In this second session we first have a look at the concept of *constructor*, i.e., a special kind of method common to every class whose task is to construct new objects of that class. If we don't define and implement it, Java does it for us, providing a *default constructor*, with no arguments. The default constructor is what you implicitly call when you construct a new object of the class by writing

```
NameOfTheClass nameOfTheObject = new NameOfTheClass();
```

The constructor has indeed the same name of the class.

However, we can implement constructors as well, also providing them a not empty argument list. Moreover, we can implement two different constructors for the same class, with different argument lists (for example, one empty, one not empty): this brings us to the concept of *overloading* of methods, and this applies not only to constructors, but to all methods. We say that we are *overloading* a method when we define and implement a method with same name, but with different argument list. We usually do this when we ask our methods to perform the same action (for example, initialise the object) in different ways.

We then illustrate the uses of `this`, that returns a reference to the object that calls the method (possibly the constructor) where `this` appears. We see that this is used inside a constructor in order to solve possible name clashes and that can be returned at the end of a method in order to allow for example for multiple calling. Another use of `this` (only inside constructors, and only once per constructor) is to call another constructor.

The last topic of this session regards `static` methods and `static` fields. When you say something is `static`, it means that it is not tied to any particular object instance of its class. So even if you have never created an object of that class you can call a static method or access a piece of static data. You can call static methods just by typing the name of the class, without specifying the object which calls that method. Since static methods don't need any objects to be created before they are used, they cannot directly access non-static members or methods (since non-static members and methods must be tied to a particular object).

In order for you to be able to have a look at the code also after the lecture, this is a list of the classes we see, in the order we look at them and with reference to the topic they are supposed to cover.

- `com.andreamazzon.session2.oophelloworldwithconstructor`: first example of how we can implement the default constructor, i.e., the constructor with no arguments.
- `com.andreamazzon.session2.treesconstructor`: easy example of a constructor with arguments. In particular, the constructor takes an `double` as argument, which sets the height of tree. Here we also see what happens if we don't initialize a primitive field of a class and we then use it.
- `com.andreamazzon.session2.bankaccount`: example of constructor overloading. The default constructor sets the commission equal to zero, whereas the overloaded constructor takes as a `double` as argument which sets the commission.
- Code in `com.andreamazzon.session2.oophelloworldwithoverloading`: here we observe a first example of method overloading, together with another example of constructor overloading. The class `MessageWithOverloading` has two methods, with same name but different argument list. They both print a `String`, but in different ways: one getting it from a class field and the other one as an argument.
- `com.andreamazzon.session2.triangleperimeter`: other example of method overloading. Here you can see how we can use method overloading to make our like easier.
- Code in `com.andreamazzon.session2.cosine`: again an example of method overloading, computing the cosine of an angle. If the value of the angle in radians is a multiple of π, we can compute it without using the `Java` method. Here you see how method overloading is of course performed also when it's not the number of arguments that changes, but their *type*.

- Code in `com.andreamazzon.session2.power`: this is an exercise for you, about method overloading. The class `Power` is devoted to the computation of the exponentiation a^b, where a is the base and b the exponent. However, this produces a complex number if a is negative and b is not integer, and in this case the `Math.pow(base, exponent)` of Java returns `NaN` (not a number). This might be confusing for an unexperienced user! Thus we want to have two different implementations of the method `computePower`: one where the base is `double` and the exponent `int`, since in this case we have no problems (and we just compute the power with the `Math.pow(base, exponent)` method) and one when the base is `double` and the exponent `double`: in this case, if the base is negative and if the exponent is not an integer number (this is not the case, for example, if `exponent = 2.0`) we want to print a warning message, without calling the Java method. If not, we call the Java method to compute the exponentiation. **Hint:** you can verify if a `double variable` is actually an integer by cheking if `variable == Math.floor(variable)`.

- `com.andreamazzon.session2.divisiblewithconstructor`: first example of the use of `this` in order to get the reference of the object which is calling the constructor. This permits to solve name clashes between the name of the field and the name of the variable given as an argument inside the constructor.

- `com.andreamazzon.session2.bankaccountwiththis`: this is the example we have seen above for the constructor overloading, revisited: we show again an example of the use of `this` to solve name clashes inside the constructor, together with another use of `this`, i.e., to call a constructor from another overloaded constructor. This permits to avoid duplicating code.

- `com.andreamazzon.session2.divisions` here you find a very easy example of a class with a `static` method, and an example about how it can be called.

- Code in `com.andreamazzon.session2.staticexample`: example of the functioning of `static` fields and methods. We see that the value of `static` fields is shared by all the instances of the class where the method is defined.

- Code in `com.andreamazzon.session2.mortgages`: here we show how this last feature of `static` fields can be applied in an useful way.