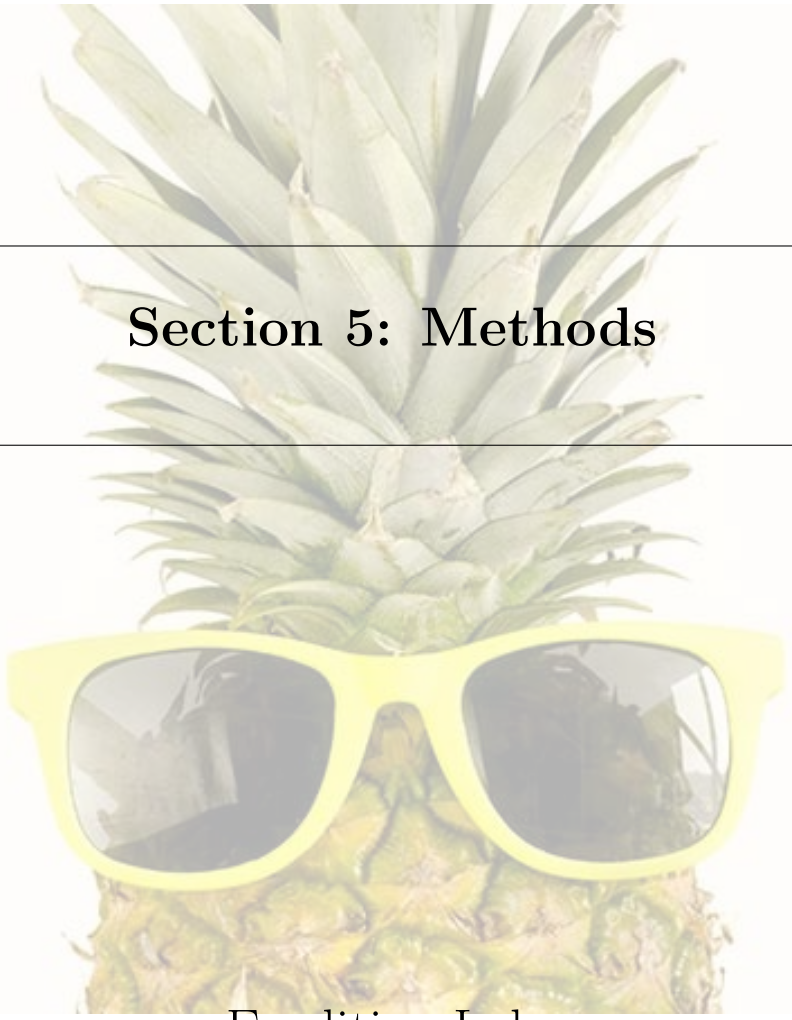

Section 5: Methods

Erudition Labs

Computer Science 101: Introduction to Java and Algorithms

May 17, 2019

Contents

1 Terminology

- ***Method call (aka call a method)*** – This simply means that we want to use a method that we wrote or that someone else wrote. We want to to execute the code with the parameters we give it and give us back the value.
- ***Pass parameter*** – Often when we define a method, we also define parameters that the method needs for it to do its work. When we actually call the method, we often say that we are passing parameters to it. We are talking about the values that we give to the method when we call it.
- ***Function (or method* Signature** – The method signature is what the compiler remembers when we define a method. When we overload methods, the only thing the compiler looks at is the method signature. This is made up of the return type, the method name and the types of the parameters. For example, int calcArea(int x, int y), the method signature would be int calcArea(int, int). If we were to simply change the parameter names, the method signature would be the same.
- ***Go out of Scope*** – In this section we define scope. When we go out of scope, are going out of the block where that variable is defined. Once that happens, that variable no longer exists in memory and cannot use it anymore.
- ***Globals*** – Often use this to refer to global variables.

2 Pre-Chapter

2.1 Memory Revisited – Stack

For this Chapter it will be helpful to know a bit more about the Stack (stack memory). It is a LIFO data structure. LIFO stands for "Last in First Out". All this means is that the last thing that you put onto the stack will be the first thing that you take off of this stack. This is helpful for maintaining a top-down order when returning from method calls.

2.2 Modifier Keywords (Access Control – public)

So far you have seen two modifier keywords, static and final. Final we introduced in a different chapter and static you have just seen around with the main method. We will talk about static in the next subsection. Here i'd like to introduce "Access Control Modifiers". These keywords do exactly what they sound like they do, they allow you to specify who has access to your variables. There are four levels, public, private, protected and just nothing. By nothing I simply mean how you have been declaring variables this whole time, like "int a = 5;" These variables are package private, meaning they can be accessed by pretty much anything within a package. "Private" is only accessible from within classes and "protected" can be accessed

from within the class and its subclasses. What does all this mean??? Don‘t worry about it too much yet as we will be going back to all of those as well as what a “package“ is later.

