How to Write Recursive Functions on Your Own

In this tutorial, you'll learn - well, you've already read the title - how to write recursive functions on your own. If you're the person to whom recursive functions are a mystery, then this tutorial is *definitely* for you.

Traditional books try to explain recursive functions with a well-known example – finding the factorial of a number. And probably you're desperately trying to find a book where different types of examples are explained, because you're asking yourself – "do we need to learn something so complex just for this nonsense and of-no-use problem of finding factorials?" Grieve not. We're here to the rescue! For the time being, just forget what you already know about recursions. Let's start with a real-life programming problem and try to solve it with simple logics – without using recursion or such hard-to-understand concepts.

Our problem is very simple – we have to print the contents of a folder. However, the complex part of this is, there may be many children of that folder and each may contain again more children folders – we have to print the names of *all* the folders in the hierarchy. To keep things simple, let's assume there are no files contained in any of the folders.

Now, how do we solve it? First, as depicted in *figure 1*, let's assume there are only two children of the given folder. In this case, the given folder is the '/' (root) folder. Now, we can easily write the following code (in Java) to print the names of the folders.

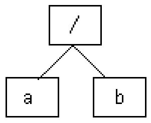

Figure 1

```
 1 import java.io.File;
 2
 3 public class FolderContentsPrint {
 4     public static void main(String[] args) {
 5         File root = new File("/");
 6         File[] folders = root.listFiles();
 7         for (File folder : folders) {
 8             System.out.println(folder.getName());
 9         }
10     }
11 }
```

Well, it was simple enough! Let's try to think of a bit more complex situation – the folder **a** contains two children – **c** and **d**, but the folder **b** doesn't contain anything (*figure 2*). Now, we have to change the program to print the contents of the folder **a** along with the contents of /. The added lines of code are highlighted with red text color.

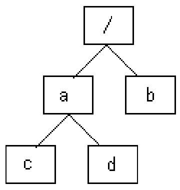

Figure 2

```
 1 import java.io.File;
 2
 3 public class FolderContentsPrint {
 4     public static void main(String[] args) {
 5         File root = new File("/");
 6         File[] folders = root.listFiles();
 7         for (File folder : folders) {
 8             System.out.println(folder.getName());
 9             File[] subfolders = folder.listFiles();
10             for (File subfolder : subfolders) {
11                 System.out.println(subfolder.getName());
12             }
13         }
14     }
15 }
```

So, that's it. However, we can improve the readability of the program a bit. Consider the case of the folder **b**. It doesn't contain any child. So, for that folder, the contents of the inner `for` loop (line 11) simply won't execute as the `subfolders` array won't contain any element. We can include a checking to see whether the `subfolders` array contains any element, and if not, we won't even consider *touching* the `for` loop. The modified program might look as follows (the previously added lines are highlighted with green text color and the newly added lines with red text color):

```
 1 import java.io.File;
 2
 3 public class FolderContentsPrint {
 4     public static void main(String[] args) {
 5         File root = new File("/");
 6         File[] folders = root.listFiles();
 7         for (File folder : folders) {
 8             System.out.println(folder.getName());
 9             File[] subfolders = folder.listFiles();
10             if (subfolders.length > 0) {
11                 for (File subfolder : subfolders) {
12                     System.out.println(subfolder.getName());
13                 }
14             }
15         }
16     }
17 }
```

Okay, let's try to make it a bit more complex again. Now, we assume **c** contains two children named **e** and **f** (*figure 3*). So, we can edit the program again as follows:

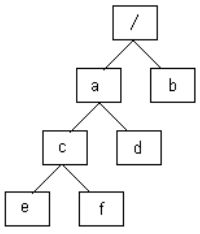

Figure 3

```
 1 import java.io.File;
 2
 3 public class FolderContentsPrint {
 4     public static void main(String[] args) {
 5         File root = new File("/");
 6         File[] folders = root.listFiles();
 7         for (File folder : folders) {
 8             System.out.println(folder.getName());
 9             File[] subfolders = folder.listFiles();
10             if (subfolders.length > 0) {
11                 for (File subfolder : subfolders) {
12                     System.out.println(subfolder.getName());
13                     File[] subSubfolders = subfolder.listFiles();
14                     if (subSubfolders.length > 0) {
15                         for (File subSubfolder : subSubfolders) {
16                             System.out.println(subSubfolder.getName());
17                         }
18                     }
19                 }
20             }
21         }
22     }
23 }
```

Enough! Are we going to add lines of code for each folder level??? It's totally absurd. We must find a simpler solution which would work for any level of folders. The question is – *how*?

