8.5 Rationale for subclasses and submethods

The *main rationale* for subclasses and submethods is to be able to represent *classification hierarchie*s. In the next section, we describe classification hierarchies, but we have already seen examples of such hierarchies. In section , the figure showing the relationships between `Datum`, `Reference`, `Value` and 'specializations' of `Value` is one example, and the figure showing `Date` and `TimeOfDay` as special `Value` types is another example.

There is also a *secondary rationale* that is of a *technical* nature in the sense of how subclass and submethods works. We elaborate on this technical rationale in the rest of this section, but you should keep in mind that the main rationale is the represent classification hierarchies.

The subclass mechanism ensures that an instance of a subclass has all the attributes defined in its superclass. This means that a reference to an instance of a subclass can be assigned to a reference variable typed with a superclass and the common attributes in the superclass may then be accessed through this reference variable.

The subclass mechanism makes it possible to write code that can handle objects of a givne class and objects being instances of subclasses of such a class. This is often referred to as the principle of *substitutability*.

As mentioned in section mkRef("Extended type rule"), we may write code that works for all `Account` objects including instances of subclasses of `Account`. The code is safe in the sense that that the compiler can verify that all attributes of an object being accessed are in fact defined for this object.

There is, however, an issue with code executed in subclasses and methods (virtual and non-virtual) defined in subclasses. For a given class, execution of a method on an instance of the class, has a specific effect on the object. If the method or class is virtual and extended in a subclass, the new code may 'spoil' the effect and this may be problematic.

As an example, the method `deposit` of class `Account` increments the balance of the `Account`-object. If `deposit` is virtual, we may extend the definiton of `deposit` in a subclass like `SavingsAccount` and here we may set `balance` to any value, which implies that `balance` may not have been increased. In general, this kind of code should be avoided. We want our subclasses to be *behavioural compatible* with their superclass. The compiler/language cannot guarantee that this is the case – it is up to the programmer to ensure this.

One of the advantages of the class/subclass mechanism, is that it supports *reuse of code*. For the bank system, the common code for `SavingsAccount` and `CreditAccount` is placed in class `Account` and needs thus not to be repeated in `SavingsAccount` and `CreditAccount`. This is in practice a major benefit of the class/subclass mechanism.

In most of the literature of object-oriented programming reuse is considered the main advantage of the class/subclass mechanism. As said, we think the main advantage is the ability to represent classification hierarchies – reuse is an important but secondary sidebenefit. In the next section, we explain what we understand as a classification hierarchy.