The main point of this subsection is simply to introduce the term “access modifier“ and let you know what the “public“modifier does. As said before, public allows anyone to access your variable or method or whatever. When you you write code that other people will use, they will be using your public methods and variables. This keyword allows you to define who is allowed to access what.

For example, you have been using the public modifier this whole time, just look at main.

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

Granted, you have no choice but to make your main method like this, but it‘s like that for a reason. Main is your main entry point to the program, so it makes sense that anyone should be able to access it, including the operating system.

2.3 static

This keyword is considered a modifier, but not an access modifier. The static keyword is mainly used for memory management with variables, methods, blocks etc. It‘s hard to talk too much about this until we get to classes and objects. All you need to know about static right now is when applied to methods. Just know that static methods belong to the class and can modify static variables. The thing to really know for this section is that static methods cannot call non-static methods and cannot use non-static data. This simply means that, for now, all of your methods need to be static. If, however, you would like to learn more now, i‘d recommend <https://www.javatpoint.com/static-keyword-in-java#staticm>

3 Methods (Video Series Lecture 28 and 29)

3.1 What is A Method and Why Use It?

This is a major concept in programming. In Java we use methods and functions interchangeably. A method is a means of writing code once and writing it well. If we have some piece of code that we are going to be using multiple times in a multiple places, it would be worth it to just write it once and use it in those places. The idea is that, the more we have to write, the higher the chances we have of making a mistake. But if we write that piece of code as a single, self-contained unit, then we only need to write it once. If we make mistakes, then the mistakes are only in that one place. As opposed to copying and pasting those pieces of code all over the place and having to fix all of them when you find a problem.

Methods allow you to design a piece of code that does one thing, and does it very well. This allows us to have better designed, more easily maintainable and more secure code.

3.2 Methods

Methods take the form of

```
modifier returnType nameOfMethod (Parameter List) {
   // method body
}

//for example, as seen in the video series
public static void printVal() {
    int a = 11;
    System.out.println("Value = " + a);
}
```

Your can read about modifiers in subsection 2.2. The return types are all the types that we talked about in the first chapter such as int, String, boolean, double etc. This is the type of value that we are going to return from the method (we often refer to this as the return type). Then we have the name of the method. Just like naming a variable, you have to give the method a name for you to reference later. Finally, we give a list of parameters to the method. These are the values that we want the method to use when it executes its code. Note that the parameters are usually copies of the variables that you pass in. So when you method executes, it is dealing with copies of your data instead of the original values. This means that when you execute the method, anything you do to those values does not affect the original values.

If we look at the example, we have "public static void printVal()" The method is accessible by anyone, returns nothing and has no parameters. Let's look at a method that you have used over and over.

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

}
```

Yes that's right, the main method. The main method is the main entry point of the program so it makes sense that it is public for everyone to use. It returns nothing and takes parameters of type String. The parameter that we pass for main is a String array variable called "args" (short for arguments, aka command line arguments. Don't worry about what this means yet).

3.3 Parameters

Parameters are variables that we give a method to do some work with. For all of the primitive types, the method actually makes a copy of the parameters that we pass in. This means that we can mess with those parameters without having to worry about the original variables. So if we want to make an affect on the program, we can return the result of the work done on those copies.

You can pass as many variables as you would like, but long method signatures are discouraged because they can be complex, hard to read or just plain too long. Generally 3 to 4 variables are the maximum. Anything more and you would pass the parameters using an array or on object. We will cover objects in the next chapter.

4 Scope

4.1 Block Scope

Note that we cannot access the parameters from outside of a method itself because of what we call scope. Scope has to do with how long variables exist in memory. If we just kept them all in memory then we would eventually run out of memory. So it is important that we clear out things that we don't need anymore. One such way that we do that is with scope resolution. Basically anything defined between curly braces will exist until we exit the curly braces. We call this block scope.

Methods are an example of block scope. We define a method and everything in that block exists until the method returns. It turns out we can actually define blocks where ever we want. So first, lets look at method scope.