Let's take a closer look at the above program. Did you notice that lines 7-10 has been repeated in lines 11-14 with just a few changes in the names of variables used? If we were going to add more lines for the next level of folders, we would repeat those same lines again. So, how can we get rid of this situation, i.e., eliminate the repeating lines of code? There are two possible solutions – we can either execute a loop several times, or call a procedure for some specific number of times. Let's try to experiment with the second solution. We can come back to the first one later.

So, we have to put the repeated lines of code into a function. Then let's do it – just put those lines into a function with a meaningful name – e.g. `printFolders` – and don't bother with the function parameters and return types for now. The function should look like the following:

```
1 printFolders() {
2     for (File folder : folders) {
3         System.out.println(folder.getName());
4         File[] subfolders = folder.listFiles();
5         if (subfolders.length > 0) {
6 
7         }
8     }
9 }
```

We have to fill up *three* gaps in the above function to make it a *complete* function – the parameters, return type and the contents of the `if` statement (i.e., line 6). First, let's try to find out the possible parameters of the function. To find them out, we have to dig into the code and try to detect any *undeclared* variables. Yep, you've got it – the variable `folders` is undefined. So, let's define the `folders` variable as a parameter. Wait a minute though – what about the *type* of this variable? If we look at the `for` loop, we can find that it is a collection – in this case, an array of `File` objects. Now, the return type. If we observe the code, we can come to the conclusion that it doesn't return anything. All that the function does is to print the names of the contents of the folder. Therefore, the above function with return type and parameters should look like below:

```
1 void printFolders(File[] folders) {
2     for (File folder : folders) {
3         System.out.println(folder.getName());
4         File[] subfolders = folder.listFiles();
5         if (subfolders.length > 0) {
6 
7         }
8     }
9 }
```

Now remains the final part – the contents of the `if` statement. As we've already learned, the lines 2-8 in the above code will be repeated inside the `if` statement. But the `printFolders()` function contains all the necessary repeating statements. Therefore, we can simply call the `printFolders()` function inside the `if` statement. However, this function needs an argument of type `File[]` to be passed when called. What should it be in this case? The `subfolders` variable, of course. So, the complete function will look like the following:

```
1 void printFolders(File[] folders) {
2     for (File folder : folders) {
3         System.out.println(folder.getName());
4         File[] subfolders = folder.listFiles();
5         if (subfolders.length > 0) {
6             printFolders(subfolders);
7         }
8     }
9 }
```

To make the function a bit more readable, we can replace the parameter with just a `File` object. We can list the contents of that `File` object *inside* the function:

```
1     void printFolders(File root) {
2         System.out.println(root.getName());
3         File[] folders = root.listFiles();
4         if (folders.length > 0) {
5             for (File folder : folders) {
6                 printFolders(folder);
```

```
            }
        }
    }
```

Our final program (which prints the contents of a given folder *along with* the name of that given folder) should look somewhat like below:

```java
import java.io.File;

public class FolderContentsPrint {

    public static void main(String[] args) {
        File root = new File("/");
        FolderContentsPrint obj = new FolderContentsPrint();
        obj.printFolders(root);
    }

    void printFolders(File root) {
        System.out.println(root.getName());
        File[] folders = root.listFiles();
        if (folders.length > 0) {
            for (File folder : folders) {
                printFolders(folder);
            }
        }
    }

}
```

Done! You've written a recursive function! Now, before we formally state the steps for writing a recursive function, let's try to grasp the process better.

At this point, we'll examine the traditional factorial problem. The following program demonstrates how to find the factorial of 5 using *loop*:

```java
public class Factorial {
    public static void main(String[] args) {
        int number = 5, result = 1;
        for (int i = number; i > 0; i--) {
            result = i * result;
        }
        System.out.println(result);
    }
}
```

The above program can be re-written using a `while` loop instead of a `for` loop:

```java
 1 public class Factorial {
 2     public static void main(String[] args) {
 3         int number = 5, result = 1;
 4         while (number > 0) {
 5             result = result * number;
 6             number--;
 7         }
 8         System.out.println(result);
 9     }
10 }
```

