9. Methods

***Note**: some example programs are from, or based on, examples from Java for Everyone (C Horstmann), the course text.*

One thing you may have noticed in your solution to the first coursework is that you are repeating (or nearly repeating) chunks of. It would be nice to only write this code once. Also, even if code is not repeated, programs can get complex and it would be nice to be able to break them into smaller chunks, or *subprograms,* which are easier to manage and understand. Obviously, since there's a chapter on it☺, a way of doing this exists. Also we've seen some examples already.

- `System.out.println("Lemon sorbet");` - prints Lemon Sorbet to the screen.
- `int value = in.nextInt();` reads in an integer from the keyboard.

In fact, *all* the things we've seen that end in `(..)` – are examples of these kinds of subprograms.

Names – Procedures, Functions and Methods

If you look at the two examples above you can see that they are conceptually different – the first one (`System.out.println(…)`) just goes away and does something – prints out whatever is in the brackets; the second one (`in.nextInt`) does something and *returns* a value (in this case an integer) to the program that called or invoked it.
Historically, the first kind of subprogram is called a *procedure* and the second kind a *function* (and is closely related to *mathematical* functions). However, in object oriented programming languages like Java we just call them all *methods*.

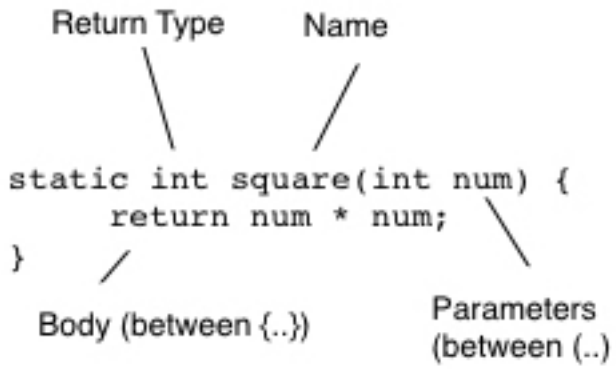
Simple Example – the Square Method

Suppose we are writing a program that needs to compute the squares of integers many times and we want to write a method to do it:

```
static int square(int num) {
     return num * num;
}
```

The first line defines the *signature* of the method – what it's called, the type of data that it *returns*, and the types of the *arguments* or *parameters*: that is, the data that it will work with. In the example above, the method is called square, it returns and `int`, and it has one argument/parameter, which is an `int` (`int num`). As well as the type of the argument, it also has a name (`num`) – this is used inside the body of the method to refer to the value of the parameter.

As this method returns a value, we must use the keyword return to specify what that value is – in this case, `num*num`. The various parts are labelled below.



§

KEY POINT – Static

Notice that we've used the word `static` at the start – I'm not going to explain this yet but, for now, *you need to put it in* (try taking it out and see what the compiler says).

We call the square method like this:

```
int squareValue = square(5);    // squareValue == 25
System.out.println(square(3)); // prints 9

int val = 4;
int result;
result = square(val);           // result == 16

int newVal = 6;
result = square(val + newVal); // result now == 100
```

As you can see it's quite flexible – we can:

- Use numbers (*literals*) like 4 directly as arguments.
- Use variables as arguments (note: they do *not* need to have the same name as the one we use inside the method definition).
- Use more complex expressions (like `a + b`) as arguments.
- Assign the result to a variable.
- Use it directly – for example, print it out.

Here's a complete program to show you where the method definition goes.

```
import java.util.Scanner;

class SquareExample {
     public static void main(String[] args) {
          Scanner in = new Scanner(System.in);

          System.out.print("Number? ");

          while(in.hasNextInt()) {
               int squareValue = in.nextInt();

               int result = square(squareValue);
               System.out.println("The square of "
                    + squareValue +
                    " is " + result);

               System.out.print("Another number " +
                    (anything else to quit)? ");
          }
     }

     //Method goes outside main but inside class
     static int square(int num) {
          return num * num;
     }
}
```

This version uses two `int` variables – one for the argument and one for the result. It keeps printing out the squares of numbers as long as the input is an integer. But we can actually simplify the while loop by calling the method in the print statement:

```
          while(in.hasNextInt()) {
               int squareValue = in.nextInt();

               System.out.println("The square of "
                  + squareValue +
                  " is " + square(squareValue));

               System.out.print("Another number " +
                    (anything else to quit)? ");
          }
```

