Fundamentals of Programming 2

Lecture 4

Aurelia Power, TU Dublin – Blanchardstown Campus, 2019

* Notes based on the Java Oracle Tutorials, Deitel & Deitel, and Horstman

Introduction to Methods

- Modularisation allows us to develop and maintain large programs, that is, to construct them from small, simple pieces or modules – aka divide and conquer approach.

- Modules in Java are called **methods and classes**.

- Java programs are written by combining new methods and classes that you write with predefined methods and classes available in the Java Application Programming Interface and in various other class libraries.

- Related classes are typically grouped into packages so that they can be imported into programs and reused.

- **3 Main Reasons/Motivations to use methods:**

1. program development becomes very manageable
2. software re-usability
3. separation of concerns

Java API

- Java contains many predefined classes that are grouped into categories of related classes called packages
- Known as the Java Application Programming Interface (Java API), or the Java class library
- Overview of the packages in Java SE 11 (we are on java 11!!)
  - https://docs.oracle.com/javase/11/docs/api/index.html?overview-summary.html
- Additional information about a predefined Java class's methods
  - https://docs.oracle.com/javase/9/
- The Java API provides **a rich collection of classes and methods for performing**:
  - common mathematical calculations
  - string manipulation
  - character manipulation
  - input/output,
  - error checking … and many other useful operations.
- This makes the programmers job easier because these methods provide many of the capabilities programmers need.
- In fact, it is recommended to use the methods that are already defined in the API because they have efficient implementations, rather than implementing them yourself, unless you are required to do so in this module ☺ …for learning purposes.

What are methods?

- So, what exactly is a method??? (similar to Python functions)

Like a Python function, a method is a very specific sequence of instructions with a meaningful name.

- Each method should be limited to performing a **single, well-defined task**, and the **method name** should express that task.

- And we already have encountered some predefined methods from the java API that carry out very specific tasks and which have very telling names:
  - nextInt() from the Scanner class which takes the next token from the input and tries to parse it into an int;
  - length() from the String class which returns the number of characters of the given string;

What are methods?

- We also can write our own methods to define specific tasks that may be used at many points in a program.

METHOD = ACTION

- These are referred to as **programmer-defined methods.**
- The statements in method bodies are written only once, are hidden from other methods, and can be reused from several locations in a program.
- A method is **invoked/called** by a method call.
- When the called method completes its task, it either returns a result or simply returns control to the caller (the method where we specified the call/invocation).

Method Calls

- Each method may be supplied with several arguments when called, or none;
- Method arguments may be constants, variables or expressions.
- We can make 2 types of method calls:

1. calls to **instance methods** when we use an object to call methods that are defined for it:

EXAMPLE 1:

```
Scanner sc = new Scanner (System.in); // sc is an object of type Scanner
int a = sc.nextInt(); /* calling the method nextInt on the sc object */
```

EXAMPLE 2:

```
String str = "FOP1"; // str is an object of type String
int a = str.length(); /* calling the method length on the str object */
```

Method Calls

2. calls to **static methods** when we need a method to perform a task that does not depend on any object, and in this case we can simply use the name of the class where the method was defined; for the time being, we will focus on static methods.
   - A static method applies to the class in which it's declared, so it also known as a class method → why is the main method declared static??? Because it allows the JVM to invoke main without creating an object of the class.
   - You can call any static method by specifying its class name, followed by a dot (.) and the method name.

EXAMPLE from the Math class which contains only static methods → for this reason, it is also known as an utility class

- Each of the methods defined in the Math class are called by preceding the name of the method with the class name Math and the dot (.) separator

Math.abs(-1); // returns 1 or Math.pow(2, 3); //returns 8

- If you define a static method and invoke it in the same class you don't need to even specify the name of the class, simply the name of the method will suffice.