```
public static void main(String args[]) {
   int x = 4;
   int y = 5;
   System.out.println(calcArea(x, y));
}

public static int calcArea(int x, int y) {
 return x * y;
}
```

In this method we have two parameters, x and y. In the main method, we call the calcArea method. When

this happens, we actually pass in copies of the original x and y. The method parameters x and y cannot be accessed outside of the calcArea method. But why?

Well to figure this out, we need to look at what happens to the stack (stack memory). When a method is called, the program creates what we call a stack frame. A stack frame is just a collection of variables and information about the method being called, as well as how to get back to the rest of the program when the method is done. So when a method is called, the return address is pushed on to the stack. The return address is just the address of the next code to be executed once the method returns. This way, the program can pick up where it left off. Then the stack frame is pushed on to the stack. Now the stack frame consists of the parameters and the space needed to store those parameters. Once the method is done and returns, the stack frame is popped off of the stack, (aka taken out of memory) so those variables and the spaces no longer exist. In essence, the parameters and their space in memory only exist within that stack frame and thus there is no way to access it from out side of that frame.

Now recall from when we talked about switch statements. If you were to just google it, you would find something in the form of

```
switch(expression) {
  case x:
    // if x true do this
    break;
  case y:
    // if y true do this
    break;
  default:
    // if none are true do this
}
```

However, I mentioned that I prefer to write it this way,

```
switch(expression) {
  case x:
  {
    // if x true do this
  } break;
  case y:
  {
    // if y true do this
```

```
  } break;
  default:
  {
    // if none are true do this
  }
}
```

It is by no means mandatory, but I do it to enforce block scope on the different cases. One of the mistakes that I often catch people doing is that they use the first way, but the declare some variable in a case statement and then use it later in some other case statement without realizing that it is already some value. Using the extra curly braces as blocks for each case enforces the fact that you can only use variables defined in that case statement within that case. This prevents us from accidently overwriting variables or use the wrong values when we define them in the other cases. We say that these variables are local to that block. In the case of methods, we say that the parameters and the variables that are defined inside that method are local to that method.

4.2 Global Scope

We also have global variables. These variables are accessible from everywhere in the program. However, many programmers have deemed global variables as evil because too many of them become confusing and hard to maintain. The fewer globals the better. These types of variables are declared outside of the main method at the top of the program.

5 Returning

You'll notice when we define out methods that we must include the return type. So far we have pretty much used the "void" return type for everything, which just means we don't return anything. This is good enough for simple programs, but once we start creating larger programs, it becomes confusing and inefficient to write code this way. Since parameters are only copies of the data we want to work with, in order to make actual change to your data, you would have to define a bunch of variables, make void methods to operate on them and then store them all. We would have to define every variable we would ever need, which would take up all our memory. It simply isn't feasible to write programs using only void methods and global variables. So what can we do about it? We can have the method return a copy of the solution and then just free up the memory that the method was using.

6 Method Overloading

Sometimes we need to create methods that do something similar but we may have different parameters depending on the situation. You can methods of the same name as long as the method signature is different. I think the best way to describe this is with an example. The popular example is simply a function that calculates the sum of it's parameters.

```
class Main {
    //This sum takes two int parameters
    public static int sum(int x, int y) {
        return (x + y);
    }

    //This sum takes three int parameters
    public static int sum(int x, int y, int z) {
        return (x + y + z);
    }

    //This sum takes two double parameters
    public static double sum(double x, double y) {
        return (x + y);
    }

    //sum of however many numbers we are given
    public static int sum(int[] nums) {
    int sum = 0;
      for(int i=0;i<nums.length; i++) {
         sum += nums[i];
      }

      return sum;
    }

    public static void main(String args[])
    {
        System.out.println(sum(10, 20));
        System.out.println(sum(10, 20, 30));
        System.out.println(sum(10.5, 20.5));
        System.out.println(sum(new int[] {4, 5, 7, 23, 55, 100}));
```

```
    }
}
```

In this program, we would like to be able to handle integers and decimal numbers. So to do so, we create methods that handle integers, decimals and an array of integers. These methods all give the same result, aka the sum, but deal with different data and calculate that sum differently. This becomes convenient for naming and for when someone else has to use your code. We use this quite a bit when working with classes which we will cover in the next chapter.

7 Good Practices

7.1 Naming

In Java we tend to do things in camel case. That means the for methods, we start the name of the method with a lower case and then capitalize any other words. So for example, "thisIsCamelCase". Use descriptive names to name your methods but also don't make the names too long. It should be easy to remember and tell the programmer exactly what it does.

7.2 Parameters

There is no limits to how many parameters you have for a method, but it's a good idea to have as few as possible.