9.6 Guidelines for Methods

Like anything in programming deciding how to break a problem into methods takes practice. However, here are some guidelines.

- **Repeated Code** – this is an obvious case and a definite *code smell*: if your program has repeated code then it makes sense to put it in a method. It's easy to think this is the only reason to have methods - it isn't.
- **Single Task** – *methods should do one thing*: keep them simple and focused on solving a single problem. If you find yourself saying 'this is a method to do X and then Y', make it *two* methods (i.e *refactor* it) – one to do X, and one to do Y.
- **Keep Small** – methods should be *short*: a rule of thumb is a maximum of 25 lines of about 80 columns of text. You need to be a bit flexible and sometimes you need to go over this: but in general, methods that are bigger than this get to be hard to understand (and smaller is fine). Many successful programmers write *really* short methods - only a few lines long - most of the time. So consider *refactoring* your code into smaller methods.
- **Multiple Return Statements** - methods can have more than one return statement *but* doing this *can* make the code harder to read (essentially for the same reason that **break** and **continue** do). So be careful!
- **Use Parameters** – a method is easiest to understand if it only deals with information that is supplied by parameters because that makes it self-contained and easier to understand. Try to avoid what are commonly called 'global' variables (using parameters will also make your methods more flexible).
- **Not Too Many Parameters** – this slightly conflicts with the point above; but don't overdo the parameters: if you get more than about five then your method is probably too complicated so consider breaking into smaller ones (even if it's already quite short).
- **Logical Breakdown** – as you think about the problem you are trying to solve, and break it into parts, you can't go far wrong if you start by making each logical part a method at least as a starting point.

These are simple rules of thumb, and sometimes you'll have to break them; also as you get more experienced, you will see more cases where you should break them. But these simple rules apply most of the time to most problems.

Commenting Methods

Something I haven't done much here (oops…) but which I should is to put a brief comment at the start of each method saying what it does. For example:

```
//Method to compute the square of an integer
static int square(int num) {
     return num * num;
}
```

Common Mistake from Coursework

Suppose I ask you to write a method to compute how long it would take for an amount of money to double, given a specified interest rate, where the amount and interest rate are specified by a use – what would you do? Based on what gets written in coursework, lots of people would do this **– which is wrong!**

```
/*
Not a very good method because it violates the "do one thing" rule - should be separate
method(s) to read in the numbers
*/
static double doubleMoney() {

    Scanner in = new Scanner(System.in);

    System.out.print("Enter amount: ");
    double amount = in.nextDouble();

    System.out.print("Enter interest rate: ");
    double rate = in.nextDouble();

    final double TARGET = amount * 2;
    double count = 0;

    while (amount < TARGET) {
        amount += amount * rate;
        count++;
    }
    return count;
}
```

The reason this is wrong is because it does *two* things – not one.

- It reads in data from the user.
- It computes the time taken for the value to double.

Why is this bad? *Suppose the user input does not come from the keyboard* – but a file, or a web page, or a network connection. The method is now *useless* and we need to write another one. We should try to make our code as re-useable as possible – it may not matter for a simple method like this, but a lot of code is very complex, and very expensive to write. So make sure you write methods that do only one thing. *The correct way to write this is:*

```
static double doubleMoney(double amount, double rate) {

          final double TARGET = amount * 2;
          int count = 0;

          while (amount < TARGET) {
                   amount += amount * rate;
                   count++;
          }
          return count;
 }
```

How do we get the input? We could write a *separate method* for this:

```java
static double getValue(String message) {
          System.out.print(message);
          Scanner in = new Scanner(System.in);
          while(!in.hasNextDouble()){
                   System.out.print("Must be a number!");
                   in.nextLine();
          }
          double val = in.nextDouble();
          in.nextLine();
          return val;
 }
```

We can use this to get an double value, with a message we specify:

```java
double time = doubleMoney(getValue("Enter amount: "),
                        getValue("Enter rate: "));
System.out.println(time);
```

Instead of one method that can only be used for one specific thing, we now have two methods that can each be used more flexibly - we call one twice to get two different double values (using a different message each time) and then call our doubleMoney method passing them as parameters. You may find this example easier to understand written out using separate variables - this does the same as the one above:

```java
double amount = getValue("Enter amount: ");
double rate = getValue("Enter rate: "));
double time = doubleMoney(amount, rate);
System.out.println(time);
```

KEY POINT: Methods should do ONE Thing

When you write a method it should do **ONE thing**. If when you describe what a method does you find yourself using the word 'and' it's quite likely you should have written more than one method instead.