EXERCISE 1-2

Methods

"A method is a named sequence of Java statements that can be invoked by other Java code." [2]
A method signature defines what you need to know about a method before calling it. It is the method specification and defines the API for the method. The signature of a method specifies the following:

- The name of the method
- The number, order, type, and name of the parameters used by the method
- The type of the value returned by the method
- The checked exceptions that the method can throw

A method signature looks like this:

```
modifiers type name (paramlist) [ throws exceptions ]
```

Modifiers are either zero or special keywords such as `public`, `static`, `private`, ...
Type defines the return type of the method. If the method does not return a value, type has to be `void`.
Exceptions are optional. They will be discussed later.

Methods without Parameters

Methods, in simplest form, do not have any parameters, but just a name. When a Java method expects no parameters, its parameter list is simply `()`, not `(void)`.
For instance, the following example displaying a headline

```
System.out.println("This is our headline!");
System.out.println("————————————————————");
```

can be declared as method in the following form:

```
static void printHeader() {   // method header
   System.out.println("This is our headline!"); // method body
   System.out.println("————————————————————");
}
```

Additonally, we can call this method as often as it is requested in the flow of a computer program.

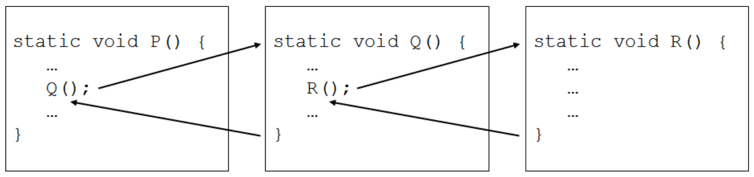
```
printHeader();
```

Each call executes the statements in the body of the method.
The declaration of a method consists of a header and a body. The header defines the name of the method and possible parameters. As `printHeader()` has no parameters the parameter list is empty (only round brackets). The keyword `static` defines that the method is called statically (method exists once during program execution). The keyword `void` defines that the method has no return value. The body of the method consists of two curly brackets, which hosts statements, and declarations of the method.
The following example shows how methods are embedded in the flow of a computer program.

```
class Program {
   static void printHeader() {   // method header
      System.out.println("This is our headline!"); // method body
      System.out.println("————————————————————");
   }

   public static void main(String args[]) {
      printHeader(); // call of method
      ...
      printHeader(); // call of method once again
      ...
   }
}
```

The method `printHeader()` is part of the class `Program`. Classes can have any methods, which are called by `main` or other methods.
The following example shows a method `P` that calls a method `Q` that calls the method `R`:



A method exchanges the call of another method with the statements of this method, and continues after this method call.

Naming Conventions for Methods Methods can have any names. However, for reading comprehensions, it is appreciated that the name of a method expresses its activity. Consequently, the name of a method should start with a verb. In Java, the name of a method starts with a small letter. In case that a name consists of several words, each following word should start with a capital letter.

Parameters

Parameters are values, which are submitted from a caller to a method. Then, the method can work with these values. The use of parameters enables to call methods in different contexts.

There are formal and active parameters.

A *formal parameter* belongs to the declaration of a method. Formal parameters are treated as variables, they are named memory cells. Formal parameters are local to the method in which they are declared. They are only known in this method.

An *active parameter* is a value that is committed by calling of a method. Active parameters (or arguments) can be of any expression.

During the call of a method, active parameters are handled over to formal parameters. There, the following actions take place:

- Expressions of active parameters are computed.
- The value of these expressions are assigned to the corresponding formal parameters.

The formal parameter gets a copy of the active parameter. Changing the value of the formal parameter in the method, just changes the copy, not the active parameter (might be a constant, variable or expression).

In Java, methods can have a variable-length argument list. Called vararg, these methods are declared such that the last (and only the last) argument can be repeated zero or more times when the method is called. The vararg parameter can be either a primitive or an object. An ellipsis (...) is used in the argument list of the method signature to declare the method as a vararg. [4]

The syntax of the vararg parameter is as follows:

```
public static void setDisplayButtons(String... names) {
    for (int i=0; i<names.length; i++)
        System.out.println(names[i]);
}
```

The method can be invoked as follows:

```
setDisplayButtons("About");
setDisplayButtons("About", "Contact", "Products");
```

Functions

Methods, which return a value, are called functions. Methods, which do not return a value, are called procedures.

