Methods
(a.k.a. functions a.k.a. procedures)

1 Motivation: why use methods?

- modularizing code
- reusing code
- easier debugging

2 Defining Methods

syntax:

```
modifiers returnType methodName ( list-of-parameters )
{
    method body
}
```

returnType can be any of the data types that we learned so far, or void; void methods do not return a value

list-of-parameter is a comma separated list of the form:

type1 var1, type2 var2, ..., typeN varN

modifiers specify how the method can be accessed (for now will always use public static)

methodName has to follow the naming rules for identifier
3 Calling Methods

Calling a method results in executing the code in that method.

- value returning methods are used like values, for example,
  ```java
  int smaller = min ( num1, num2 );
  ```
- void methods are use like statements (in fact, you used these already), for example,
  ```java
  System.out.println("Hello world");
  ```

**Syntax:**

```
methodName( list-of-arguments );
```

**Activation record** = *activation frame* - information about a method that contains: values of parameters and (local) variables. Whenever a method is called an activation record is placed in the area of memory called *stack*.

4 Passing Parameters by Value

- When the function is called, the values/variables specified in the list-of-arguments must match the list-of-parameters from the function definition in
  - order,
  - number,
  - compatible type.
- In Java all parameters are **passed to methods by value**. This means that the values of the argument variables are copied to variables specific to the method.

What is the output of the following code?

```java
public class Increment {
    public static void main ( String[] args ) {
        int x = 1;
        System.out.println("Before the call, x is " + x);
        increment(x);
        System.out.println("after the call, x is " + x);
    }

    public static void increment(int n) {
        n ++;
        System.out.println("n inside the method is " + n);
    }
}
```

5 Overloading Methods

**Overloading methods** - defining multiple methods with the same name, but different signatures. A **signature** of a method consists of its name and its parameter list.

How would you write a method that determines max of its two parameters? How would you write a method that determines max of its three parameters?

**Source Code:** MaxOverloaded.java demonstrates three methods all of which are called max, but their parameter lists are different.
6 Scope of Variables

**scope of variable** the part of the program in which the variable can be referenced (used)

**local variable** = variable with local scope - variable declared inside a method; method parameters are also local variables

**block scope** - variable defined inside a block of code (code surrounded by curly braces { } ) can be references only within that block; nested blocks cannot have repeating variables

**Example 1:** When blocks are not nested, the variable name can be reused.

```java
public static void main(String[] args) {
    ...                      
    int x = 5;
    System.out.printf("%d",x);
}                          
{                           
    int x = 7;              
    System.out.printf("%d",x);
}                           
}
...
```

**Example 2:** If blocks are nested, the re-declaring a variable with the same name will cause a compile-time error.

```java
public static void main(String[] args) {
    ...                      
    int x = 5;
    System.out.printf("%d",x);
}                          
{                           
    int x = 7;              
    System.out.printf("%d",x);
}                           
}
```

7 Method Abstraction and Stepwise Refinement

These are probably the most important concepts regarding Java programming that you learned so far.

**Method abstraction** - a person (a client/programmer/user) can use a method without knowing its implementation. The only thing needed should be the method’s signature (method name, list of parameters), its return type and its documentation. The author of the method can change its implementation and, as long as the header of the method remains the same, the client code does not need to be modified.

**Stepwise refinement** - when writing a large program, one should use a divide and conquer approach to decompose the problem into subproblems, which then get decomposed into smaller subproblems and so on, until the final set of problems is of manageable size. What is manageable size? The "small problems" should be easily described as a single task and should be easily turned into simple methods or few lines of code.

Source code: CalculatorWithoutMethods.java vs CalculatorWithMethods.java