Another Example – returning Yes or No

Recall that we've seen some code that repeatedly asks a user to enter a response until they say yes or no. You can imagine that you could need to do that a lot in a program, meaning it's an ideal choice for a method:

```
static String yesOrNo() {
     Scanner in = new Scanner();
     String inValue;

     do {
          System.out.print("Enter yes or no: ");
          inValue = in.nextLine();
     while(!inValue.equalsIgnoreCase("yes") &&
           !inValue.equalsIgnoreCase("no"));
     return inValue;
}
```

Notice this one – as well as being much more complex – has *no* parameters. This is OK – *but we still need to put the brackets* in even though there's nothing between them. The way we call this is:

```
String result = YesOrNo();
```

or

```
do {
     //some stuff
while(YesOrNo().equals("yes"));
```

Notice that our principle of D.R.Y. – Don't Repeat Yourself – is being violated here and we should have constants (final variables) for 'yes' and 'no'. In fact a better version might be:

```
static String yesOrNo(String stopVal, String contVal) {
     Scanner in = new Scanner();
     String inValue;

     do {
          System.out.print("Enter " + stopVal + " or " +
                           contVal + ": ");
          inValue = in.nextLine();
     while(!inValue.equalsIgnoreCase(contVal) &&
          !inValue.equalsIgnoreCase(stopVal));

     return inValue;
}
```

and then do something like this when we call it:

```
final String CONT_VAL = "yes"; //or "ja", or "ναί", or…
final String STOP_VAL = "no";  //or "nein", or "όχι", or…

do {
     //some stuff
while(YesOrNo(STOP_VAL, CONT_VAL).equalsCONT_VAL));
```

Notice also that this version has *two* arguments – and they are written like this:

```
typeName argumentName, typeName argumentName,…
```

the name of the type, then the name of the argument, then a "," - for as many arguments (of different types, or the same types) as you need.

Methods that Don't Return Results

The first example we showed above (`System.out.println`) didn't actually return anything – it just *does* something. In some languages sub-programs that return results and those that don't are separate things – but in Java, those that don't return results are just a special case: instead of putting in a return type like `int` or `String`, we use the special 'no return value' type (or keyword) `void`. For example:

```
static void sayHello(String name) {
     System.out.println("Hello " + name + "!");
}
```

Notice the word `void` in place of the return type names we used in the examples above. Also notice that there is no `return` statement – because it doesn't return a result.
To call this method, we can do:

```
sayHello("Mr President");

String name = "Dave";
sayHello(name);

String forename = "Darth";
String surname = "Vader";
sayHello(forename + " " + surname);
```

As before you can see that there are lots of options for how you construct the arguments – they can be constants, variables, or operations based on constants and/or variables.

Parameters and What You Can/Can't Do

Let's go back to our first example

```
static int square(int num) {
     return num * num;
}
```

Now we've seen void methods, it seems we could change it to this:

```
static void square(int num) {
     num = num * num;
}
```

That is, instead of returning the new value, we're *changing* (or trying…) the value of the parameter we passed in. Does this work? Here's a complete program with both methods (we've changed the name of one because you can't have two methods with the same name in this case):

```
class SquareTests {
     public static void main(String[] args) {
          //The one that returns an int
          int number = 5;
          int squareVal = square(number);
          System.out.println("The 'returning' method: "
             + squareVal);

          //The one that tries to change it's argument
          //Try to change number to number*number
          squareVoid(number);
          System.out.println("The 'void' method: "
             + number);

     }

     static int square(int num) {
          return num * num;
     }

     static void squareVoid(int num) {
          num = num * num;
     }
}
```

The result you should get from this is:

```
The 'returning' method: 25
The 'void' method: 5
```

That is, the second void method does *not* change the value of the parameter. This is because parameters are *copies* – *not* the original variables. Try changing the program by putting a `println` statement into the void method:

```
static void squareVoid(int num) {
     num = num * num;
     System.out.println("Inside 'void' method: " + num);
}
```

