This session is mainly concerned with abstract classes and interfaces, besides a brief view of `enum` types. In particular, we see how abstract classes and interfaces can be useful, also looking at more examples on polymorphism.

Last time we saw an example of polymorphism in `com.andreamazzon.session4.polymorphism.shapes`. However, we saw that we had to give a dummy implementation of the method `computeArea()` in the base class `Shape`, because we are not able to compute the area for a general shape, but only for a specific shape like `Circle`, `Square` or `Triangle`.

When the base class has a method which can be implemented in a meaningful way only in derived classes, we define the method to be `abstract`, and the class to be `abstract` as well: we just give the returning type, the name and the argument list of the method, without implementation. We see a first example of how this works in

`com.andreamazzon.session5.abstractclasses.basicexample`,

and we look at how this can lead to a better solution of the shape example in

`com.andreamazzon.session5.abstractclasses.shapes`.

In both the examples you see again polymorphism: we have methods whose argument list is represented by objects of the parent type, accepting anyway objects of the classes extending those abstract classes.

But what if we want a class to inherit from two parent classes? This is not possible, mainly for a reason highlighted in

`com.andreamazzon.session5.nomultipleinheritance`.

The two parent classes could indeed have both a method with same name, same argument list but different implementation, or/and a field with same name initialized with different values in the two classes. In this case, it would be impossible to say which implementation should be inherited from the derived class, or to which value the field should be initialized.

Name conflict with methods would not be a problem, however, if the two parent classes are *fully abstract*, i.e., if they do not have any implemented method, so, only abstract methods. Interfaces are what we are looking for.

An interface allows the creator to establish the form for he *public part* of a class: method names, argument lists, and return types, but no method bodies. An interface can also contain fields, but these are implicitly static and final. In this sense, an interface provides only a form, but no implementation. We write that a class `implements` an interface, and you can think about this as if it `extends` the fully abstract class represented by that interface. Any class *implementing* an interface must give the implementation of all the methods defined in the interface, or define them to be `abstract`. Moreover, they are implicitly `public`: you don't have to specify any access modifier when you list them in the interface and you have to declare them as public in the classes implementing the interface. This makes sense, because an interface establishes indeed the frontend face with the outside of classes implementing that interface. Having a look at an interface and seeing that a class implements that interface, you already know which methods you can ask an object of that class to call.

We see a first example of the use of interfaces in

`com.andreamazzon.session5.interfaceexample`,

also providing you the syntax you have to use when you construct classes implementing an interface. Here we see also an example about how polymorphism works with interfaces, and how downcasting can be dangerous at running time.

In

`com.andreamazzon.session5.multipleinterfaceimplementation.animals`

we see that it is possible for a class to implement more than one interface, something that is prevented to classes with inheritance. Still, we note that there can be conflicts if the two interfaces share a (`static`, `final`) field giving it different values.

Moreover, as we see in

`com.andreamazzon.session5.inheritinginterfaces`,

an interface can extend more than one interface. However, problems may arise when a class implements two interfaces which both have a method with same name, same argument list but different return type. We see an example of such a behaviour in

`com.andreamazzon.session5.nameclasheswithinterfaces`.

In the packages

`com.andreamazzon.session5.packageaccessinterfaces`

and

`com.andreamazzon.session5.anotherpackage`

we look at an example that might seem rater senseless at first: a public class implementing a package access interface. This might be actually the way to proceed if one wants to prevent programmers to create new classes of the interface in another package, still being able to create objects of the classes of the interface (which are defined in the package of the interface).

(Question for you now: can it it make sense, and if yes in which cases, to have a package access class with public methods?)

At this point, we do an excursion from interfaces in order to look at a new type, i.e., `enum`: this is useful when you have to group together and use a set of constant values (enumerated types, typically strings), and you can see a first example in

`com.andreamazzon.session5.enumtype.currencies`.

Another example of `enum` type, combined with interfaces and polymorphism, can be found in

`com.andreamazzon.session5.interfaceandpolymorphism.band`.

For every package listed here, you can find a more detailed description of the code itself and of the main topic shown in the `package-info` file.