MATH class

- The **class Math** contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
- It is found in the java.lang package so we don't need to import it.
- Some more examples are shown below (you must also check the java API).
  - abs(x) – returns the absolute value of x; for example, abs(23.7) returns 23.7, abs(0.0) returns 0.0, and abs(-23.7) returns 23.7
  - ceil(x) – returns the rounded value of x to the smallest integer not less than x; for example, ceil(9.2) returns 10.0 and ceil(-9.8) returns -9.0
  - floor(x) – on the other hand, returns the rounded value of x to the largest integer not greater than x; for example, floor(9.2) returns 9.0 and floor(-9.8) returns -10.0
  - sqrt(x) – returns the square root of x; for ex., sqrt(900.0) returns 30.0
  - pow(x, y) – returns the value of x raised to the power y; for ex., pow(2.0, 7.0) returns 128.0 and pow(9.0, 0.5) returns 3.0
  - max(x, y) returns the larger value among x and y; for ex., max(2.3, 12.7) returns 12.7 and max(-2.3, -12.7) returns -2.3

MATH class

- The class Math also declares commonly used **mathematical constants/final variables**:
  - Math.PI is 3.141592653589793 and defines the ratio of a circle's circumference to its diameter
  - Math.E is 2.718281828459045 and defines the base value for natural log
- These fields are class variables and are declared in class Math with public, final and static modifiers:
  - public allows you to use these fields in your own classes, from any class;
  - final indicates a constant—value cannot change;
  - static allows them to be accessed via the class name Math followed by dot (.) operator and the name of the method
- REMEMBER: the way the compiler knows how to make the difference between methods and variables is that methods always have a set of round parentheses () where the list of arguments are supplied (NOTE: the parentheses may be empty if no arguments are required).

How to Implement your own Methods

Syntax of a concrete public static method:

```
public static returnType methodName( parameterType parameterName, . . . )
{
    method body [statements to execute and return statement only for return-value methods;]
}
```

Notes:

- The returnType specifies the data type of the value returned by the method; if the method does not return any value, then the returnType is void;
- The parameter list can be empty, or it can contain several parameters; if it contains parameters, for each one of them you must specify the datatype and a name;
- If the method is void then no return statement is required inside the method body; but if the method returns a value, then the body must contain as the last statement a return statement which returns a value whose datatype must be the same as the one specified in the header.

How to Implement your own Methods

- To begin with, we will write only concrete public static methods:

1. Specify the public and static modifiers;

2. Specify the type of the return value, for example, you can specify that it should return a value of type double, or you can specify that it returns no value by using the void keyword;

3. Pick a meaningful name for the method which should indicate what the method does, for example, cubeVolume

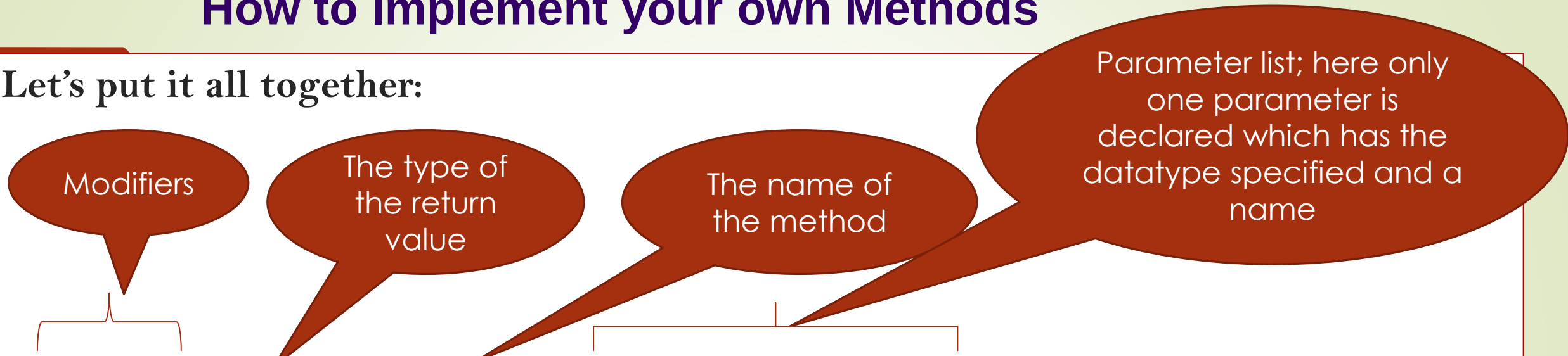
4. Declare the datatype and provide an identifier for each **parameter** (if any); for example, double sideLength specifies that the argument that needs to be supplied when the method is called must be of type double.

