Chapter 3
Why *Object* Orientation?

3.1 Introduction

The previous chapter introduced the basic concepts behind Object Orientation, the terminology and explored some of the motivation. This chapter looks at how Object Orientation addresses some of the issues that have been raised with procedural languages. To do this it looks at how a small extract of a program might be written in a language such as C, considers the problems faced by the C developer and then looks at how the same functionality might be achieved in an Object-Oriented language such as Scala, Java or C#. Again do not worry too much about the syntax you will be presented with, it will be Scala but it should not detract from the legibility of the examples.

3.2 The Procedural Approach

As has already been stated, Object Orientation provides four things:

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism.

It has been claimed that these four elements combine to provide a very powerful programming paradigm, but why? What is so good about Object Orientation?

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_3

3.2.1 A Naked Data Structure

Consider the following example:

```
record Date {
        int day;
        int month;
        int year;
}
```

This defines a data structure for recording dates. There are similar structures in many procedural languages such as C, Pascal and Ada. It is naked because it has no defenses against procedures accessing and modifying its contents.

So what is wrong with a structure such as this? Nothing, apart from the issue of visibility? That is, what can see this structure and what can update the contents of the structure? For example, code could set the day to 1, the month to 13 and the year to 9999. As far as the structure is concerned the information it holds is fine (that is day = 01, month = 13, year = 9999). This is because the structure only knows it is supposed to hold an integer, it knows nothing about dates per se. This is not surprising, it is only data.

3.2.2 Procedures for the Data Structure

This data is associated with procedures that perform operations on it. These operations might be to test whether the date represents a date at a weekend or part of the working week. It may be to change the date (in which case the procedure may also check to see that the date is a valid one.

For example:

- `isDayOfWeek(date);`
- `inMonth(date, 2);`
- `nextDay(date);`
- `setDay(date, 9, 23, 1946);`

How do we know that these procedures are related to the date structure we have just looked at? By the naming conventions of the procedures and by the fact that one of the parameters is a data (record).

The problem is that these procedures are not limited in what they can do to the data (e.g. the setDay procedure might have been implemented by a Brit who assumes that the data order is day, month and year. However, it may be used by an American who assumes that date order is month, day, year. Thus the meaning of setDay(date, 9, 23, 1946) will be interpreted very differently. The American views this as the 23rd of September 1946, while the Brit views this as the 9th of the 23rd month, 1946. In either case, there is nothing to stop the date record being updated with both versions. Obviously the setDay() procedure might check the new date to see it was legal, but then again it might not. The problem is that the data is naked and has no defense against what these procedures do to it.

Indeed, it has no defense against what any procedures that can access it, may do to it.

3.2.3 Packages

One possibility is of course to use a package construct. In languages such as Ada packages are commonplace and are used as a way of organizing code and restricting visibility. For example,

```
package Dates is
     type Date is ....
     function isDayOfWeek(d: Date) return BOOLEAN;
     function inMonth(d: Date, m: INTEGER) return
                                                  BOOLEAN;
…
```

The package construct now provides some ring fencing of the data structure and a grouping of the data structure with the associated functions. In order to use this package a developer must import the package (e.g. using with and uses in Ada). They can then access the procedures and work with data of the specified type (in this case Date). There can even be data that is hidden from the user within a *private part*. This therefore increases the ability to encapsulate the data (hide the data) from unwelcome attention.

3.3 Does Object Orientation Do Better?

This is an important question "Does Object Orientation do any better" than the procedural approach described above? We will first consider packages, then inheritance …

3.3.1 Packages Versus Classes

It has been argued (to me at least) that a package is just like a class. It provides a template from which you can create executable code, it provides wall around your data with well-defined gateways, etc. However, there are a number of very significant differences between packages and classes.

Firstly, packages tend to be larger (at least conceptually) units than classes. For example, the `TextIO` package in Ada is essentially a library of textual IO facilities, rather than a single concept such as the class String in C#. Thus packages are not used to encapsulate a single small concept such as `Date,` but rather a whole set of related concepts (as indeed they are used in C# itself where they are called namespaces). Thus a class is a finer level of granularity than a package even though it provides similar levels of encapsulation.

Secondly, packages still provide a relatively loose association between the data and the procedures. A package may actually deal with very many data structures with a wide range of methods. The data and the methods are related primarily via the related set of concepts represented by the package. In contrast a class tends to closely relate data and methods in a single concept. Indeed, one of the guidelines presented later in this book relating to good class design, is that if a class represents more than one concept, split it into two classes.