The following example, maximum of two numbers, illustrates the declaration of a function in Java:

```
static int max (int x, int y) {
    if (x > y) return x; else return y;
}
```

The data type of the resulting value has to be declared in a function (here: `int`). The resulting data type has to be placed before the name of the method. The result of a function is delivered by the keyword `return`. Normally, at the end of the sequence block of the method. A function has to have a `return`-statement. Otherwise, there will be a compile-time error.

Functions can only return a single value. If more than one value is requested, we will have summarize several values in an object.

The `return`-statement is also allowed in procedures. In contrast to functions, it does not return a value but escaping the method.

Functions should be implemented without provoking side effects. That means that a function should not change global variables but just delivering a return value.

Local and Global Names

Beside statements, a method can contain declarations as well. Any declared name (and the formal parameters) in a method is local to this method. All outside declared names of this method are global.

Local Variables In Java, variables and constants can be declared in a method. It is not allowed to declare further methods in a method. The following method declares the three local variables `x`, `y`, and `z`:

```java
static void P (int x) {
   int y;
   float z;
   ...
}
```

These three variables can only be used in method `P`. They are not visible outside the method. Memory will be allocated each time the method is called. At the end of the method, memory will be released. Consequently, local variables survive only during execution of their declaring method.

Global Variables A variable will be global, if it is declared outside a method (or on class level). The following example declares two global variables `a`, and `b`:

```java
class Program {
   static int a;
   static float b;

   static void P (int x) { ... }
   public static void main (String args[]) { ... }
}
```

Global variables can be used in any method of the class `Program`. So, the variables `a`, and `b` can be used in `P` and `main`. The keyword `static` can be used for delcaring global variables but this is not a must. Memory for the static variables `a`, and `b` will be allocated at the beginning of program execution and will be released at the end of the program. So, the values of global variables are valid during whole program execution. They keep their values also in case of method calls.

Local Constants Java allows to declare local constants in methods.

```java
static void P() {
   final boolean ADDED = true;
   ...
   if (ADDED) System.out.println(...);
   ...
}
```

Normally, local constants are declared without the keyword `static`.

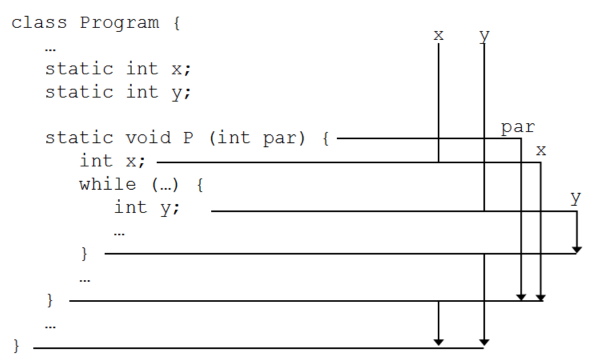
Local variables should be used instead of global variables. Global variables should be used if they are requested in several methods. Besides, a global variable makes sense if it has to keep its value over several method calls.

Visibility of Variables

The visibility (dt. Sichtbarkeitsbereich) or scope (dt. Gültigkeitsbereich) of a variable describes the piece of a program where a variable can be accessed. This ranges from its declaration to the end of the block in which the variable was declared. In case of a

- local variable of a method: from its declaration till the end of the method.
- global variable of a class: from its declaration till the end of the class.

The scope of a global variable will be interrupted if a local variable of the same name is declared. The local variable hides the global variable. The following example illustrates the scope of variables:



```
class Program {
   …
   static int x;
   static int y;

   static void P (int par) {
      int x;
      while (…) {
         int y;
         …
      }
      …
   }
   …
}
```

In Java, global variables and names of methods are visible before declaration. So, Java allows calling methods, which are declared later in the program.