You should see:

```
The 'returning' method: 25
Inside 'void' method: 25
The 'void' method: 5
```

As you can see the value of `num` *inside* the method *does* change – but because it's a *copy*, and that copy is *thrown away* after the method finishes, the original parameter *does not change*. If you think about it, if the value of a parameter could change outside the method then you couldn't do things like:

```
squareVoid(5);
```

because this would mean trying to change the value of 5 to 25 – which doesn't make sense. On the other hand:

```
int result = square(5);
```

does – because it returns a *new* value and doesn't try to change the parameter. What actually happens when you execute:

```
squareVoid(5);
```

is:

- A copy is made of the value 5;
- that value is copied into a new variable `num` inside the method;
- that new variable is squared and becomes 25;
- if we are using the version which has the `println` in it, that new value of 25 is printed out;
- if the method returns a value, that value is copied back into whatever variable it's going to be stored in;
- finally, the new variable is thrown away when the method ends.

KEY POINT: Parameters are Copies

When you pass a parameter to a method, a copy is made and the method works on the copy. This means you cannot change the value of simple types like `int, double, boolean` which are passed as parameters – you can only return new values. It *specifically says simple types* in the previous

sentence because the situation with *complex types* – like `ArrayLists` – is more complex as we'll see below.

Advanced Aside: Call by Value

The way parameters are passed in Java is known as *Call by Value* – because it is the *value* that is passed, *not* the actual variable. This is the simplest parameter passing mechanism, and the way Java does it, the only one needed. But there are other ways of doing it:

- *Call by Reference* – the actual variable is passed, so you can change it. Java doesn't do this but has a way of 'simulating' it (see later).
- *Call by Value-Result* – the parameter is passed as a copy, but then the value is copied back out to the original variable. Java doesn't do this (most languages don't). In most cases this behaves the same as Call by Reference except in some obscure cases.

Passing Complex Data

We specifically said about that you cannot change the value of parameters outside methods, and we showed an example. We also said that things were not so simple with more complex data. Here's an example:

```
import java.util.ArrayList;
import java.util.Arrays;

class ChangeArrayList {
     public static void main(String[] args) {
          //Create an ArrayList quickly from an array
          ArrayList<Integer> list =
               new ArrayList<>(Arrays.asList(1, 2, 3));

          //Call method to add 1 to all items in the list
          addOne(list);

          //Print out the list
          for(int val : list) {
               System.out.println(val);
          }
     }

     //Method to add 1 to elements of an ArrayList
     static void addOne(ArrayList<Integer> valList) {
          for(int i = 0; i < valList.size(); i++) {
               valList.set(i, valList.get(i) + 1);
          }
     }
}
```

(As an aside, notice how we quickly populate an `ArrayList` with a set of values.)
If you run this you'll find the output is 2, 3, 4 – but we put 1, 2, 3 in the `ArrayList` – so obviously we've managed to change it.

Here's a contradictory example:

```
import java.util.ArrayList;
import java.util.Arrays;

class ChangeArrayList {
     public static void main(String[] args) {

          //Create the list
          ArrayList<Integer> list =
               new ArrayList<>(Arrays.asList(1, 2, 3));

          //Call method to delete it completely
          zapArrayList(list);

          //Print it out
          for(int val : list) {
               System.out.println(val);
          }

     }

     //Method to set ArrayList to null to delete it
     static void zapArrayList(ArrayList<Integer>
        valList){
          valList = null;
     }
}
```

This one has a method that *explicitly* sets the `ArrayList` passed in as an argument to *null* – i.e. it basically deletes it. And yet, after the method returns the contents of the `ArrayList` are still present.

