Chapter 2
Elements of Object Orientation

2.1 Introduction

This chapter introduces the core concepts in Object Orientation. It concisely defines the terminology used and attempts to clarify issues associated with hierarchies. It also discusses some of the perceived strengths and weaknesses of the Object-Oriented approach. It then offers some guidance on the approach to take in learning about objects.

2.2 Terminology

Class A class defines a combination of data and procedures that operate on that data. Instances of other classes can only access that data or those procedures through specified interfaces. A class acts as a template when creating new instances. A class does not hold any data but it specifies the data that is held in the instance. The relationship between a class, its superclass and any subclasses is illustrated in Fig. 2.1.

Subclass A subclass is a class that inherits from another class. For example, in the last chapter, Student Employee is a subclass of Temporary Employee. Subclasses are, of course, classes in their own right. Any class can have any number of subclasses.

Superclass A superclass is the parent of a class. It is the class from which the current class inherits. For example, in the last chapter, Temporary Employee is the superclass of Student Employee. In Scala, a class can have only one superclass.

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_2

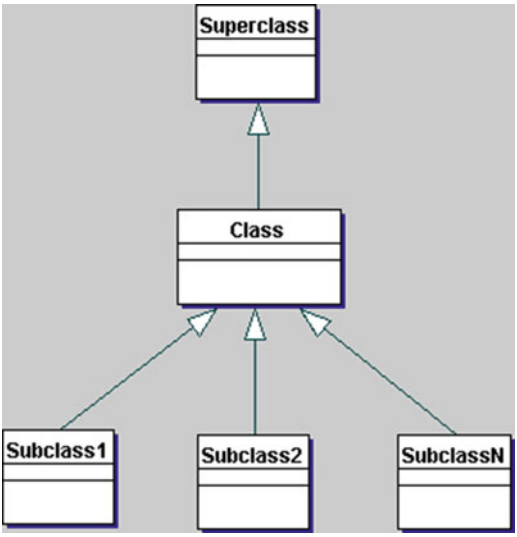
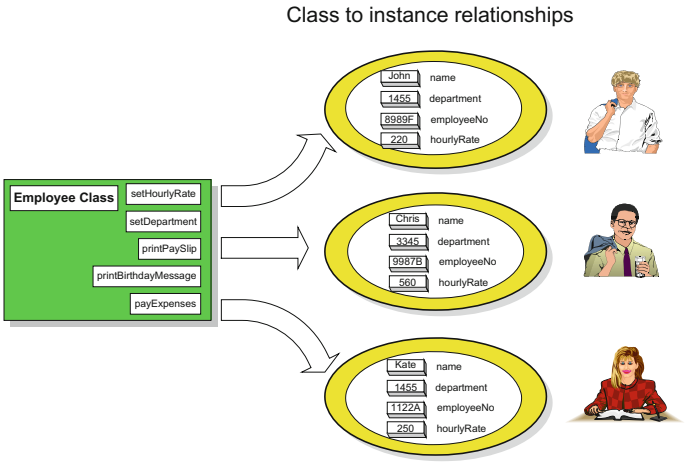


Fig. 2.1 Relationship between class, superclass and subclass

Instance or object An instance is an example of a class. All instances of a class possess the same data variables but contain their own data. Each instance of a class responds to the same set of requests.



Instance variable This is the special name given to the data which is held by an object. The "state" of an object at any particular moment relates to the current values held by its instance variables. (In Scala, there are also class-side variables, referred to as static variables, but these will be discussed later.) Figure 2.2 illustrates a definition for a class in pseudo-code. It includes some instance variable definitions: fuel, mileage and name.

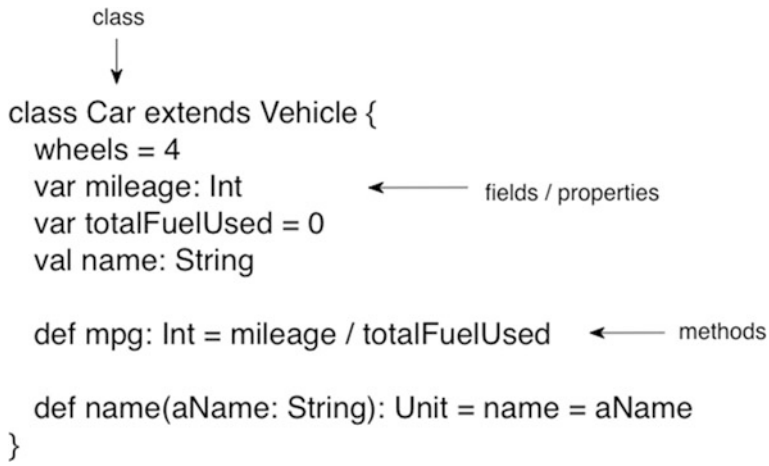


Fig. 2.2 A simple Scala class definition

Method A method is a procedure defined within an object. In early versions of Smalltalk, a method was used to get an object to do something or return something. It has since become more widely used; languages such as CLOS and Scala also use the term. Two methods are defined in Fig. 2.2: one calculates the miles per gallon, while the other sets the name of the car object.