Lifetime of Variables

If a variable is not visible, it does not mean that this variable does not exist as well. It might be covered by other variables, and it might be visible afterwards. The lifetime of a variable depends on being a local or a global variable. Global, static variables are created at the beginning of the program, and live till the end of the program. Local variables are created at the beginning of its declaring method, and live till the end of this method.

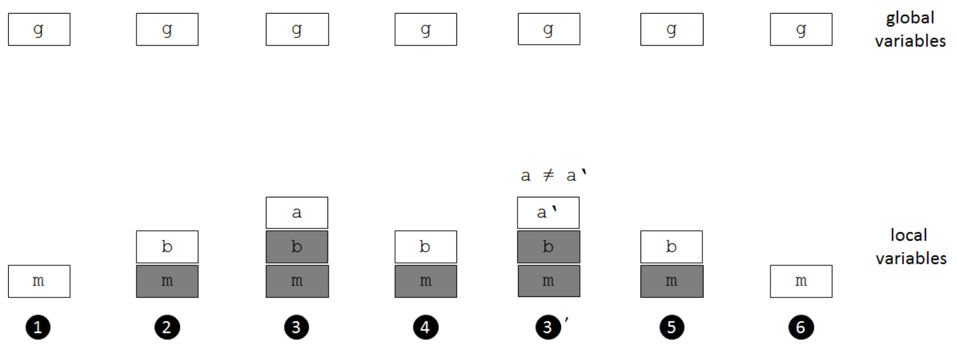
The following example illustrates lifetime and scope (white boxes) of variables:

```
class Program {
   …
   static int g;

   static void A() {
      int a;
      … ❸ …
   }

  static void B() {
      int b;
     ❷ … A(); … ❹ … A(); … ❺
   }

   public static void main (String arg[]) {
      int m;
     ❶ … B(); … ❻
   }
}
```



Local and global variables allocate different memory cells. Memory of global variables is of fix size. Memory of local variables can grow and shrink. The above example works as follows:

1. The program starts. Memory is allocated for the global variable `g`. This variable lives till the end of the program. Additionally, the method `main` is called, whereby memory for its local variable `m` is allocated. Both, `g`, and `m` are visible.
2. The call of method `B` provokes memory allocation for its local variable `b`. Now, the local variables `b`, and `m` exist. However, the variable `m` is not visible because it was declared in `main`, and the program is currently in method `B`. The variable `m` still exists, it will be re-visible when `B` returns back to `main`.

3. The call of method `A` provokes memory allocation for its local variable `a`. Now, the variables `a`, `b`, `m`, and `g` exist but only `a` and `g` are visible.

4. `A` is terminated. Memory of its local variable `a` is released. Now, the variables `b`, `m`, and `g` exist but `b` and `g` are visible.

3'. `A` is called once again. Memory for its local variable `a` is provided. However, this is not the same memory cell than before. The local variable `a` is a new variable, the old value of `a` was not kept.

5. Coming back from A, the variables `b`, `m`, and `g` are existing.

6. Coming back from `B`, the variables m and `g` are existing. With the termination of `main`, its local variable `m` will disappear. With the end of the program, the global variable `g` disappears as well.

Overloading

Normally, all names, which are declared in a block have to be different. Nevertheless, methods are exceptional. Two methods can have the same name in case that their list of parameters is different. The parameters have to differ either in amount or in data type. Consequently, it is possible to write different methods for displaying data:

```java
static void print (int i) { ... }
static void print (float f) { ... }
static void print (int i, int length) { ... }
```

We say that the method is overloaded. The process is called **overloading**. So, a method may have different meanings. If the method is called the method fitting to the current parameter will be chosen.

