Apart from the very beginning, when we have a look at how arrays are managed in Java, this session is fully dedicated to a very important topic in Java and in object oriented programming in general, i.e., access modifiers, implementation hiding and encapsulation. Not everything in Java can be accessed from everybody and from everywhere: for now (waiting for Session 4, when we will see inheritance and then another access modifier, `protected`) we see that fields and methods of classes can be:

- `public`, i.e., can be called and accessed from everywhere (by `import` if they get called from outside the package);
- with *package access*, i.e., they can be called and accessed from classes of the same package. In this case, they have no access modifier;
- `private`, i.e., can be called and accessed only from objects of the same class (or from `static` methods of the class).

Classes may also be `private`, but this makes sense only for `inner classes`, that we will see later on.

Building things such that not everything is public makes sense in order to separate the part of the program an user can interact with (for example, calling methods and getting data) from the implementation. Hiding the implementation, you are free to change anything that is not public (e.g., package access, protected, or private) without breaking client code. In other words, this is useful to break up the playing field into class creators (those who create new data types) and client programmers (the class consumers who use the data types in their applications).

Wrapping data and methods within classes in combination with implementation hiding is often called *encapsulation*. We will see quite many examples of how encapsulation works and why it is useful.

In order for you to be able to have a look at the code also after the class, 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.session3.arrayexample.ArrayRandom`: this class provides some examples to understand the way arrays are defined, initialised and printed in Java. Pay attention to what happens if you assign an array to be equal to another. Look at the example of the use of `clone()`.
- `com.andreamazzon.session3.useful.Print`: in this class you can see the implementation of some methods to print without calling `System.out.print`. These methods are `public`, because they are supposed to be used from classes from outside the package where they are defined. For example, they are used in `com.andreamazzon.session3.simplerprinterexample.SimplerPrinterExample`, where we can see a first example about how we call a `public` method from outside the package of the class where it is defined. Example of `import` and `import static`.
- Code in `com.andreamazzon.session3.packageimporting`: besides another example of `import` and `import static`, here we see an example of possible conflicts when we import classes with the same name from different packages, or different packages that have a class with same name: in this case, `java.util` and `com.andreamazzon.session3.useful` both have a class `Vector`. This issue can be solved importing just one class directly and referring to the other when we need it by typing the name of the package where it is, look at row 42 in the class `ImportsAndConflicts`.
- Code in `com.andreamazzon.session3.accesslevels`: some examples of access modifiers for fields and methods of a class, and of a class with no access modifier, i.e., package access only. The way this works is also tested from outside the package from

  `com.andreamazzon.session3.otherpackage.AccessTestOutside`.

- `com.andreamazzon.session3.privateconstructor.countingObjects`: this is an example of how a private constructor can be used in order to prevent the number of objects of one class to exceed a limit we set. In this case, we want to construct only one object of the class `PrivateConstructorClass`.

- `com.andreamazzon.session3.privateconstructor.enhancedmortgages`: this is an exercise for you, again on private constructors. You can see that this example is very similar to the one in `com.andreamazzon.session2.mortages`. Again, we don't want to exceed a given budget lending money to clients. Last time we just printed a warning message. Now we really want to prevent the construction of new objects if the total amount of money exceeds the budget. You can find a `private` constructor. Your task is then to implement a `public static` method

  ```
  constructMortgage(String name, int age, double amount)
  ```

  that returns a reference to an object of type `Mortgage`. In particular, it will:

  - update `loansSum`, summing `amount` to the old value;
  - call the `private` constructor if `loansSum <= budget`;
  - print a message saying that we have no more money and return `null` if `loansSum > budget`.

- `com.andreamazzon.session3.encapsulation.travelspeed`: this is an example about *encapsulation*. In particular, `setters` and `getters` are shown, in combination with `private` fields: `private` fields cannot be accessed directly because we want to have *control* on the way they are handled, so `setters` and `getters` permit the users to set and get their value undirectly. In this case, for example, the user understands km but we work with miles, so a conversion has to be performed.

- `com.andreamazzon.session3.encapsulation.gas`: other example of encapsulation. In this case, a client only understands degrees Fahrenheit, but we have to work with Kelvins for example in order to return the pressure of the gas with Gay Lussac law.

- `com.andreamazzon.session3.encapsulation.complexes`: this time we work with complex numbers, for a client who only understands their cartesian representation, i.e., with real part and imaginary part. However, we see that some operations (for example, the computation of the absolute value of a complex number and the product of two complex numbers) are better performed with the polar representation. So we work with polar coordinates and encapsulation to let the implementation be hidden from the eyes of the user.

- `com.andreamazzon.session3.encapsulation.logarithm`: exercise for you again: a client wants to compute the logarithm of a number. Anyway, we want to prevent him/her to give a negative input and then get unwanted behaviours. The field `double number` of `LogarithmCalculator` is thus `private`. You have to implement a `getter` to get the value of `number` (this is extremely easy!) and a setter `public void setNumber(double number)` which sets `this.number = number` if `number > 0` and prints a warning message if not.

- `com.andreamazzon.session3.encapsulation.lazyinitialization`: this last example shows you a class simulating a sequence of pseudo random natural numbers using a so called *linear congruential generator*. In particular, we want that the sequence is generated only when it is needed (this is lazy initialization) and only once. The method `generate()` should therefore be `private`, as well as the fields of the class.