

In this sixth and last session we see the use of **Generics**, error handling at run time with **Exceptions** and *inner classes*.

We start with **Generics**: we use this feature when we don't want to specify the type of a field of a class, or the type returned by a method, controlling at the same time that the type remains unchanged for a specific instantiation of the class where the field or the object are defined. This last requirement is the reason why we don't simply declare the type of the field, or the returning type of the method, to be **Object**, which is the largest class from which every other class inherits: in this last case, for one object we can set the type of the **Object** type field to be first **Integer** and then **Double**. If we want to prevent this behaviour, **Generics** is the right tool.

The code in the package

```
com.andreamazzon.session6.generics.containers
```

provides a first example of the syntax and use of **generics**: **GenericClass<T>** is a **generic** class with a field **T genericField** whose type is not specified when we define the class, but has to be the same for a specific instance of **GenericClass<T>**. Here **T** placed inside angular brackets is a placeholder for the type of **genericField**: if we want to create an object **newObject** for which **genericField** has type **Double** we write

```
GenericClass<Double> newObject = GenericClass<Double>.
```

Another application is seen in the packages

```
com.andreamazzon.session6.generics.genericinterfaces,
```

where a **generic** interface **GenericPointer** is defined with a method **next()** which returns an object of type not specified, but same for every object calling that method,

```
com.andreamazzon.session6.generics.fibonacci,
```

and

```
com.andreamazzon.session6.generics.readstring,
```

where two classes implementing **GenericPointer** are defined, one for which **next()** returns an **Integer** and one for which **next()** returns a **Character**.

The second topic is run time error handling with **exceptions**: Java exception handling is a way to guarantee that a run time error is noticed, and that something happens as a result. In particular, an **exception** is an object that is *thrown* from the site of the error and can be *caught* by an appropriate exception handler designed to handle that particular type of error. When you throw an **exception**, two main things happen. First, the **exception** object is created in the same way that any Java object is created: on the heap, with **new**. Then the current path of execution (the one you could not continue) is stopped and the reference for the exception object is ejected from the current context.

We see two examples: the first one, contained in

```
com.andreamazzon.session6.exceptions.commonexception,
```

shows a typical **exception** automatically thrown by Java, i.e., **ArrayIndexOutOfBoundsException**. Together with other **exceptions** automatically thrown by Java, you don't need to write an **exception** specification saying that a method might throw such an **exception**: they are so common that if you had to do so, your code would have been too messy.

On the other side, in

```
com.andreamazzon.session6.exceptions.divisions
```

we see how to **throw** and **catch** some **exceptions** that we write ourselves: these are classes extending the class **Exception**, and when they are *thrown* we create new objects of these classes. When they are *caught*, we ask these objects to call methods where we print the type of the error.

Finally, we have a look at *inner classes*: these are classes defined inside another class, exactly as we usually do. In particular, in

```
com.andreamazzon.session6.innerclasses.innerouteraccess
```

we see a first example of an inner class, and how private fields of the outer class can be accessed from the inner class and vice versa. We note that an object of the inner class must keep a reference to the particular object of the enclosing class that was responsible for creating it (and we see how).

This is not true for *nested classes*, that are **static** inner classes, meaning that you don't need an outer-class object in order to create an object of the nested class, so that the object of the nested class do not keep a reference to any object of the enclosing class. We see such an example in

```
com.andreamazzon.session6.innerclasses.nestedclasses.
```

Inner classes are the only classes that can be made **private**: as we show in

```
com.andreamazzon.session6.innerclasses.privateinnerclasses,
```

the creation of objects of **private** classes is possible only from inside the class where they defined. Here the **private** inner class **SequenceSelector** of **Sequence** implements an interface **Selector**. When an object of such a class is created in **SequenceTesting** by calling the public **getSequenceSelector()** method of **Sequence**, it is upcasted to **Selector**, and in this way everything not regarding the methods of the interface is hidden to the user.

In the package

```
com.andreamazzon.session6.innerclasses.anonymusinnerclasses
```

we have a look at how *anonymous* inner classes can be defined inside a class. In our case, the *anonymous* inner class defined inside the class **Letter** implements the interface **Envelope**.