Training Application Developer

A2 - Programming curriculum

Organize code in Java

Author: *Harmen Steenbergen*

Date: *12-5-2023*

Crebo: *25187*

Version: *2.1*

Table of contents

Overview 43
..... 4

Prerequisites 43
..... 4

Materials 43
..... 4

Sources 43
..... 4

Instructions 43
..... 4

Description 54
..... 5

Goals 54
..... 5

Review score 54
..... 5

Study units 65
..... 6

Use of a method 65
..... 6

Exercises 76
..... 7

Arguments 76
..... 7

Exercises 98
..... 9

Static 109
..... 10

Exercises 1110
..... 11

Scope of a variable/method 1211
..... 12

Rules 1312
..... 13

Exercise 1413
..... 14

Recursion 1514
..... 15
Final assignments 1817
..... 18
Assignment 1: Calculate formulas 1817
..... 18
Assignment 2: Clean up string of characters 1918
..... 19
Sources 2019
..... 20

Overview

Level:Domain A Level 2
Duration:80 hours Method:Weekly schedule

Prior knowledge

A1 Introduction to programming in Java.

Materials

- Your laptop with;
- A working Java IDE

Sources

- See Appendix Sources

Instructions

- Use of methods/functions to structure code
- Use of arguments to reuse methods
- Scope of variables, when do you need which scope.
- Recursion

Description

Goals

After studying this module:

- you are able to write "neat" code.
- Can you use methods/functions
- Can you explain why variables are sometimes reachable and sometimes not
- Can you use recursive methods

Assessment

- After completing this module, you have made 3 hand-in assignments. You must submit these assignments via magister.
- The teacher will give an oral work made about you.
- The assignments and the oral form your final grade for this module.

.

Study components

Use of a method

So far in Java you have only written a piece of code in the main method. This is the method that indicates the starting point of a program in Java. In a Java project, there is a class that is started, in which it searches for the method as below:

```
Public Static Void main(String[] args) {

}
```

If your program becomes more extensive and you would put all the code in the main method, this will be a very long and confusing code. The first thing you're going to do to fix this is create separate methods for pieces of code.

Example

```
public class ConsoleApp {

   public static void main(String[] args) {
      Int Dice;
      Dice = roll dice();
      System. out.println(Dice);
   }

   private static int gooiDobbelsteen() {
      Return (Int)(1 + 6 * Math. Random());
   }

}
```

Explanation code

Within the *ConsoleApp* class , a second method is placed that mimics a roll with a die and returns the result of the roll.

- The *Math.random()* gives a random comma number between 0 and 1.
- By multiplying this number by 6 and increasing it by 1 you get a number from 1 to 7 (but not 7 itself)
- After that, the result is typecast to an integer, leaving only the number before the comma (1 to 6).
- The return command returns this number as a result of the function.

The definition of the method states *private static int*:

- ***private*** means that the method exists only within the class in which it stands. So *roll Dice* may only be used within the *ConsoleApp* class.

- ***static*** specifies that this method may be used without first creating an object from the class in which the method is located. This method must be *static* because the method it uses, the main method, is also *static*. Later, we'll adjust that by using multiple classes.
- ***int*** indicates the return type of the method. In this case, the result of the method is an integer, i.e. an integer. As a result, the method may return only an integer value, and the method must return a value of type integer. Without the correct return statement, you will see an error message and you will not be able to compile your code. If you want a method to be does but returns no result, use *void* English for emptiness (nothing) as a return type.

Exercises

For the following exercises you can watch a video:

Click here for the video!

Exercise 1

Create a method named *makeSquare* which returns a String. The string contains a square of 5 x 5 pluses. When you print it on the console it looks like:

```
+++++
+++++
+++++
+++++
+++++
```

Exercise 2

Create a method named evenDay that returns whether today's date is an even number, result is `true` or `false` (boolean).

From the Calendar object you can get today's date.