There is no logical difference between these two versions of the program. Now, we'll try to write the program using recursion. First, let's focus on our target – obviously, it's to find out the repeating lines of code. If we look at the `while`-loop version of the program, we can easily detect that lines 5 and 6 are the targeted lines of code. (In case of the `for`-loop version of the program, however, it's not obvious at first sight that `i--` is also a part of the repeating statements; that's why I also supplied the `while`-loop version.) Therefore, according to our experience from the previous problem (printing the contents of a given folder), we have to put these two lines into a function – say, `getFactorial()`. We'll be ignoring the return type and parameters for the time being.

```
getFactorial() {
    result = result * number;
    number--;
}
```

Now, can't we determine the parameters of this function? Of course! The variables result and number are not defined. So, they must be the parameters. Okay, then what about the return type? It seems that the function returns nothing. So, the return type should be `void`. The function then becomes as follows:

```
void getFactorial(int result, int number) {
    result = result * number;
    number--;
}
```

So, where are we going to call the function recursively? It should be after the 3rd statement, something like below:

```
void getFactorial(int result, int number) {
    result = result * number;
    number--;
    getFactorial(result, number);
}
```

Great! But hey, it's going to recursively call the function *infinite* times! So, we must provide a *condition* on which the recursion will stop. What is it then? If we observe the `while`-loop version of the program, we can understand that the loop will continue *while* `number > 0`. So, our function should be calling itself *as long as* it finds that `number > 0`. The modified version of the function is as follows:

```
void getFactorial(int result, int number) {
    result = result * number;
    number--;
    if (number > 0) {
        getFactorial(result, number);
    }
}
```

Have we completed writing our recursive function for finding the factorial of a number? Sorry, the answer is 'no'. Why? Because, we need to get the value of `result`. As `result` is a local variable, when the function will return, its value will be destroyed. However, among the different possible solutions, we'll prefer *one* solution (as it's conceptually the easiest) – make the variable global. Our function might look like the following then (the global variable isn't shown here):

```
void getFactorial(int number) {
    result = result * number;
    number--;
    if (number > 0) {
        getFactorial(number);
    }
}
```

Our complete program should be somewhat similar to the following program:

```java
public class Factorial {
    int result = 1; //the global variable [known as instance variable in java]

    public static void main(String[] args) {
        int number = 5;
        Factorial obj = new Factorial();
        obj.getFactorial(number);
        System.out.println(obj.result);
    }

    void getFactorial(int number) {
        result = result * number;
        number--;
        if (number > 0) {
            getFactorial(number);
        }
    }
}
```

However, to make the program more efficient and professional, it can be thoroughly changed as follows:

```java
public class Factorial {

    public static void main(String[] args) {
        int number = 5;
        Factorial obj = new Factorial();
        System.out.println(obj.getFactorial(number));
    }

    int getFactorial(int number) {
        if (number > 0) {
            return (number * getFactorial(number - 1));
        } else {
            return 1;
        }
    }

}
```

Notice that the above program is conceptually advanced than our previous program. When you become proficient in developing recursive functions, then you'll be able to develop *efficient* recursive functions on your own.

Our experiment with recursive functions is done. Now, finally, let's take a look at the steps of writing a recursive function:

1. Determine the repeating statements in your program.
2. Just put those statements into a function with a suitable name.
3. Determine the parameters of the function by finding out all *undeclared* variables used inside it.
4. Determine the return type of the function by observing its behavior. In preliminary stages, you'll often find that it is of type `void`. But when you become proficient, you'll notice that by bringing some changes into the body of the function, you can make the function return something and thus make it more efficient and standard-looking.
5. Insert the recursive call to the function into an appropriate place within the function body.
6. Determine and add the necessary condition(s) to stop the function calling itself recursively infinite times.
7. Make any necessary changes to the function to make it more efficient and professional.

Remember that you can convert loops into recursive functions and vice-versa.

One final point before we conclude: Using recursive functions instead of loops doesn't make your program run more efficiently. Function calls are always expensive as various variables and register values are pushed-popped into and from the stack when a function is called or returned from. Then why on earth would we use recursive functions at all?! The answer is simple – to make your program conceptually easier to understand.

So? What are you waiting for? Get your hands dirty by messing with recursive functions! NOW!!!

Sharafat Ibn Mollah Mosharraf
12th Batch (2005-2006),
Dept. of Computer Science & Engineering,
University of Dhaka.
E-mail: sharafat_8271@yahoo.co.uk
Home Page: www.sharafat.info
Blog: http://blog.sharafat.info