The Answer? Complex Objects are *References*

If you didn't read it the first time, now go back to *Chapter 4 – Conditionals*, and read the section **Strings and Equals Danger: Skip on First Reading if New to Programming** and especially look at **Figure 1.** Here's a version of that picture that will help:

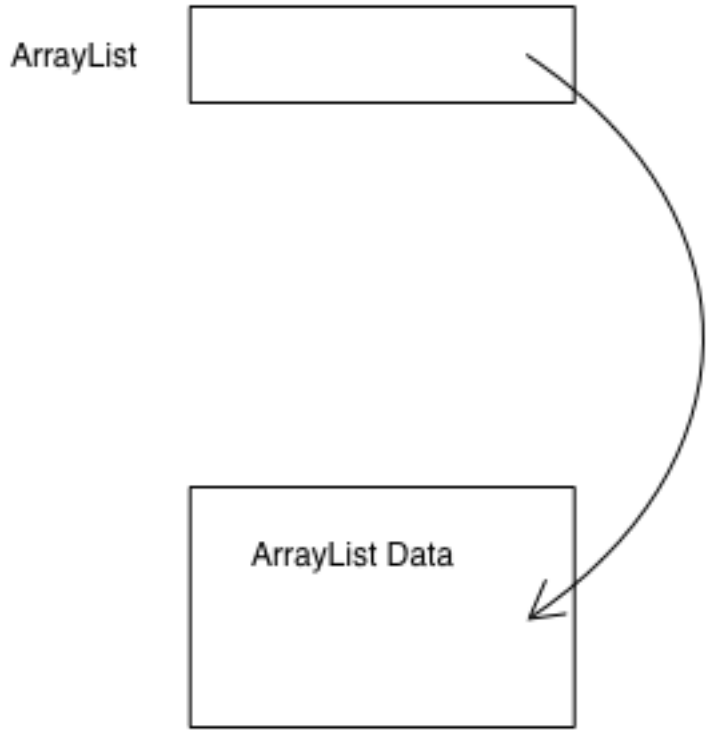


Figure 1: References to Data

Complex data like an `ArrayList` is not stored in the same way as simple data. Instead the actual variable is a *reference* to the real data, which is elsewhere in memory. This is done for a perfectly good reason, which I'm happy to explain to anyone who wants to know, but is a bit off topic here. What this means is that *the reference is the value that is passed* – it cannot be changed (which why `zapArrayList` didn't work) *but the data it refers to can be changed* (which is why `addOne` *did* work). Here's another picture to help:

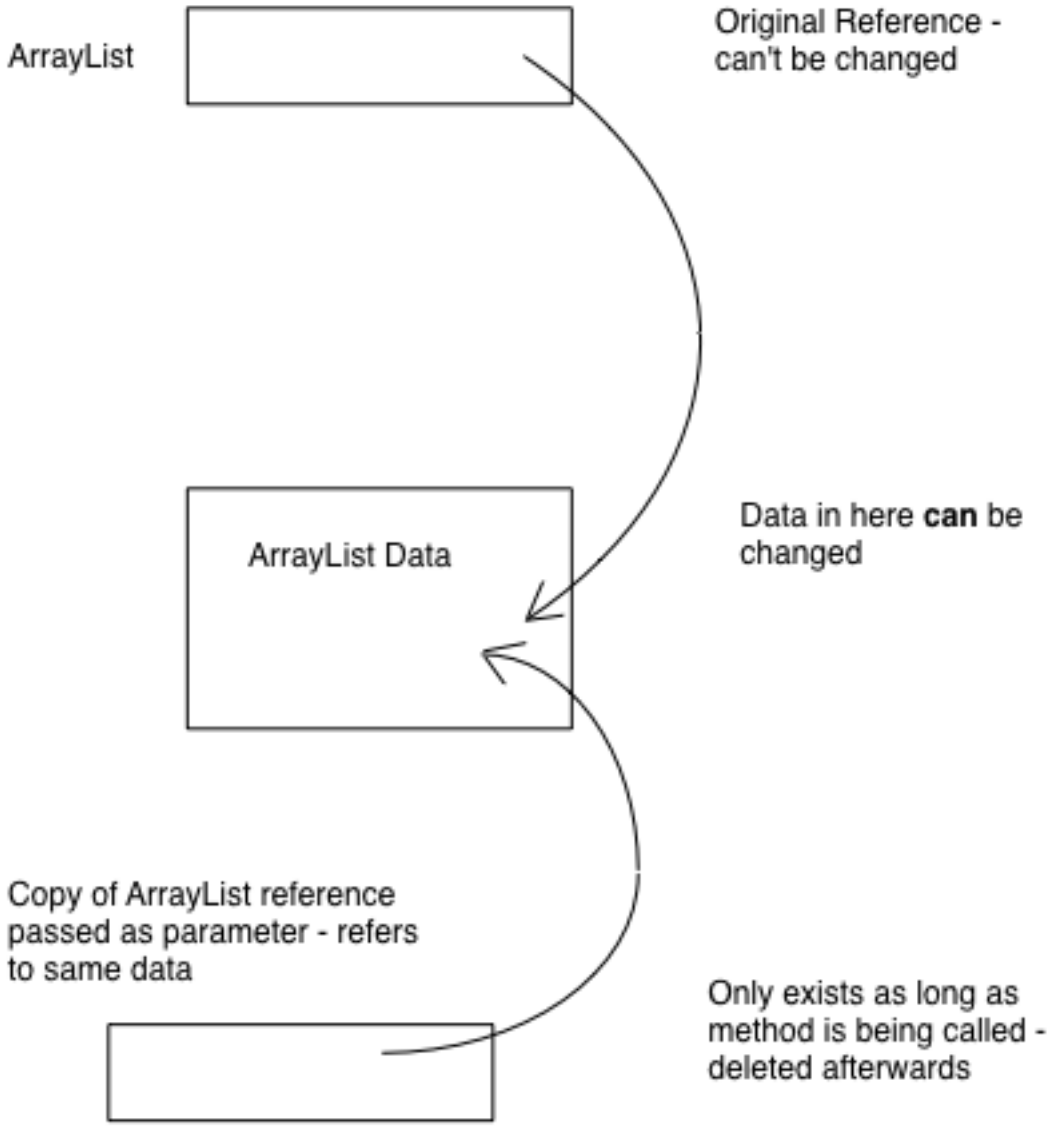


Figure 2: Original Reference and Parameter

When we pass something like an `ArrayList` as a parameter, make a copy of the reference, and pass that reference – but the reference still points to the same data. Whatever changes we make to the copy of the reference (by, say, setting it to null) get discarded when the method ends. But we can make changes to the actual data that the reference copy 'points' to.

The main Method

Hopefully after reading this, the line you've been writing for weeks now should make a bit more sense:

```
public static void main(String[] args)
```

We still need to deal with public and static, but:

```
void main(String[] args)
```

should now be clear – this is a method, called `main` with one argument which is an array of Strings. The convention in Java (and other languages like C, C++ and C# - most widely-used modern languages) is that when you run a program, the system looks for a method called `main` and starts the program there. The `main` method is just an ordinary method like any other, and it's possible to pass in parameters as an array of Strings. Here's a program that deals with parameters:

```
class SquareArgs {
     public static void main(String[] args) {

          if (args.length == 0) {
               System.out.println("Supply a number");
          } else {
               int value = Integer.parseInt(args[0]);
               System.out.println("Square " +
                 square(value));
          }
     }

     static int square(int num) {
          return num * num;
     }
}
```

The if statement first checks to see if there are any arguments because if `args.length==0` then the array args is empty and the user hasn't typed any values in. Otherwise we:

- Assume that the first thing they've typed is the argument and ignore anything else they might have written after it (`args[0]` – remember that arrays in Java start with element zero).
- Users `int.parseInt` to turn the string into an integer – this will crash if the user types a non-integer, which is not great but this is meant to be a simple example.

The way we 'call' the `main` method with parameters is slightly different to the way we would call it within a program and we don't use brackets:

```
java SquareArgs 5
Square 25
```

if we leave off the arguments:

```
java SquareArgs
Supply a number
```

if we type extra numbers, they are ignored:

```
java SquareArgs 7 3
Square 49
```

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: if your program has repeated code then it makes sense to put it in a method.
- **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 – 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.
- **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.
- **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 part a method.

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 & Exams

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 exams and sometimes coursework, lots of people would do this **– which is wrong!**

```
static int 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;

     int 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 int 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:

```
static double getValue(String message) {

     System.out.print(message);
     Scanner in = new Scanner(System.in);

     while(!in.hasNextInt()){
          System.out.print("Must be a number!");
          in.nextLine();
     }
     double val = in.nextDouble();
     in.nextLine();
     return val;
}
```

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

```
int 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.