Exercise 3

Create a method named getOsInfo that extracts the name, version, and architecture of the operating system (OS) from the system properties. As a result, you get a String containing the requested info separated by a space.

Tip: https://docs.oracle.com/javase/tutorial/essential/environment/sysprop.html

Arguments

The dice example executes the code and returns the result. No information is given when the method is called, but you often want that. You can do this by using arguments.

Example 1

```
Public .class ConsoleApp {

    Public Static Void main(String[] args) {
      Int example = 10;
      Int Result = increaseWithA(example);
System.  out.println(example);
System.  out.println(Result);
    }

    private static int increaseWithA(Int input) {
      Return ++input;
    }

}
```

Explanation code

The *increaseWithA* method uses the value that is passed and increments it by 1, then the result is returned.

- ++input increases the value in input by 1 before the value is used. This is different from input++, because there the value of input is used first and only then increases by 1. In this example, input++ would cause the method to return the value given in the *input* argument . Try that for yourself.
- (int input) indicates that this method waits for an integer value to be passed on the call. This value is stored within the method in the variable input. You can only use this variable within the method.

Example 2

```
Public .class ConsoleApp {

    Public Static Void main(String[] args) {
      Int a = 10;
      Int b = 5;
  Int sum = telOp(a,b);
System.  out.println(sum);
    }

private static int  telOp(int  number1,  int getal2) {
    return number1 + number2;
    }
}
```

Explanation code

The *appraise* method adds the two arguments together and returns the result (the sum).

- (int number1, int number2) the arguments are separated by a comma. Each argument is preceded by the type, in this case both int.

Exercises

Exercise 4

Do exercise 1 again, but now you should be able to give the size of the square as an argument to the method.

Exercise 5

Create a method named getMax that takes an array of integer values as an argument and returns the largest value from the list.

Exercise 6

Create a method called countChar that takes a String and a Char as arguments. The method counts how many times the character in the second argument (char) appears in the first argument (String) and returns the value.

Exercise 7

Create a method named reverseString that takes a String as an argument. The result of the method is the input in reverse order, so 'exercise' becomes 'gninefeo'.

Static

We have always used *static* in the examples because otherwise the main method cannot use the method. Ultimately, we want to have as few *static* methods as possible . The main method can be seen as a small starting point of your program that only creates the objects to keep the program running. That is why we are going to create a second class in the project and use a method from it.

Example

```
ConsoleApp.java
Public .class ConsoleApp {

   Public Static Void main(String[] args) {
      Dice Dice = New Dice();
      Int Dice;
      Dice = dice.roll();
      System. out.println(Dice);
   }
}
```

```
Dice.java
Public .class Dice {

   Public Int throw() {
      Return (Int)(1 + 6 * Math. Random());
   }
}
```

Explanation code

A two source file is now created within the project for the Dice class. The file must have the same name as the public class it contains. So in this case, the source file is called Dice.java (note the capital letters).

In the Dice class, we make the throw method with the code that mimics a dice roll. This method doesn't have to be `static`, because we're going to create an object from the Dice class first. This is done in the `main` method of the `ConsoleApp` class. An object created from a class is called an instance of the class.

Code in main

- `Dice dice = new Dice();`
  On this line, three things are done at once:
  1. `Dice Dice` - A variable called dice is made of type Dice (can contain a Dice object).
  2. `new Dice()` - A new instance of the Dice class is created, the result is a Dice object.
  3. `... = ...` - The new Dice object is stored in the dice variable
- `Int` Dice creates an integer variable called Dice where we can later store the value of the rolled dice.

- `dobbel = dice.roll()` calls the throw method of the dice object and puts the result in the variable dice.

Exercises

For exercises 8, 9 and 10, a video is:

Click here for the video!

Exercise 8

Create a project with a start class like the example. Create a second class called Test. In the test class, you create a `containsNumbers` method that takes a String as an argument and tests whether there are numbers in the input. The result of the test is a `boolean`.