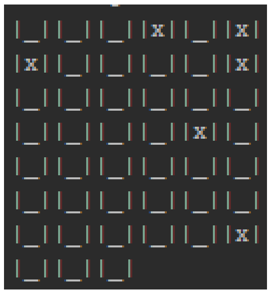
```java
print(1000); // call method print(int i)
print(7.68); // call method print(float f)
print(1000,5); // call method print(int i, int length)
```

Refactoring

"Refactoring is the process of changing a software system in a way that does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence, when you refactor, you are improving the design of the code after it has been written." [3]

"Refactoring changes the programs in small steps, so if youmake a mistake, it is easy tofind where the bug is." [3]

Displaying Lotto Write an algorithm (in main(...)) that displays your filled in lotto ticket (dt. Lottoschein) as follows:



```
|_||_||_||x||_||x|
|x||_||_||_||_||x|
|_||_||_||_||_||_|
|_||_||_||_||x||_|
|_||_||_||_||_||_|
|_||_||_||_||_||_|
|_||_||_||_||_||x|
|_||_||_|
```

- "|_|" ... represents a not typed number (1 to 45)
- "|x|" ... represents a typed number, e.g. 4 6 23 7 12 42
- Use the argument list (String[] args) to read the filled in lotto numbers.

After implemented the algorithm, refactor with the help of IntelliJ. Mark the corresponding source code lines and use "Refactor - Refactor This - Method".

Do It

1. *Triangles.*

   Write a method/function `area(a, b, c)` that gets the side lengths of a triangle as parameters, calculates, and returns the area of this triangle by using the formula of Heron:

   $s = \frac{a+b+c}{2}$

   $area = \sqrt{s(s-a)(s-b)(s-c)}$

   The square of a number `x` can be calculated as follows: `Math.sqrt(x)`.

2. *Rectangles.*

   Write a program `Rectangle` that identifies the (interrelated, dt. gegenseitige) location of two rectangles. The rectangles are located anywhere in the Cartesian coordinate system. A rectangle is defined by two corners A(ax,ay) and B(bx,by). The coordinates are whole numbers. The program delivers one of the following terms:

   - disjoint. The intersection of the two rectangles is empty. There is no common point.
   - same. Location and size of the two rectangles is the same.
   - contained. The intersection of the two rectangles is determined by one of the two rectangles. All points of one rectangle are part of the second rectanlge but not vice versa.
   - aligned. The intersection of the two rectangles is a line. All common points are located along a line.
   - touching. The intersection of the two rectangles is a point. There is just this one common point.
   - intersecting. The intersection of the two rectangles is another rectangle with area > 0.

   The program reads eight numbers from the shell: ax, ay, bx, by (first rectangle) and cx, cy, dx, dy (second rectangle).

   Example. Rectangle X: A(2,2), B(7,5) and Rectangle Y: C(3,4), D(0,6). Then, the program returns the term intersecting.

3. *Conversion of binary numbers.*

   Write a method/function `convertBinaryToNumber(sequence)` that gets a sequence of zeros and ones, and converts this binary number into the corresponding decimal number.

4. *Ask the compiler.*

   Answer the following questions by trying them out.

   (a) What happens if you invoke a value method and you do not do anything with the result; that is, if you do not assign it to a variable or use it as part of a larger expression?

   (b) What happens if you use a void method as part of an expression? For example, try
   `System.out.println("hello!") + 7;`

Vocabulary [1]

In the following, we define several terms, which are useful for further understanding.

argument A value that you provide when you invoke a method. This value must have the same type as the corresponding parameter.

invoke To cause a method to execute. Also known as “calling” a method.

parameter A piece of information that a method requires before it can run. Parameters are variables: they contain values and have types.

flow_of_execution The order in which Java executes methods and statements. It may not necessarily be from top to bottom, left to right.

parameter_passing The process of assigning an argument value to a parameter value.

local_variable A variable declared inside a method. Local variables cannot be accessed from outside their method.

signature The first line of a method that defines its name, return type, and parameters.

void_method A method that does not return a value.

value_method A method that returns a value.

return_type The type of value a method returns.

return_value The value provided as the result of a method invocation.

overload To define more than one method with the same name but different parameters.

References

[1] Allen B. Downey and Chris Mayfield. *Think Java.* O'Reilly Media, 2016.

[2] Benjamin J. Evans and David Flanagan. *Java in a Nutshell.* O'Reilly Media, 2015.

[3] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 2018.

[4] Robert Liguori and Patricia Liguori. *Java 8 Pocket Guide.* O'Reilly Media, 2014.