   NOTE: 1-4 form **the header/signature of the method**

5. Define the body using a pair of curly braces within which we specify the instructions that the method needs to carry out, that is, **its implementation.**

How to Implement your own Methods

Let's put it all together:




```
public static double cubeVolume ( double sideLength ) //method signature/header
{ //begin method body
    double volume = sideLength * sideLength * sideLength;
    return volume; /* if you specify that your method returns a value, then the last
                      statement must be a return statement that returns a value which
                      matches exactly the one declared in the header */
}//end method body
```

- The elements in red are compulsory in every concrete method definition: that is, return type, name, parameter list, and the pair of curly braces are all compulsory
- Unless a method is called by another method (e.g. the main method), the statements in the header and the body will never be executed.

```java
import java.util.Scanner;

/**
 * @author Aurelia Power *
 */
public class CubeDemo {
    /**
     * @param args
     */
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the side length of a cube: ");
        while(sc.hasNextDouble())
        {
            double side = sc.nextDouble();
            System.out.println("The volume of the cube with the length side of "
                    + side + " is " + cubeVolume(side));
            System.out.print("Enter the side length of a cube: ");
        }
        sc.close();
    } //end main

    /**
     * computes the volume of a cube
     * @param sideLength the side of a cube
     * @return the volume of the cube
     */
    public static double cubeVolume(double sideLength)
    {
        double volume = sideLength * sideLength * sideLength;
        return volume;
    }//end of cubeVolume
}//end class
```

Method call … how many times is the method invoked???

It depends on when the user decides to terminate the input by entering something that cannot be parsed into a double

Method declaration and implementation

SAMPLE OUTPUT

```
Enter the side length of a cube: 2
The volume of the cube with the length side of 2.0 is 8.0
Enter the side length of a cube: 3
The volume of the cube with the length side of 3.0 is 27.0
Enter the side length of a cube: 7
The volume of the cube with the length side of 7.0 is 343.0
Enter the side length of a cube: v
```

Let's look more in depth to our program

- Once we have defined our method, we can call it inside the loop several times;
- When the method is called, variables are created that will be initialised to the argument(s) values that are supplied; as mentioned before, they are called parameters;
- Assume that the user entered **2** when prompted:

1. The value of 2.0 is assigned to the variable side;
2. Then that value will serve as the argument to the method call of cubeVolume as follows: cubeVolume(2.0);
3. Then the value of 2.0 initialises the parameter sideLength which we have defined in the parameter list
4. sideLength value is then used to compute the volume of a cube as follows: 2.0 * 2.0 * 2.0, so the result of 8.0 will be assigned to the volume variable;
5. then that value of 8.0 is returned to the main method when we used the return statement
6. Finally, the main method uses that value of 8.0 in the print statement to output the value of the cube with the given side length of 2.0.

NOTE: every time the method is called, the parameter **sideLength** and the variable **volume** inside the method body are created, and then destroyed once the control is returned to the main method

Let's look at when the method is called with 2!

2 variables are created:
- result1 which will store the value returned by the method
- sideLength which is the parameter passed to the method

The parameter variable is initialised to the value of the argument that was passed in the call:
- In our case the parameter variable is **sideLength** and the argument is **2**, so sideLength will be initialised to the value of 2

Let's look at when the method is called with 2!

3 About to return to the caller

```
double volume = sideLength * sideLength * sideLength;
return volume;
```

result1 =

sideLength = 2

volume = 8

The method then computes the value of the expression: **sideLength *sideLength*sideLength,** which in our case is **8** (2 * 2 * 2), and assigns this value to the variable **volume.**

4 After method call

```
double result1 = cubeVolume(2);
```

result1 = 8

Finally, the method **returns a value (and control) to the main method; in other words, the returned value of 8** is transferred to the caller (in our case, the main method) which is now assigned to the variable **result1.** Additionally, all the variables created in the method cubeVoulume are cleared (wiped from memory).

Your turn…

- Describe what happens when you call it with 10:

```
double result = cubeVolume(10);
```

- Here is another **program (assume class declaration):**

```
public static int someMethod(int x){
    int y = x*x; return y;
}
public static void main(String[] arguments){
    int a = 4;
    System.out.println(someMethod(a++));
}
```

What does it print???