Exercise 9

Add to the Test class of the previous exercise a `getNumbers` method that takes a String as an argument and returns only the numbers from the input. The result of the method is a String.

Exercise 10

Add to the Test class of exercise 8 a `countDecimals` method that takes a double as an argument and returns how many numbers are after the decimal point (period). The result is integrity.

Scope of a variable/method

When creating a variable or a method, you specify what the scope is. The scope determines where you can use the variable or method. You do this by putting the reserved words `public`, `private` or `protected` in front of it. What `protected` means comes later when you learn more about classes and objects.

In addition to the declaration with public or private, the scope of a variable is also determined by the place where you create the variable.

```
Public .class Main {
   Public Static Void main(String[] args) {
      Scope  scope = New Scope();
      System.  out.println(scope. string1);
      System. out.println();
      System. out.println(scope.getStringsFromVariables());
      System. out.println();
      System. out.println(scope.getStringsFromMethods());
      System. out.println();
      System. out.println(scope.getOtherString());
   }
}
```

```
Public .class Scope {

   Public String string1 =  "A string in scope";
   Private String string2 =  "Another string in scope";

   Private String getString(Int i) {
      if(i == 1) {
         Return string1;
      } else {
         Return string2;
      }
   }

   Public String getStringsFromVariables() {
      Return string1 + "\n" + string2;
   }

   Public String getStringsFromMethods() {
      return getString(1) + "\n" + getString(2);
   }

   public String getOtherString() {
      String other =  "Another string";
      Return other;
   }

}
```

Explanation

- The `variable string1` is `declared public` and can be used within the class but also in `Main` where an object of the `Scope` class has been created.
- The variable `srting2` is `declared private` and may be used within the class, in the `Main` class this variable is not available.
- The `other` variable is declared within the block of the `getOtherString()` method and can only be used there. For each variable within a method, it can only be used within the first block in which it is located, i.e. within the braces that enclose the variable first.
- The `getString(int i)` method is privately declared and may be used within the class, outside of it, for example in `Main`, this method is not available.
- The getStringsFromVariables() and `getStringsFromMethods()` methods are publicly declared and can be used within the class and are also available in `Main` via the `scope` object of the Scope class.

Rules

- In general; Make the scope of a variable as small as possible.
- Give a larger scope variable a longer descriptive name.

Exercise

```
Public .class Main {

   Public Static Void main(String[] args) {
      Scope  scope = New Scope();
      System. out.println(scope.repeatStringHorizontal("xyz", 10));
      System. out.println();
      System. out.println(scope.repeatStringVertical("xyz", 10));
   }

}
```

```
Public .class Scope {

   public String repeatStringHorizontal(String repeat, Int count) {
      StringBuffer output = New StringBuffer();
      for(Int c = 0;  c < count ;  c++) {
         output.append(repeat);
      }
      Return output.toString();
   }

   public String repeatStringVertical(String repeat, Int count) {
      StringBuffer output = New StringBuffer();
      for(Int c = 0;  c < count ;  c++) {
         output.append(repeat);
         output.append("\n");
      }
      Return output.toString();
   }

}
```

Take the example and test the code. Move the StringBuffer `output = new StringBuffer()` to the top of the class and remove it from the other method. See what the result is.

Recursion

Recursion is a trick to simplify code of problems with a mathematical background in particular. Recursion calls the same method in a method, but with a different parameter. This sounds complicated, so we'll discuss an example.

Example: Multiplying two numbers

Multiplication is something you learned in elementary school. Now we're going to look at it a little more mathematically. We consciously overlook the fact that multiplying in a programming language is very easy (by the way, there are still programming languages that cannot multiply, this usually has to do with the simplicity of the platform)

Imagine: you want to calculate 4*3, we all know that the outcome is 12, but do you calculate that?