Message One object sends a message to another object requesting some operation or data. The idea is that objects are polite, well-behaved entities which carry out functions by sending messages to each other. A message may be considered akin to a procedure call in other languages.

Single or multiple inheritance Single and multiple inheritance refer to the number of superclasses from which a class can inherit. Scala is a single inheritance system, in which a class can only inherit from one class. C++ is a multiple inheritance system in which a class can inherit from one or more classes.

2.3 Types of Hierarchy

In most Object-Oriented systems there are two types of hierarchy; one refers to inheritance (whether single or multiple) and the other refers to instantiation. The inheritance hierarchy (or *extends* hierarchy) has already been described. It is the way in which an object inherits features from a superclass.

The instantiation hierarchy relates to instances rather than classes and is important during the execution of the object. There are two types of instance hierarchy: one indicates a *part-of* relationship, while the other relates to a using relationship (It is referred to as an *is-a* relationship.).

The difference between an *is-a* relationship and a *part-of* relationship is often confusing for new programmers (and sometimes for those who are experienced in

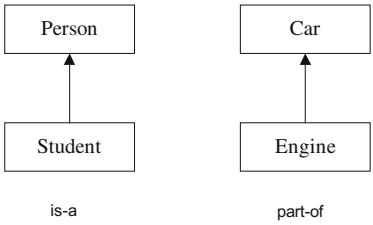


Fig. 2.3 *is-a* does not equal *part-of*

one language but are new to Object-Oriented programming languages, such as Scala). Figure 2.3 illustrates that a student *is-a* type of person, whereas an engine is *part-of* a car. It does not make sense to say that a student is *part-of* a person or that an engine *is-a* type of car!

In Scala, *extends* relationships are generally implemented by the subclassing mechanism. It is possible to build up large and complex class hierarchies which express these *extends* relationships. These classes express the concept of inheritance, allowing one class to inherit features from another. The total set of features is then used to create an instance of a class. In contrast, *part-of* relationships tend to be implemented using instance variables in Scala.

However, *is-a* relationships and classes are not exactly the same thing. For example, if you wish to construct a semantic network consisting of explicit *is-a* relationships between instances you will have to construct such a network manually. The aim of such a structure is to represent knowledge and the relationships between elements of that knowledge, and not to construct instances. The construction of such a network is outside the scope of the subclassing mechanism and would therefore be inappropriate.

If John is an instance of a class Person, it would be perfectly (semantically) correct to say that John *is-a* Person. However, here we are obviously talking about the relationship between an instance and a class rather than a subclass and its parent class.

A further confusion can occur for those encountering Scala after becoming familiar with a strongly typed language. These people might at first assume that a subclass and a subtype are essentially the same. However, they are not the same, although they are very similar. The problem with classes, types and *is-a* relationships is that on the surface they appear to capture the same sorts of concept. In Fig. 2.4, the diagrams all capture some aspect of the use of the phrase *is-a*. However, they are all intended to capture a different relationship.

The confusion is due to the fact that in modern English we tend to overuse the term *is-a*. We can distinguish between the different types of relationship by being more precise about our definitions in terms of a programming language, such as Scala. Table 2.1 defines the relationships illustrated in Fig. 2.4.

To illustrate this point, consider Fig. 2.5, which illustrates the differences between the first three categories.

The first diagram illustrates the potential relationships between a set of classes that define the behaviour of different categories of vehicle. The second diagram presents the subtype relationships between the categories. The third diagram

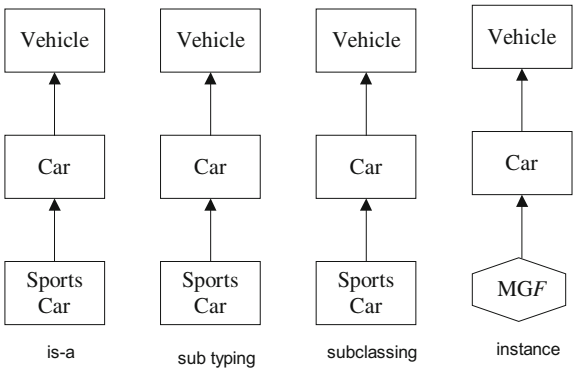


Fig. 2.4 Satisfying four relationships

Table 2.1 Types of *is-a* relationships

Specialisation	One thing is a special case of another
Type	One type can be used interchangeably with another type (substitutability relationship)
Subclassing or inheritance	An implementation mechanism for sharing code and representations
Instantiation	One thing is an example of a particular category (class) of things