Types of Methods

- In Java, **methods** are used to **create mini programs** within a larger program.
- Then they can be called over and over again as needed, from anywhere in your program, as long as they are

- In general, there are **3 types of methods** based on the return type and parameter list:

    1. A method that **carries out some action**.

    2. A method that is **passed parameters** to carry out some action.

    3. A method that **carries out some action** and **returns a value**. It may also be passed some parameters.

Type 1 - A Method that carries out some action

- This type of method simply carries out some action that does not depend on any parameters;
- It is typically void;
- For example, printing 50 stars to the screen;

```
public static void draw50Stars( ) /*method signature*/ {
    for (int x = 0; x < 50;  ++x){
        System.out.print('*');
    }
} //end method
```

The keyword **void** indicates that **no value is returned to the calling method.**

Thus, **.no return statement is required as the last statement in the body.**

- Once method is defined - can be used as often as you like. For instance, it can be called in the main method as follows: draw50Stars();

Type 2 - Method passed Parameters to carry out some action

- The method **draw50stars()** is limited because it always prints the same number of 50 stars…what if the user want less or more??
- Much better if user could decide on number of stars to be printed; to do so we specify how many stars to draw when we invoke the method;
- To **pass values to methods** when they are invoked, we use what are called **parameters** **which act as placeholders for the actual values of arguments.**
- We can then modify the drawStars method to pass a parameter that tells how many stars to draw.

```
drawStars(x);
```

Initialises the parameter numStars to the value of x.

```
public static void drawStars(int numStars)
{
    // code  to print the message
}
```

The round brackets contain the parameters and each must have a data type and a name. In this case one parameter is passed: it is of type int, and its name is numStars

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

    Scanner input = new Scanner (System.in);

    int x;

    System.out.println("please input number of stars to print");
    x= input.nextInt();

    //call method to draw stars
    drawStars(x);


    //print a blank line
    System. out.println(" ");
    System. out.println("Hello World");

    //call method to draw stars
    drawStars(x);

}
```

The value of x is passed to drawStars method and will initialise the parameter numStars

```
//Method to draw stars
public static void drawStars(int numStars){

    for(int j = 0; j < numStars; j++){

        System.out.print("*");
    }
}
```

The **declaration and implementation** of the drawStars method

Exercise: put these methods into a class and run the program

Type 3 - A method that carries out some action and returns a value. It may also be passed some parameters.

- For instance, we can define and implement a method that finds the maximum value among 3 double values;

- As you recall, we implemented this in Python as a function because it is better to write the code once in a function/method and call it as many times as we need.

```
public static double max(double n1, double n2, double n3)
{
    // code to find out which of the 3 parameters is the largest
    // finally, the return statement which must return a double value
}
```

```
 */
public static void main(String[] args) {
    /* call the method max 3 times which
     * returns a double value and output it*/
    System.out.println(max(2, 7, 10));
    System.out.println(max(-10, -7, -2.5));
    System.out.println(max(23.9, 7.7, 10));
}//end main

/**
 * finds the maximum of 3 values
 * @param n1, n2, n3 the numbers being checked
 * @return max the maximum of the 3 numbers
 */
public static double max(double n1, double n2, double n3)
{
    /* declare a variable max and initialise it
     * to the value of the first parameter */
    double max = n1;
    //compare max to the value of second parameter
    if( max < n2 )
    { // if max is smaller than n2 then assign the value of n2 to max
        max = n2;
    }
    //compare max to the value of third parameter
    if( max < n3 )
    { // if max is smaller than n3 then assign the value of n3 to max
        max = n3;
    }
    //finally return the value of max which is a double
    return max;
} // end max
```

Three calls to the method max.

The **declaration and implementation** of the max method

Exercise: put these methods into a class and run the program

Declaring and Implementing Methods - summary

REMEMBER

- Method header
  - Modifiers, return data type, method name and parameter list (if more than one parameter, they will be comma separated)
- Method body
  - Delimited by left and right braces – { }
  - Contains one or more statements that perform the method's task
  - Depending on the method type, a method might contain one or more return expression; statements which return a value that matches the return data type declared in the header;
  - if more than one return expression; statement in the program they must be associated with conditional constructs, but the method must contain at least one return expression; statement as the last statement.

NOTE: there are more than one way of returning control to the caller method

Returning Control from a Method Call

There are **three main ways** to return control to the point at which method was invoked.

The first 2 apply to when the method does not **return a result**, that is , it is **void:**

1. When the method-ending **}** is reached, the control is returned to the caller method;
2. When the statement **return;** is executed the control is also simply returned to the caller; (note that no value is specified), and it can be found anywhere in the method implementation, not only as the last statement;
3. The last type applies to when the method does return a result, by executing the **return *expression;*** statement and control is returned to the caller by returning the specified value.

NOTE: When a return statement is executed, control is returned immediately to the point at which the **method** was invoked.

Modifiers

Java provides modifiers to control access to data, methods and classes.

- **static**

Applies to variables and methods. It represents class-wide information that is shared by all instances of the class.

- **public**

Applies to classes, methods, and variables in such a way that all programs can access them.

- **private**

Applies also to (inner only) classes, methods, and variables in such a way that all programs can access them.

Your turn:

- What type of method it is?
- Identify the elements of the following method declaration and specify which ones are required, which ones are not;
- Then write the main method where you call the min method several times; at least one call should make use of user input.