Outcome = 4 + 4 +4

So: you take 4 (1^{e} times) and then you add 4 (2^{e} times) and add another 4. So you have added up 3 fours. Below you will find the example code:

```
public class Multiplication {

public int multiply(int number  of times, Int number) {
        if (number == 0) {
            Return 0;
        } else if (number == 1) {
            Return number;
        } else {
            Return multiply(number-1, number) + number;
        }
    }
}
```

Explanation

Suppose you call this method with `multiplication(0, 3)`:

1. number is then 0.
2. The method then returns with 0. After all, 0*3= 0

Then we try a `multiplication(1, 3)` call:

1. number of times is 1
2. The method then returns with 3, as we learned in arithmetic: 1*3= 3

Finally, the last possibility, the `multiplication(2, 3)` call:

1. number of times is 2
2. The method calls itself with number times – 1 (2-1=1)
3. In the second round, the number of times is 1
4. The second call returns with response 3
5. We add 3 (3 + 3 = 6)
6. The first call returns with response 6

Don't go on indefinitely

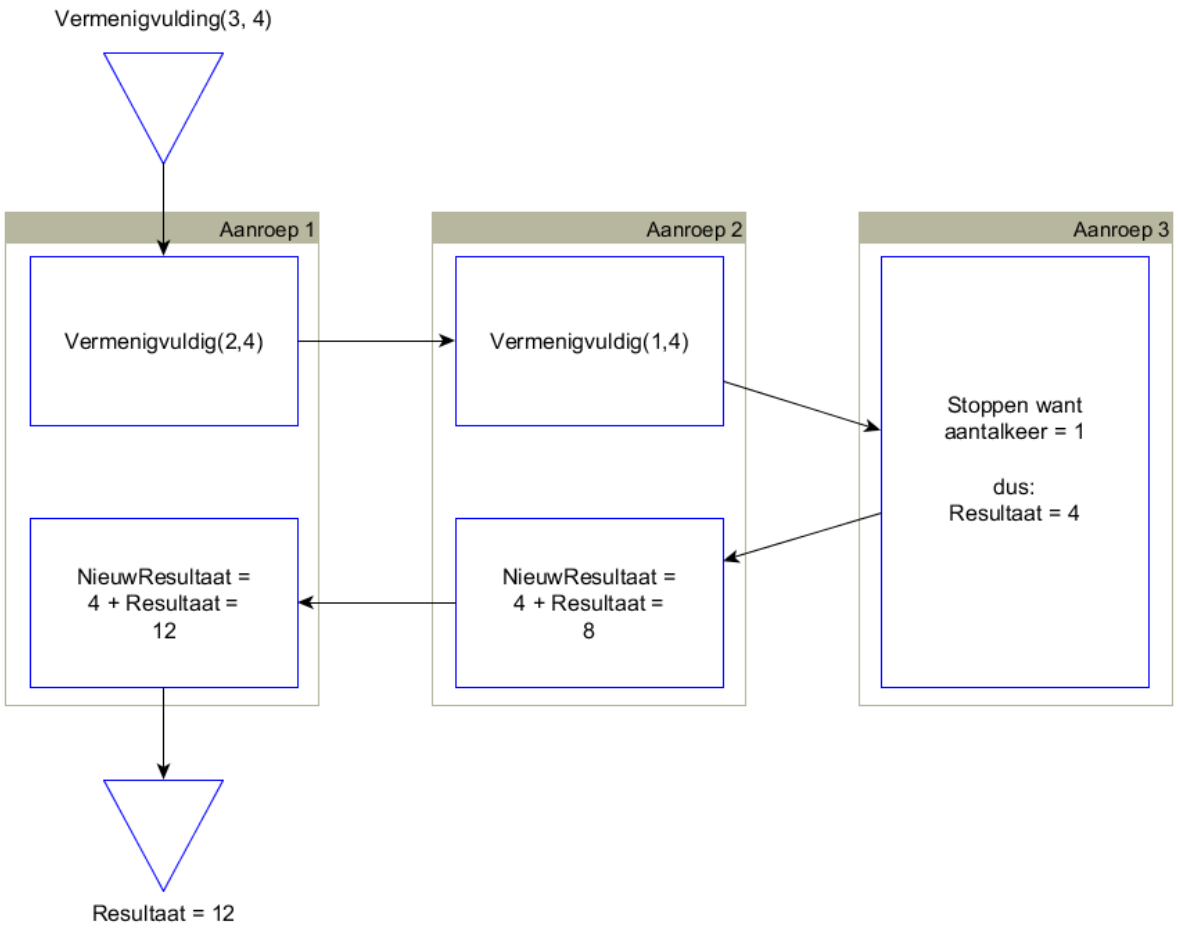
One of the most important properties of a recursive method is that it also contains a stop condition. Because otherwise you could go on indefinitely, because you are calling the same method over and over again that calls itself that calls itself....