Thus this close association between data and code and means that the resulting concept is more than just a data structure (it is closer to a concrete realization of an abstract data type). For example:

```
class Date {
    val day: Int = 1
    val month: Int 1
    val year: Int = 14
    def isDayOfWeek(): Boolean = {..}
}
```

Anyone using an instance of *Date* now gets an object which can tell you whether it is a day of the week or not and can hold the appropriate data. Note that the `isDayOfWeek()` method takes no parameters, it does not need to as it and the date is a part of the same thing. This means that a user of a Date object will never get their hands on the actual data holding the date (i.e. the integers day, month and year). Instead, they are forced to go via the internal methods. This may only seem a small step, but it is a significant one, nothing outside the object may access the data within the object. In contrast the data structure in the procedural version, is not only held separately to the procedures, the values for day, month or year could be modified directly without the need to use the defined procedures.

For example, compare the differences between an ADA-esque excerpt from a program to manipulate dates:

```
d: Date;
setDay(d, 28);
setMonth(d, 2);
setYear(d, 1998);
isDayOfWeek(d);
inMonth(d, 2);
```

Not that it was necessary to first create the data and then to set the fields in the data structure. Here we have been good and have used the interface procedures to do this. Once we had the data set-up we could then call methods such as `IsDayOfWeek` and `InMonth` on that data.

In contrast the Scala code uses a constructor to pass in the appropriate initialization information. How this is initialized internally is hidden from the user of the class Date. We then call method such as `isDayOfWeek()` and `isMonth(12)` directly on the object date.

```
val d = new Date(12, 2, 1998)
d.IsDayOfWeek()
d.InMonth(12)
```

The thing to think about here is where would code be defined?

3.3.2 Inheritance

Inheritance is the key element that makes an Object-Oriented language more than an object-based language. An object-based language possesses the concept of object, but not of inheritance. Indeed, inheritance is the thing that marks an Object-Oriented language as different from a procedural language. The key concept in inheritance is that one class can inherit data and methods from another, thus increasing the amount of code reuse occurring as well as simplifying the overall system. One of the most important features of inheritance (ironically) is that it allows the developer to get inside the encapsulation bubble in limited and controlled ways. This allows the subclass to take advantage of internal data structures and methods, without compromising the encapsulation a forded to objects. For example, let use define a subclass of the class `Date` (extends are used to indicate inheritance in Scala):

```
class Birthday extends Date {
    val name: String = ""
    val age: Int = 0
    def isBirthday(): Boolean = {..}
}
```

The method `isBirthday()` could check to see if the current date, matched the birthday represented by an instance of Birthday and return true if it does and false if it does not.

Note however, that the interesting thing here is that not only have I not had to define integers to represent the date, nor have I had to define methods to access such dates. These have both been inherited from the parent class `Date`.

In addition, I can now treat an instance of `Birthday` as either a Date or as a `Birthday` depending on what I want to do!

What would you do in languages such as C, Pascal or Ada? One possibility is that you could define a new package Birthday, but that package would not extend `Dates`, it would have to import Dates and add interfaces to it etc.? However, you certainly could not treat a Birthday package as a Dates package.

In languages such as Scala, because of *polymorphism*, you can do exactly that. You can reuse existing code that only knew about Date, for example:

- `def test(Date d): Unit = {..}`
- `t.test(birthday)`

This is because Birthday is indeed a type of Date as well as being a type of Birthday.

You can also use all of the features defined for Date on Birthdays:

- `birthday.isDayOfWeek()`

Indeed you do not actually know where the method is defined. This method could be defined in the class Birthday (in which it would override that defined in the class `Date`). However, it could be defined in the class `Date` (if no such method is defined in `Birthday`). However, without looking at the source code there is no way of knowing!

Of course you can also use the new methods defined in the class Birthday on instance (objects) of this class. For example:

- `birthday.isBirthday()`

3.4 Summary

Classes in an Object-Oriented language provide a number of features that are not present in procedural languages. Hopefully by the end of the book you will agree that they are useful additions to the developers' toolbox. If not, give it time, one of the problems that we all face (myself included) is a reluctance to change. To summarise, the main points to be noted from this chapter on Object Orientation are:

- Classes provide for inheritance.
- Inheritance provides for reuse.
- Inheritance provides for extension of data type.
- Inheritance allows for polymorphism.
- Inheritance unique feature of Object Orientation.
- Encapsulation represents a particularly good Software Engineering feature in Object Orientation.