```
public static int min( int n1, int n2)
{
    if( n1 < n2 )
    {
        return n1;
    }
    else{
        return n2;
    }
} // end min method
```

Your turn:

1. Identify and describe the 3 different types of methods.
2. Provide three advantages behind modularizing/using methods in your code.
3. Explain how you would invoke a static method in the same class and from a different class.
4. Identify each of the elements used to define a method in the following example; what does the method do?

```
public static int numOfEvens(int begin, int end){
        int numOfEvens = 0;
        for(int j = end; j >= begin; j--){
                if(j % 2 == 0){
                        numOfEvens++;
                }
        }
        return numOfEvens;
}
```

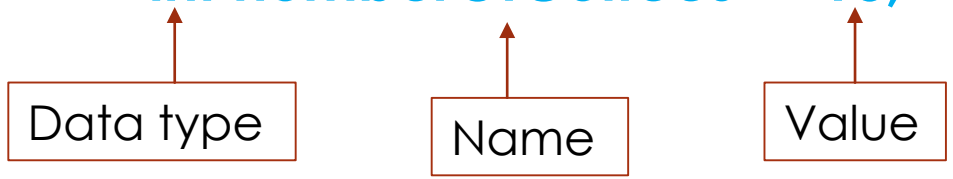
Attributes of Variables

✓ So far, we have used consistently several attributes of a variable; they include:

- **Name**
- **Data Type**
- **Value**

✓ Example:

```
int numberOfCoffees  = 10;
```



✓ But each variable has other attributes, and we will discuss 2 of them:

- **Scope**
- **Duration**

Scope

- An identifier's/variable's **scope** is where the identifier can be referenced in a program, that is, it refers to where the program can use that variable.

➔ The scope of an identifier is the portion of the program in which the identifier can be referenced/used.

- Some identifiers can be referenced/used throughout the entire program, and, in this case, they have class scope. ➔ An **instance variable** or a **static (class) variable** can be referenced from anywhere in the class, so they have class scope.
- Instance methods and static methods can also be referenced from anywhere in the class that contains them ➔ they also have class scope.

- Others identifiers can be referenced from/used in limited portions of a program, and in this case they have block scope.
- A **local variable** declared in a block can be referenced only in that block – so it has block scope.

NOTE: A block may or may not have itself nested blocks.

Class Scope

- Methods, class/static variables, and instance variables of a class have class scope.
- Class scope begins at the opening left brace "{" of the class definition and terminates at the closing right brace "}" of the class definition.
- Class scope enables methods of a class to **directly invoke all methods defined in the same class or inherited into the class** (such as methods inherited from the Object class) and to directly access all instance variables defined in the class.
- All static and instance variables and methods of a class can be considered global to the methods of the class in which they are defined.
- The methods can modify the instance variables directly and invoke other methods of the class.

Block Scope

- An identifier declared inside a block have block scope. Examples of blocks include: method block, if block, loop block, switch block.
- Block scope begins at the identifier's declaration and ends at the terminating right brace "}" of the block.
- **Local variables** declared in a method and **method parameters** declared in the method header have block scope; in other words, they can only be used in that method, from the point where they were declared.
- Similarly, **local variables** declared in an if, for, while, do-while, switch constructs (as part of a method) have block scope and they can only be used in/referenced from that block/construct, from the point where they were declared up to where that construct ends.
- If you try to access a local variable before it was declared or after its block scope, you'll get a compilation error.
- A block may contain any number of variable declarations.