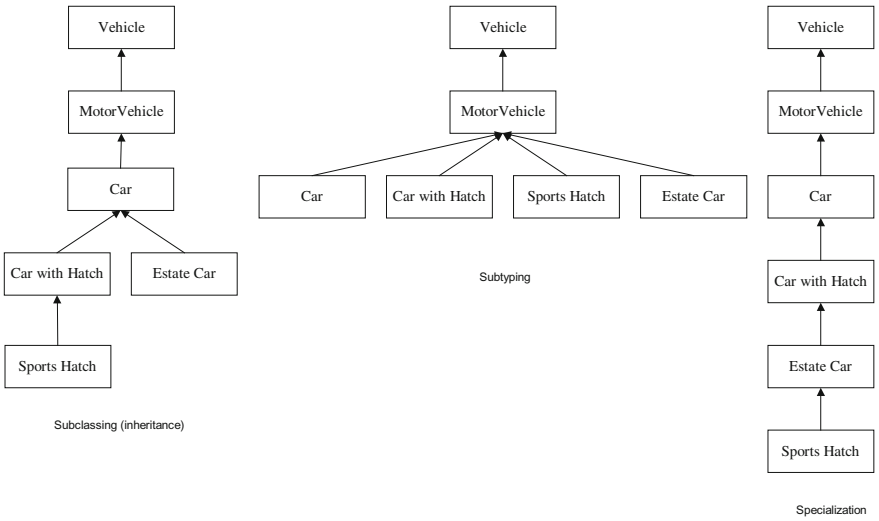


Fig. 2.5 Distinguishing between relationships

illustrates a straight specialisation set of relationships. Notice that although *estate car* is a specialisation of *car with hatch*, its implementation (the subclassing hierarchy) indicates that it does not share any of its implementation with the *car with hatch* class. It is worth noting that type relationships are specifications, while classes (and subclasses) are implementations of behaviour.

2.4 The Move to Object Technology

At present you are still acclimatising to Object Orientation. It is extremely important that from now on you do your utmost to immerse yourself in Object Orientation, object technology and Scala. This is because when you first encounter a new language or paradigm, it is all too easy to say that it is not good because you cannot do what you could in some other language or paradigm. We are all subject to the "better the devil you know than the devil you don't" syndrome. If you embrace Object Orientation, warts and all, at least for the present, you will gain most.

In addition, it is a fact of life that most of us tend to fit in learning something new around our existing schedules. This may mean, for example, that you are trying to read this book and do the exercises while still working in C, VisualBasic, Ada, etc. From personal experience, and from teaching others about Scala, I can say that you will gain most by putting aside a significant amount of time and concentrating on the subject matter involved. This is not only because Object Orientation is so different, but also because you need to get familiar not only with the concepts but also with Scala and its development environment.

So have a go, take a "leap of faith" and stick with it until the end. If, at the end, you still cannot see the point, then fair enough, but until then accept it.

2.5 Summary

In this chapter, we reviewed some of the terminology introduced in the previous chapter. We also considered the types of hierarchy which occur in Object-Oriented systems and which can at first be confusing. We then considered the pros and cons of Object-Oriented programming. You should now be ready to start to think in terms of objects. As has already been stated, this will at first seem a strange way to develop a software system, but in time it will become second nature. In the next chapter we examine how an Object-Oriented system might be developed and structured. This is done without reference to any source code as the intention is to familiarise you with objects rather than with Scala. It is all too easy to get through a book on Smalltalk, C++, Scala, etc., and understand the text but still have no idea how to start developing an Object-Oriented system.

2.6 Exercises

Research what other authors have said about single and multiple inheritance. Why do languages such as Smalltalk and Scala not include multiple inheritance?

Look for terms such as class, method, member, member function, instance variable and constructor in the books listed in the further reading section. When you have found them, read their explanation of these terms and write down your understanding of their meaning.

2.7 Further Reading

Suggested further reading for this chapter includes Coad and Yourdon (1991), Winston and Narasimhan (2001) and Meyer (1988). In addition all the books mentioned in the previous chapter are still relevant.