Recursion

3/22/21

Reduction

- If we have a large problem to solve, we can break it into small pieces/subroutines

```
public void newCarPosition(int roadLength){

        int x = Canvas.random(roadLength);
        // doing more stuff with x

}
```

```
(inside Canvas)
public int random (int x){
        //Doing something
}
```

Reduction

- Earlier in the year we didn’t know how random worked but we didn’t care - we trust it to work and just use its result
- Letting other methods do simpler work for us makes writing the more complex newCarPosition method a lot easier

```
public void newCarPosition(int roadLength){

        int x = whatever the method returns when it’s done
        // doing more stuff with x

}
```

```
(inside Canvas)
public int random (int x){
        //Doing something
}
```

Reduction

You can keep calling methods within one another to solve more and more sub-parts of the problem:

```
public void someMethod()
{
        // stuff
        int r1 = anotherMethod();
        // more stuff
        // and more stuff
}
```

```
public int anotherMethod()
{
        // stuff
        int result = oneMoreMethod();
        // do something with result
        return result;
}
```

```
public int oneMoreMethod()
{
                // stuff
                // no more methods
                // more stuff
                return some int;
}
```

Code here doesn’t get called until the oneMoreMethod() call finishes running (and if it doesn’t you have a problem)

This would be a really simple computation, so you don’t need to break it down further

Recursion

Recursion is a particularly powerful kind of reduction, which can be described loosely as follows:

- If the given instance of the problem can be solved directly, solve it directly.
- Otherwise, reduce it to one or more simpler instances of the same problem.

If the self-reference is confusing, it may be helpful to imagine that someone else is going to solve the simpler problems, just as you would assume for other types of reductions.

Recursion

```
public int someMethod(int x){
        // stuff
        int x = someMethod(int y);
        // more stuff
        // and more stuff
}
```

We call the same method, this time with simpler inputs - breaking down the original problem into a sub-problem, with easier inputs

This makes it easier for us to work around the method output

Example: Factorials

Factorial (n!) : product of all the numbers 1...n. So

4! = 4 * 3 * 2 * 1

Example: Factorials

4! = 4 * 3 * 2 * 1

Notice:

4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1!

1! = 1 (by definition) - this is so simple you don’t have anything else to do

In code: trace through a factorial method, public static int fact(int x)

4! → call fact(4)

```
public static int fact(int 4){
    if(4 == 1){
        return 1;
    }
    else{
        return 4*fact(3);
    }
}
```

Now we have to evaluate this in order to have something to return

```
public static int fact(int 3){
    if(3 == 1){
        return 1;
    }
    else{
        return 3*fact(2);
    }
}
```

```
fact(4){
    return 4*fact(3);
}
```

Now we have to evaluate this in order to have something to return

```
public static int fact(int 2){
    if(2 == 1){
        return 1;
    }
    else{
        return 2*fact(1);
    }
}
```

```
fact(4){
    4 * fact(3){
        3* fact(2);
    }
}
```

Evaluate this in order to have something to return

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

We are done calling methods - we can’t get any simpler than this (“base case”)

```
fact(4){
    4 * fact(3){
        3 * fact(2){
            2 * fact(1)
        }
    }
}
```

fact(1) = 1

```
fact(4){
    4 * fact(3){
        3 * fact(2){
            2 * fact(1)
        }
    }
}
```

fact(2) = 2

```
fact(4){
    4 * fact(3){
        3 * fact(2){
            2 * 1
        }
    }
}
```

fact(3) = 6

```
fact(4){
    4 * fact(3){
        3 * 2


        }
    }
}
```

fact(4) = 24

```
fact(4){
    4 * 6
}
```

fact(4) = 24

```
fact(4){
    return 24;



}
```

We’re done!

code

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

This is the base case - when writing recursive methods start thinking about what the base case should be and then how to get there

This is the recursive part of it (the recursive call)