6 between 1 and 5 — or, more generically, given the two parameters `num` and `max`, the method should return the sum of the factors of `num` between 1 and `max`. To do this, we'll first determine the sum of all the factors of `num` between 1 and `max` − 1; we can do this utilizing the magical assumption, since `max` − 1 is smaller than `max`. Then we can add `max` if it itself is a factor of `num` and return that.

Again, though, we need to worry about the base case. As we descend into the recursion, each layer has `max` being 1 smaller than before. Once it reaches 0, we should descend no further: This will be our base case. In this case, there are no numbers between 1 and 0, so we'll return 0.

All the above reasoning is encoded in the program of Figure 17.7.

Figure 17.7: A recursive version of `Perfect`.

```
public class PerfectRecur extends Program {
    public void run() {
        int query = readInt("Type an integer: ");
        int sum = sumFactorsTo(query, query - 1);
        if(sum == query) {
            println(query + " is perfect");
        } else {
            println(query + " isn't perfect: The sum is " + sum);
        }
    }

    public int sumFactorsTo(int num, int max) {
        if(index == 0) {
            return 0;
        } else {
            int sub = sumFactorsTo(num, max - 1);
            if(num % max == 0) {
                sub += max;
            }
            return sub;
        }
    }
}
```

Source : http://www.toves.org/books/java/ch17-recur/index.html