Chapter 7
Inheritance and Composition

There are numerous natural hierarchies in the real world. Also we, human beings, have created many systems that have hierarchies. In hierarchies, sometimes things flow from top to bottom, and at some other times, things propagate from bottom to top. Depending upon domains, these things can be properties, behavior, instructions, etc. When we write programs to solve problems related to hierarchies, it is natural to model real world hierarchies with some programming features. In this way, it is convenient to relate programs to the original problem. Also it is much easier to maintain the programs and hence reduces cost over a period of time. Scala provides features to represent hierarchies. Also it provides features to combine things, which maps to syntheses or compositions, in the real world.

Inheritance is a mechanism by which one entity can get resources from another entity; it can be used to model *is-a* relationships. In the real world, one entity can inherit from multiple other entities. It is also true that multiple entities can inherit from a single entity. Scala supports these real world scenarios through classes, inheritance, composition, and traits. Further, inheritance is like a parent–child relationship. One of the major advantages of inheritance is that it maintains the original relationship and promotes code reuse.

Composition is a mechanism to combine multiple entities to form a larger entity; it can be used to model *has-a* relationships. For example, a car is composed of numerous parts. If we try to model a car, we might end up with a *Car* class and many other classes related to parts, which can become fields of the *Car* class. For example, there can be a *Wheel* class to represent wheels. Since wheels are parts of a car, they can be combined with other classes that are parts of a car too. The whole combination, *Car* class, represents a real world car.

7.1 Extending Classes

A class can inherit all non-private members from another class by using the reserved word *extends*. When a class *A* extends class *B*, the type of class *A*, which is *A*,
becomes a sub-type of the type of class $B$, which is $B$. Figure 7.1 shows one super class $Vehicle$ and two sub classes—$Car$ and $Truck$.

```scala
class Vehicle(make: String, model: String, year: Int) {
  def calculatePrice(varFactor: Int): Double = {
    val price = ((year - 2000) * varFactor) * 1000
    price
  }
}

class Car(make: String, model: String, year: Int, mode: String) extends Vehicle(make, model, year) {
  // Car specific methods here
}

class Truck(make: String, model: String, year: Int, operation: String) extends Vehicle(make, model, year) {
  // Truck specific methods here
}
```

Fig. 7.1: Extending a Class

The class $Vehicle$ is a super class that has a common method called $calculatePrice$, which is inherited by sub classes $Car$ and $Truck$. Please do not worry about the correctness of the formula, because the focus, here, is on inheriting the class members. Also price is calculated differently by different manufacturers. Please note that the class $Vehicle$ has a constructor defined in the same line, with three parameters. Similarly, each of the sub classes has a constructor defined in the same line to that of the class declaration. The sub class $Car$ has an additional constructor argument called $mode$ and the sub class $Truck$ has an additional constructor parameter called $operation$.

The purpose of the constructor argument $mode$, in $Car$, is to represent whether a car is for personal use or for public transportation. Similarly, the constructor argument, $operation$, in $Truck$, is to model whether a truck is operated for shipping or for farming. Please note that three parameters are passed to the super class constructor while declaring sub classes. This is a feature that Scala provides, you don’t have to write a separate statement to pass parameters to the super class constructor. Please note that a class can directly inherit from only one class using the reserved word $extends$. When we need to model real world multiple inheritance scenarios, we can use traits, which will be discussed in Chapter 8.