

Recursion

Lecturer: Arran Stewart

Outline

We discuss:

- ▶ What is recursion?
- ▶ What are some examples of recursive programs?

Calling methods

- ▶ In Java, instructions for the computer to carry out are contained in *methods*.
- ▶ For instance, the insertion sort code we looked at last week is contained in the method:

```
public static void insertionSort(short[] arr)
```

in the class `SortingAlgorithms`.

Calling methods

- ▶ And we also have seen that methods can call other methods.
- ▶ For instance, in our array-based implementation of a stack, the `pop()` method looked like this:

```
public void pop() {  
    if (isEmpty()) {  
        throw new Underflow( "Stack empty so can not pop" );  
    }  
    topOfStack = topOfStack - 1;  
}
```

- ▶ The `pop()` method here calls the `isEmpty()` method.

Recursive methods

- ▶ However, a method can also call *itself*.
- ▶ This behaviour is known as **recursion**.
- ▶ Recursion is an extremely powerful technique for expressing some complex programming tasks
 - ▶ This is because it provides a very natural way to break problems down into smaller parts
- ▶ However, there are costs associated with recursion we need to be aware of.

Recursive methods

A simple example:

```
public static void forever() {  
    System.out.println("hello world");  
    forever();  
}
```

How will this behave when run?

Ending recursion

A method that runs forever is not (usually) very useful.

Consider this example:

```
public static void myMethod(int i) {  
    if (i <= 1) {  
        System.out.print("finishing");  
        return;  
    } else {  
        myMethod(i - 1);  
    }  
}
```

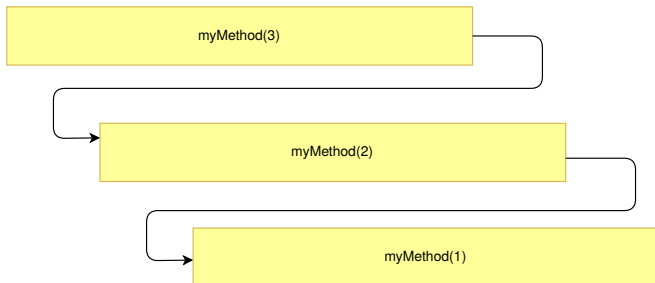
How will this behave if we call it as follows: `myMethod(3)`?

Local variables

- ▶ Each recursive call to the method creates a “new copy” of the method - parameters and all local variables are created again
- ▶ The compiler keeps track of them all and ensures they do not get mixed up.

What happens in the method call?

We can imagine the recursive method calls looking something like this:



Final method call

- Eventually, the `i <= 1` condition will be satisfied, and the method will not call itself again, but instead end.

```
public static void myMethod(int i) {  
    if (i <= 1) {  
        System.out.print("finishing");  
        return;  
    } else {  
        myMethod(i - 1);  
    }  
}
```

Factorial example

- ▶ The factorial function is defined as follows:

$$n! = n(n-1)(n-2)\dots(1)$$

- ▶ Can we write a recursive method to calculate it?

factorial method

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

We use long just to
give the method
slightly greater range

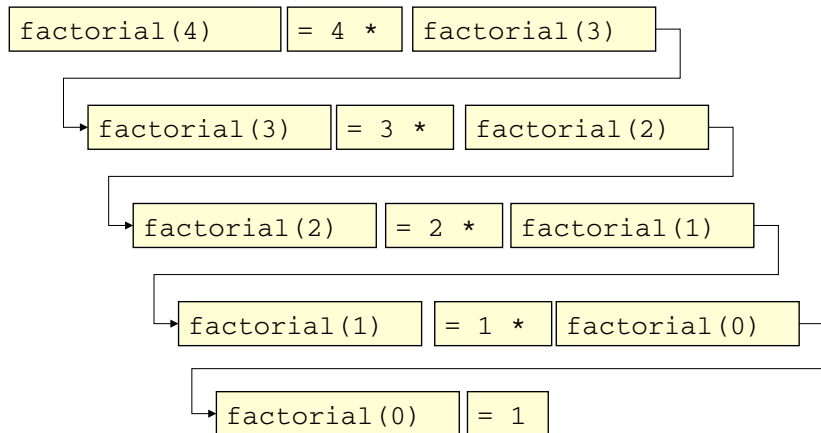
The method factorial calls *itself*,
but with a smaller argument

factorial method behaviour

How does the factorial method behave, when called with, say, 4 as an argument?

