This session is devoted to two ways of reusing the implementation, one of the greatest advantages provided by object-oriented programming: *composition* and *inheritance*. *Composition* regards the creation of objects of an existing class inside a new class, *delegating* to those objects the implementation of some methods, or part of it. For example, we see a class `Car` which has an object of type `Engine`, to which it *delegates* the implementation of the method `start`: with composition, you reuse the functionality of the code. On the other hand, with *inheritance* you create classes which *extend* existing classes, adding some fields or methods or, maybe even more importantly, changing the implementation of existing methods: this last feature is known as method *overriding*.

It's not always easy to recognise if you need inheritance or composition for the solution of a specific problem. We can say that inheritance involves a relation of type *is-a* (for example, a `Circle` is a `Shape`, and then the class `Circle` inherits from the class `Shape`) whereas composition involves a *has-a* relation (a `Car` has an `Engine`).

However, the borders are not always that clear. Generally speaking, composition may be preferred over inheritance because it is often simpler and more flexible: for example, the member objects of your new class are typically private, making them inaccessible to the client programmers who are using the class.

Just to give a first intuition about the differences between the two approaches and their functionalities, we will see some examples of problems where the use of inheritance is the best solution and some others where composition comes into play.

One of the most striking features of inheritance is *polymorphism*: suppose you have three derived classes which extend (= inherit from) a base class, and suppose they all override a method of the base class, implementing it in different ways (one different way for derived class). So you want that objects of the derived class execute the method in a different way depending on their type. However, say you write a method somewhere that involves objects of the base class, making them call the overridden method. You don't want to change this last code depending on the derived type of objects, i.e., here you want to treat an object not as the specific type that it is, but instead as its base type. On the other hand, you want to treat it as its specific type when it calls the overridden method. This problem can be solved by treating an object as its base type at compilation time, and as its specific type at run type. This behaviour is known as *late binding* and allows for *polymorphism*: a method involving another, overridden method in its implementation can be written exactly in the same way for all the derived types, returning however different outputs depending on the specific type. We will see some examples of this use.

Inheritance introduces a further access modifier, i.e., `protected`: as we see in some examples, `protected` gives the second least restrictive access to methods and fields: `protected` methods and fields have package access, and can moreover been accessed and called in methods of classes which extend the class where they are defined.

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.session4.usefulmatrices.UsefulMethodsMatricesVectors`: this class is not about reusing of implementation, but it contains useful methods that deal with arrays and matrices (arrays of arrays). It will be used in `com.andreamazzon.session4.composition.binomialmodel`.

- Code in `com.andreamazzon.session4.inheritanceandconstructors.basicexample`: the code in this package gives a very simple example of the syntax of inheritance and of the way the default constructors of the derived and of the base class are called. In particular, you can note that the default constructor of the base class is implicitly called when an object of the derived class is created.

- Code in `com.andreamazzon.session4.inheritanceandconstructors.sportsman`: here we see how things work when the base class only has constructors with arguments. We have seen that in this case the default constructor cannot be called. For this reason, inside the constructor of the derived class we have to call the constructor of the base class. This is done with the keyword `super`.

- Code in `com.andreamazzon.session4.testingprotection`: in this package we illustrate how the `protected` access modifier works: the methods of the class `DerivedClass` have access to the `protected` methods and fields of the parent class `ProtectedOtherPackage` in `com.andreamazzon.session4.protectedotherpackage` even if they are not in the same package. On the other hand, in the `main` method of `TestClass` we can access `protected` fields and method of the class `ProtectedSamePackage`, because `protected` also gives package access.

- Code in `com.andreamazzon.session4.phones`: the code here gives a simple example of how an object of a derived class is able to call methods and to access fields of the base class, plus possibly some more methods which are defined in the derived class. Here we also see what happens if we construct an object with the constructor of the derived class, but giving it a reference to the base class. That is, if we attach an object of the derived class to a reference of the base class.

- Code in `com.andreamazzon.session4.overridingandoverloading.videogame`: the classes here illustrate the difference between *overriding* (give another implementation of a method of the base class) and *overloading* (defining and implementing a method with same name of an existing one but with different argument lists). We see that the derived class can both override and overload a method of the base class. Note the `@Override` annotation above the overridden method.

- Code in `com.andreamazzon.session4.polymorphism.amphibians`: this is an example of *polymorphism*: you can see how the method `behavior(Amphibian amphibian)` of the class `Amphibian` accepts as an argument a reference of type `Amphibian`, making it call three methods which are overridden from the derived classes `Frog` and `Toad`. Running the example, you can see that you can also pass it objects of type `Frog` and `Toad`, which then execute the methods in their specific way. Here *late binding* comes into play: the point is that at compilation time Java looks at the reference of the objects (the first name of the class you write when you construct the object) and at running time it looks at the specific type of the object which is attached to that reference. At compilation time, when we call the method `behavior(Amphibian amphibian)` passing it an object of type we have constructed as `Frog frog = new Frog()`, it accepts it because:

  - Due to late binding, `behavior(Amphibian amphibian)` accepts any object which has a reference of type `Amphibian`;
  - Because of that, Java is able to recognize that it should *upcast* `frog` to `Amphibian`, i.e., give it an `Amphibian` reference: this is what it does. It's like we had written `Amphibian frog = new Frog()`.

  In this way, the objects are treated as their base type at compilation time, and as their derived type at running time.

- `com.andreamazzon.session4.polymorphism.shapes`; this is another example of polymorphism, and here you have to work a little bit. We have a class `Shape` with a method `computeArea()`, that we are not able to implement in a good way for a general `Shape` object (we will see next time how such methods should be handled). We then have three classes extending `Shape`, i.e., `Triangle`, `Circle` and `Square`. Override `computeArea()` for the three classes, giving an implementation according to the type of the shape. You have to add specific sub-class fields (for example, `height` and `base` for the triangle), also implementing the constructors setting those fields. Note then how polymorphism appears in the `main` method of `ShapeAreaTest`: the method `computeArea()` can be called by any object of type `Triangle`, `Circle` and `Square` returned by the `RandomShapeGenerator`.

- `com.andreamazzon.session4.composition.car`: first example of composition. The class `Car` has an object of type `Engine`, to which it *delegates* the implementation of the method `start`.

- `com.andreamazzon.session4.composition.binomialmodel`: this is another example involving composition: we use the code of `LinearCongruentialGenerator` in the class `BinomialModelSimulator` in order to simulate a binomial model. In particular, we delegate to the methods of the object of type `LinearCongruentialGenerator` the generation of random natural numbers on which we base to simulate our process. We can say that our simulator *has* a `LinearCongruentialGenerator`.