```java
public class ScopeDemo {//begin class
    /* the following 2 variables are global to the class,
     * so they can be accessed from anywhere in the class */
    static int classVariable = 10; // class/static variable
    double instanceVariable = 2.5; // instance variable

    public static void main(String[] args) {//begin main
        System.out.println("Accessing the static variable: " + classVariable);
        ScopeDemo sd = new ScopeDemo();
        System.out.println("Accessing the instance variable: "
                + sd.instanceVariable);

        int localVariable = 12; // local variable can only be accessed/used in the main
        System.out.println("Accessing local variable: " + localVariable);

        /* declaring another local variable k in the for loop,
         * so it can only be used in the for loop */
        for(int k = 10; k > 0; k-=2){
            System.out.println("Accessing local variable that can be used only in the loop " + k);
        }

        int localVariable2 = 7;// local variable can only be accessed/used in the main
        System.out.println("Accessing local variable 2: " + localVariable2);
        System.out.println("Accessing local variable again: " + localVariable);

        //invoking/using the method sayHello
        sayHello("Aurelia");
    }//end main

    //the parameter name can only be used in the method sayHello
    public static void sayHello(String name){ //begin sayHello
        System.out.println("Hello " + name + "!");
    }//end sayHello
}//end class
```

Class scope: all the variables directly declared in the class (classVariable and instanceVariable, as well as all the methods, are global and can be accessed from anywhere in the class)

Local Scope all the variables declared as parameters or declared in a method

Duration

- An identifier's **duration** (also called its **lifetime**) determines the period during which that identifier exists in memory.

- Some identifiers exist briefly; some are repeatedly created and destroyed; others exist for the entire execution of a program.
- Duration is related to scope.
- Identifiers that represent **local variables in a method** (parameters and variables declared in the method body) have **automatic duration**.
  - ✓ Automatic duration variables are created when program control enters the block in which they are declared.
  - ✓ They exist as long as the block is active and they are **destroyed** when the block is exited.
  - ✓ **Variables of automatic duration** are referred to as **automatic variables.**

Automatic duration is a means of conserving memory because automatic duration variables are created when the block is which they are created is entered and are destroyed when the block is exited.

Duration

- **Automatic duration** is an example of the **principle of least privilege** which states that each component of a system should have sufficient rights and privileges to accomplish its designated task, but no additional rights or privileges. This helps prevent accidental and/or malicious errors from occurring in systems.

- **Static Duration:** Java also has identifiers of static duration which exist from the point at which the class that defines them is loaded into memory until the program terminates. Static duration applies to all variables and methods declared as static.

  - ✓ For static duration variables, storage is allocated and initialised once their classes are loaded into memory.

  - ✓ For static duration methods, the names of the methods exist when their classes are loaded into memory.

- **NOTE: there is a 3rd type of duration - Object Duration not covered yet**

```java
public class ScopeDemo {//begin class
    /* the following 2 variables are global to the class,
     * so they can be accessed from anywhere in the class */
    static int classVariable = 10; // class/static variable
    double instanceVariable = 2.5; // instance variable

    public static void main(String[] args) {//begin main
        System.out.println("Accessing the static variable: " + classVariable);
        ScopeDemo sd = new ScopeDemo();
        System.out.println("Accessing the instance variable: "
                + sd.instanceVariable);

        int localVariable = 12; // local variable can only be accessed/used in the main
        System.out.println("Accessing local variable: " + localVariable);

        /* declaring another local variable k in the for loop,
         * so it can only be used in the for loop */
        for(int k = 10; k > 0; k-=2){
            System.out.println("Accessing local variable that can be used only in the loop " + k);
        }

        int localVariable2 = 7;// local variable can only be accessed/used in the main
        System.out.println("Accessing local variable 2: " + localVariable2);
        System.out.println("Accessing local variable again: " + localVariable);

        //invoking/using the method sayHello
        sayHello("Aurelia");
    }//end main

    //the parameter name can only be used in the method sayHello
    public static void sayHello(String name){ //begin sayHello
        System.out.println("Hello " + name +  "!");
    }//end sayHello
}//end class
```

Static Duration: all the variables and methods directly declared in the class as static(classVariable and sayHello)

Object Duration: all instance variables and methods directly declared in the class (instanceVariable)

Automatic duration: all the variables declared as parameters or declared in a method

Your turn

```
3  public class YourTurn {
4      static int n = 7;
5
6      public static vo
7          System.out.p
8          mysteryMetho
9          int x = 7;
10         for(int i =
11             String s
12             System.o
13         }
14         System.out.p
15         mysteryMetho
16     }//end main
17
18     public static vo
19         for(int i =
20             for(int
21                 Syst
22             }
23             System.out.println();
24         }
25     }// end mysteryMethod
26 }//end class
```

For each of the variables in the following program, indicate the scope. Then determine what the program prints, without actually running the program.