factorial method behaviour

How does the factorial method behave, when called with, say, 4 as an argument?



local variables in factorial

In this case –

- `factorial(4)` has a parameter `n` equal to 4

 - `factorial(3)` has its own parameter `n`, equal to 3

 - `factorial(2)` has its own parameter `n`, equal to 2

 - `factorial(1)` has its own parameter `n`, equal to 1

local variables in factorial

When using Java – each of these local variables *does* take up space in memory when run.

So that means, if the method will be called thousands or millions of times, it may not be wise to use recursion – we may run out of space.

(But note that not all languages have this problem – some are especially designed to work well with recursive functions.)

iterative version

It is possible to write what is called an *iterative* version of factorial. Instead of using recursion, we use a for loop:

```
public long factorial(int n) {  
    long result = 1;  
    for(int i = 1; i <= n; i++) {  
        result = result * i;  
    }  
    return result;  
}
```

Any method we can write using recursion, we *can* also write using loops and iteration.

iteration vs recursion

- ▶ So why write methods using recursion?
- ▶ One reason is that the methods can often be written more simply that way. (We will see this when we discuss *trees*.)
- ▶ Which looks simpler?

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

```
public long factorial(int n) {  
    long result = 1;  
    for(int i = 1; i <= n; i++) {  
        result = result * i;  
    }  
    return result;  
}
```

iteration vs recursion

- ▶ So why write methods using recursion?
- ▶ One reason is that the methods can often be written more simply that way. (We will see this when we discuss *trees*.)
- ▶ Which looks simpler?

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

```
public long factorial(int n) {  
    long result = 1;  
    for(int i = 1; i <= n; i++) {  
        result = result * i;  
    }  
    return result;  
}
```

- ▶ The difference may not seem much for a small method.
 - ▶ For more complicated methods, however, it can become significant.

Ingredients for recursive definition

Recall our simple recursion example:

```
public static void myMethod(int i) {  
    if (i <= 1) {  
        System.out.print("finishing");  
        return;  
    } else {  
        myMethod(i - 1);  
    }  
}
```

- ▶ Every recursive method definition needs two parts - the *base case* and the *recursive part*

Base case

```
public static void myMethod(int i) {  
    if (i <= 1) { // base case  
        System.out.print("finishing");  
        return;  
    } else { // recursive part  
        myMethod(i - 1);  
    }  
}
```

- ▶ The base case does *not* call the method again.
- ▶ Instead, it describes how the method should behave for some specified parameters.
- ▶ Typically, the base case represents some sort of “trivial” case ...
 - ▶ e.g., dealing with the `int 0`; or, for lists, an empty list; etc.

Recursive part

```
public static void myMethod(int i) {  
    if (i <= 1) { // base case  
        System.out.print("finishing");  
        return;  
    } else { // recursive part  
        myMethod(i - 1);  
    }  
}
```

- ▶ The recursive part describes how the method should behave in terms of another call to the same method – but with different parameters.
 - ▶ Why “different parameters”? Because if it passed the *same* parameters, the method would behave in exactly the same way – and the recursion would never end.

Recursive part

```
public static void myMethod(int i) {  
    if (i <= 1) { // base case  
        System.out.print("finishing");  
        return;  
    } else {      // recursive part  
        myMethod(i - 1);  
    }  
}
```

- ▶ The recursive part describes how the method should behave in terms of another call to the same method – but with different parameters.
 - ▶ Why “different parameters”? Because if it passed the *same* parameters, the method would behave in exactly the same way – and the recursion would never end.
- ▶ To work correctly, the “different parameters” must be *closer* to the base case (in some sense)
 - ▶ In this case – myMethod(3) calls myMethod(2); and 2 is closer to the base case (1).

Ordering

Order is important! Don't do this:

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

If we write the factorial method like this, the recursive part will get executed – and the method will never reach the base case!

In Java, this means a *stack overflow* error will occur.

Further examples

- ▶ We will see a further example of recursion when we look at *binary search*.