Eventually the program or worse your computer crashes. In the code above, this is regulated by the rules

```
if (number == 0) {
    Return 0;
} else if (number == 1) {
    Return number;
```

We know that -1 is done every time, and so you will arrive at number times = 1, and then the method jumps back out. And number = 0 ensures that it goes well if you start with 0.

Schematically, the program runs as follows:



Exercise

Take the code below and come up with at least 3 more multiplications to make sure it's correct.

To do this, put in the class **Multiplication** a few System.out.println commands so you can see where the program is running.

```
public class Main {

   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {
TODO code application logic here
      Multiplication v = New Multiplication();

System. out.println("4 * 3 = "+Integrity. toString(v.multiply(4, 3)));
System. out.println("6 * 5 = "+Integrity. toString(v.multiply(6, 5)));
   }

}
```

Example 2: Calculating factorial

With the same way you can also calculate a faculty (see also the sources)

- 0! = 1
- 1! = 1
- 2! = 1 *2
- 3! = 1 * 2 *3
- 4!= 1 * 2 * 3 *4
- And so on

Exercise

Also work this out in Java just like the example above.

Final assignments

You can get the code for the final commands on https://github.com/DrentheCollege/A2Eindopdrachten.git with a **git clone** command.

There is a video explaining the final assignments:
Click here for the video!

Assignment 1: Calculate formulas

Create a recursive function that can do calculations.

The requirements:

- We follow the standard calculation rules
  - So calculate within the brackets first
  - First multiply and only then addition and subtraction
- We use integers (int)

So for example:

4 + 3 =7

4 *3 + 2 = 12 + 2 = 14

(3+4)*3 = 7 * 3 = 21

((2 +1) * 4) + (2 * 3) = (3 * 4) + (2 * 3) = 12 + 6 = 18

That **blot** solve you with a recursive method (definition: **calculate public int(String input**), and don't forget to return the result over and over again. (**Return**)

```
public class Assignment1 {
    public static void main(String[] args){
        String formula = "4+3";
        Integer result = FormulaCalculate(formula);
        System.out.println(formula + " = "+result);

        formula = "4 *3 + 2";
        result = FormulaCalculate(formula);
        System.out.println(formula + " = "+result);

        /* here you can also work out the other formulas and
           Add your own formulas
        */
    }
}

class FormulaCalculate {

    calculate static Integer(String formula) {
        return 0;
    }
}
```

You can check this with the above sums (use them in your code, for example), but come up with 3 yourself and check the result.

Assignment 2: Clean up string of characters

In this command you get a piece of code that is not finished. Think of a way to remove all the non-desirable characters from the string here. Create a new class that will do this, and use as many methods as possible.

Tekst.replaceAll is not allowed, it is the intention that you devise and write the program yourself.

```
public class Assignment2 {
   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {
Char[] away = {'"', '?' };

     String text =
                          "This is a text with \" and ** and ?? "+
                          "and all sorts of other undesirable signs like ® etc." ;

System.  out.println(text);
   }

}
```

Sources

- ✓ **Faculty explanation**
  https://www.wisfaq.nl/showfaq3.asp?id=19971