```
1  public class Sample
2  {
3     public static void main(String[] args)
4     {
5        int i = 10;
6        int b = g(i);
7        System.out.println(b + i);
8     }
9
10    public static int f(int i)
11    {
12       int n = 0;
13       while (n * n <= i) { n++; }
14       return n - 1;
15    }
16
17    public static int g(int a)
18    {
19       int b = 0;
20       for (int n = 0; n < a; n++)
21       {
22          int i = f(n);
23          b = b + i;
24       }
25       return b;
26    }
27 }
```

- What is mysteryMethod doing?
- an the variable n be used?
- an the variable x be used?
- ne output of line 7?
- ne output of the code on
- ne output the for loop
  e main method?
- ne output of line 14?
- ne output of line 15?
- ced the following
  nt System.out.println(--n); in
  op after line 12, what will
  utput of the for loop?
- if we placed the following statement System.out.println(s); after line 15, what will be its output?

Your turn

For each of the variables in the following program, indicate the scope. Then determine what the program prints, without actually running the program.

```
1   public class Sample
2   {
3      public static void main(String[] args)
4      {
5         int i = 10;
6         int b = g(i);
7         System.out.println(b + i);
8      }
9
10     public static int f(int i)
11     {
12        int n = 0;
13        while (n * n <= i) { n++; }
14        return n - 1;
15     }
16
17     public static int g(int a)
18     {
19        int b = 0;
20        for (int n = 0; n < a; n++)
21        {
22           int i = f(n);
23           b = b + i;
24        }
25        return b;
26     }
27  }
```

Method overloading

- Example: imagine that you are taking notes during the FOP2 lecture: you may use pen, you may use the laptop to type (for those allowed). I can approach this situation in **2 ways** to instruct you how you can take notes:

1. I can use 2 different sets of instructions for each method of taking notes which can be implemented differently:

```
usePenToTakeNotes(){ /* instructions here to take notes using pen*/ }
useLaptopToTakeNotes (){ /* instructions here to take notes using laptop*/ }
```

2. Or I can use the same set of instructions, and then provide you with the 2 tools of taking notes:

```
takeNotes(Pen pen) { /* instructions here to take notes */ }
takeNotes(Laptop laptop) { /* instructions here to take notes*/ }
```

- The latter approach has **2 advantages**:

1. It is simpler: write one set of instructions and apply it with different tools;
2. It is more flexible: for example, you can add another type of tool such as a smartphone without changing the set of instructions

Method overloading

- Similarly, Java enables several methods with the same name (carrying out the similar instructions) to be defined in the same class as long as these methods have **different sets of parameters**:

    - based on the number of parameters
    - based on the types of parameters
    - based on the order of parameters

- This phenomenon is called **method overloading**.

- Method overloading is commonly used to create several methods with the same name that perform **similar tasks**, but on different data.

- Overloading methods that perform closely related tasks can make programs more reusable and understandable.

- When an overloaded method is called, … the Java compiler selects the proper method by examining the **number, types and order** of the arguments in the call.

Method overloading

- We have already encountered method overloading many times;
- For instance, the output methods that we have used – **print, printf** and **println** – have many overloaded versions:

```
int intVal = 10;
boolean boolVal = false;
String name = "eJava";

