Programming languages 9

Jan Dietrich

1. subtyping vs subclassing

Subclassing focuses on reuse or in other words incremental modification. That means some implementation of a superclass can be reused in subclasses. Another great explanatio we heard in the lecture is: "a subclass does everything a superclass does, but more".

Animal -> Duck

Consider a class `Animal` that has a property `age` and a method of `eating`. The subclass `Duck` reuses the property `age` and method `eating` but also has a method for `quacking`

Subtyping focuses on the principle of substituability. That means, that a specific type can be substituted by any of it's subtypes and the code execution should not run into errors. In terms of code we can imagine the example, where many classes implement a specific interface. Now we can define code that requires the implementation of the interface and any instance of the classes can be used (substituted).

```
Employee -> Developer
         -> Manager
```

Type `Employee` with subtypes `Developer`, `Manager`

If a specific code section expects an `Employee`, We can safely substitute it with an instance of a `Developer` or an instance of a `Manager` because both are subtypes of `Employee`

2.

In Figure 1 we see the class `java.lang.Object` which is a superclass for the class `X509ExtendedKeyManager`.

Little side note: Every class defined in Java is a subclass of this class `java.lang.Object` and inherits some class methods.

The class `X509ExtendedKeyManager` implements the interface `X509KeyManager`, which is a subtype of `KeyManager`

Since we know that subtyping has the idea of substituability we should know that `X509ExtendedKeyManager` therefore also implements the interface `KeyManager`

In other words whenever a code section expects an `KeyManager` instance, it's save to provide it an instance of the subtype `X509KeyManager`. So when a class implements the interface of the `X509KeyManager` it also implements the interface of the supertype `KeyManager`.

3.

In the Listing1 at the comment `Hint 1` we see the **coercion** polymorphism, We expect a floating point number but an integer can be used.

At the comment `Hint 2` we see the **parametric** polymorphism with the help of generics. The class `Bern` is polymorphic and is specified with a parameter in angle brackets `Bern<Integer>` so `Integer`

4.

In Listing2 we have an Double array `Double[]` we use a Number array `Number[]` and assign the Double array.

`Double` is a Subtype of `Number`. Therefore a `Double[]` is a subtype of `Number[]`

The `Number` array assignment accepts all subtypes of `Number` (e.g. `Double`) we can therefore call it **covariant**.