System.out.println(intVal);
System.out.println(boolVal);
System.out.println(name);
```

Prints an int value

Prints a boolean value

Prints a string value

- When we use the **println** method, we know that whatever we pass as argument, it's value will be printed to the console.
- If java did not allow overloading, we would probably have to use **printlnInt** to print the value of an int, or **printlnBoolean** to print the value of a boolean, or **printlnString** to print the value of a String and so on… which is not only time consuming, but also confusing.
- So without overloading Java would have to re-define the same functionality of printing to the console many times for each data type (not only primitives, but also references);

Method overloading

- There are several rules when implementing your own overloaded methods:

1. Overloaded methods **must have different parameter list** (whether in number, type or order);
2. When called/invoked, overloaded methods accept different arguments, but they must match at least one of the parameter lists already defined;
3. Overloaded methods may or may not define a different return type;
4. Overloaded methods may or may not define different access modifiers (e.g. private and public);
5. Overloaded methods can't be defined by only changing the return type and/or the by changing the access modifier.

For now, we will use only public and static methods…

Method overloading

- Let's implement a method called **computeAverage** which calculates the average of several values;
- So we start by applying the public and static modifiers;
- Then we must provide a return type (typically decided based on what the method does)… so, our method may return a double value, since average is typically represented as a real number;
- We provide the name which is computeAverage
- Next, we specify the parameters – the data used to compute the average; but, we can calculate the average of 2 int values, or the average of 3 int values, or the average of 2 double values, or the average of one int value and one double value and so on…
- Finally, we implement the method in the method body denoted by the pair of { }; the data/parameters passed are used to implement the method.

Method overloading

- Let's implement a method called **computeAverage** which calculates the average of several values;
- So we start by applying the public and static modifiers;
- Then we must provide a return type (typically decided based on what the method does)… so, our method may return a double value, since average is typically represented as a real number;
- We provide the name which is computeAverage
- Next, we specify the parameters – the data used to compute the average; but, we can calculate the average of 2 int values, or the average of 3 int values, or the average of 2 double values, or the average of one int value and one double value and so on… thus, we can implement as many methods…
- Finally, we implement the methods in the method body denoted by the pair of { }; the data/parameters passed are used to implement the methods.

Method overloading

- Let's implement a method called **average** which calculates the average of several values;
- When we specify the parameters – the data used to compute the average - we can specify 2 or 3, and so on..
- Let's define 2 overloaded average methods: the first has 2 parameters and computes the average of the 2 values, the second has 3 parameters and computes the average of the 3 values
- So these methods will differ in terms of the number of parameters and the implementation

```
public static double average(double d1, double d2) {
    return (d1 + d2) / 2;
}
```

```
public static double average(double d1, double d2, double d3) {
    return (d1 + d2 + d3) / 3;
}
```

Method overloading

- Let's now implement a method called **multiply**;
- When we specify the parameters – the data used by the method - we can specify either 2 whole numbers in which case will carry out mathematical multiplication, or a string and a whole number in which case it will return another string multiplied.
- So these methods will differ in terms of the data type of parameters and how they are implemented.

```
public static int multiply(int a, int b) {
    return a * b;
}
```

```
public static String multiply(String a, int b) {
    String newString = "";
    for (int i = 0; i < b; i++) {
        newString += a;
    }
    return newString;
}
```

Method overloading

- Let's now implement a method called **printFormated** which prints a given message right-justified a field of a specific number of characters; so it defines 2 parameters.
- When we specify the parameters - we can specify the message first, or the number of characters first, in case we forget the order...
- So these methods will differ in terms of the order of parameters

```
public static void printFormatted(String message, int numOfChars) {
    System.out.printf("%" + numOfChars +"s", message);
}
```

```
public static void printFormatted(int numOfChars, String message) {
    System.out.printf("%" + numOfChars +"s", message);
}
```

Method overloading

```
public static void main(String[] args) {
    // calling the first set of 2 overloaded methods
    System.out.println(average(2.1, 3.2));
    System.out.println(average(2.1, 3.2, 7.7));

    // calling the second set of 2 overloaded methods
    System.out.println(multiply(1, 7));
    System.out.println(multiply("Java", 7));

    // calling the third set of 2 overloaded methods
    printFormatted("hello there", 20);
    printFormatted(20, "hello there");
}
```

```
2.650000000000004
4.333333333333333
7
JavaJavaJavaJavaJavaJavaJava
          hello there          hello there
```

- Exercise:
- place the main and the methods from the previous 3 slides in a class and run it
- Add several other calls with values of your choice.

Your turn

- Using the previous class and the overloaded methods average, what will each of the following statements in the main method output?
  - **System.out.println(verage(3.0, 4.0));**

  // 3.5
  - **System.out.println(average(3, 4));**

  // 3.5
  - **System.out.println(computeAverage(3, 4, 5));**

  // 4.0
  - **System.out.println(computeAverage(3.0, 4.0, 5.0, 6.0));**

  // it will not compile because there is no overloaded version that takes 4 doubles

- Write 2 overloaded methods for drawing dashes on the same line to the screen: one that always draws a fixed number of 100 dashes, and one that draws a variable number of dashes, depending on what number is passed when called. In what way are these methods overloaded??