Chapter 14
Objects and Instances

14.1 Introduction

This chapter will discuss the difference between objects in Scala and Instance of a class. This is important as many other Object-Oriented languages use these terms interchangeably. However, in Scala they are significantly different concepts, defined with different language constructs and used in different ways.

14.2 Singleton Objects

Scala provides another type that can sit alongside the class types. It is directly supported by the language construct *object*. A Scala object is a singleton object that is accessible to any Scala code that has visibility of that object definition. The term singleton here refers to the fact that there is a single instance of the object definition within the virtual machine executing the Scala program. This is guaranteed by the language itself and does not require any additional programmer intervention.

If you have never come across the concept of a Singleton object before it may appear an odd idea. However, it is very widely and commonly used within the Object-Oriented programming world. Examples of the singleton concept can be found in Java, C#, Smalltalk, C++, etc. and have been documented in various ways since it was first popularised in the so-called Gang of Four patterns book. The four authors of this book, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (collectively known as the "Gang of Four", or GoF for short) popularised the patterns concepts and ideals.

So what are Design Patterns? They are essentially useful recurring solutions to problems within designs. For example, "I want to loosely couple a set of objects, how can I do this?", might be a question facing a designer. The mediator Design Pattern is one solution to this. If you are familiar with Design Patterns you can use

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_14

them to solve problems that occur. Typically early in the design process, the problems are more architectural/structural in nature, while later in the design process they may be more behavioural. Design Patterns actually provide different types of patterns some of which are at the architectural/structural level and some of which are more behavioural. They can thus help every stage of the design process.

The Singleton pattern describes a type that can only have one instance constructed for it. That is, unlike other types it should not be possible to obtain more than one instance within the same virtual machine. Thus the Singleton pattern ensures that only one instance of a type is created. All elements that use an instance of that type; use the same instance.

The motivation behind this pattern is that some classes, typically those classes that involve the central management of a resource, should have exactly one instance. For example, a object that managements the reuse of database connections (i.e. a connection pool) could be a singleton.

However, implementing a singleton in some language can be more complex than initially thought, as it is necessary to ensure that it is not possible to have multiple instances of the singleton concept. Scala solves this problem by making the singleton concept part of the language.

You have already seen examples of the syntax used for singletons as that is the syntax we have used for each of our application entry points (whether with an explicit `main` method or by using the `App` trait). The syntax is:

```
object ObjectName { body }
```

Objects can

- extend classes,
- mix in traits,
- define methods and functions
- as well as properties (both vals and vars).
- But they cannot define constructors.

A simple singleton object is shown below:

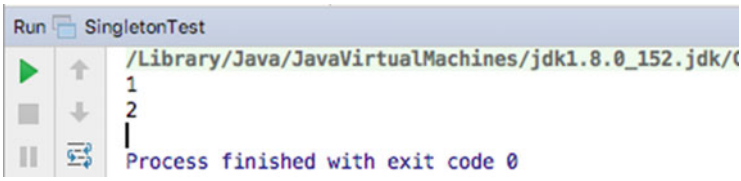
```
object ConnectionPool {
  var count = 0;
  def next = {count = count + 1}
}
```

This simplified *mock* connection pool object defines a count of the number of connections being managed and a method next that adds to the count. This object does not need to be instantiated to be used instead it can be referenced directly by any client code:

```
object SingletonTest extends App {
  ConnectionPool.next
  println(ConnectionPool.count)
  ConnectionPool.next
  println(ConnectionPool.count)
}
```

In the above client code we are accessing the functionality in next and the data in count directly by reference to the name of the object (we did not create a new object using the keyword new nor did we run a constructor).

The result of running this simple application is shown in below figure, indicating that the same object is used throughout as the count is incremented from 1 to 2:



14.3 Companion Objects

Companion objects are singleton objects' for a class—they can be used to provide utility functions such as factory methods that will support the concept being modelled by the class. As a companion object is an object, it is a singleton instance that sits alongside the class.

To define a companion object it must:

- Have the same name as the Companion Class.
- Must be defined in the same source file as the Class.

When used in this way companion objects are useful placeholders for static style behaviour (as found in languages such as Java or C#).

From the point of view of the user of the class; the companion object and the class appear to be a single concept. For example, consider the following definition of a `Session` class and a companion object.

```
/**
  * The Companion class
  */
class Session(var id: Int) {
  override def toString = "Session[" + id + "]"
}

/**
  * Its Companion (singleton) object
  */
object Session {
  private var counter = 0
  private def next = counter = counter + 1
  def create = { next; new Session(counter) }
}
```

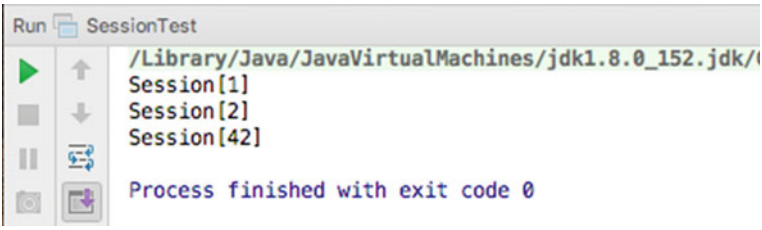
Notice that the object `Session` has a private `counter` (initialised to Zero) and a private method `next`. By default all methods and properties are public (accessible anywhere); here we are making the property counter and the method next visible only within the Object Session.

The utility method `create` then uses the method `next` to increment the counter before it creates a new `Session` instance. Note that we are using a ';' here to separate the two statements as they are on the same line and Scala would infer that they were related. I am also surrounding them with curly brackets '{..}' as this is a multiple statement method and thus need to group them.

From a client of the Session concepts point of view they can now create a new session using the `create` (factory) method or by using the keyword *new* and the class name directly:

```
object SessionTest extends App {
  val s1 = Session.create
  println(s1)
  val s2 = Session.create
  println(s2)
  val s3 = new Session(42)
  println(s3)
}
```

The first two session instances above are created using the Session objects Create method and the third session instance is created using the new keyword. From the client programmers point of view there is a single concept here Session which can be instantiated into two ways. The advantage of the Create method is that it handles ensuring an increment of the session ID whereas the use of new allows any Session id to be used. The out of this program is shown below.



14.3.1 Companion Object Behaviour

It may at first seem unclear what should normally go in a method defined in the class as opposed to what should go in a method defined in a companion object when defining a new class. After all, they are both defined in the same file and relate to the same concept. However, it is important to remember that one defines the behaviour of which will be part of an instance and the other the behaviour of which can be shared across the whole concept being implemented.

In order to maintain clarity companion object methods, should only perform one of the following roles:

- *Application Entry Point* This role is very important as it is how you can use a companion object as the root of an application. It is common to see main methods (or the App trait) which do nothing other than create a new instance the main class of an application and fire off the appropriate behaviour. For example,

```
object Editor extends App {
  val editor = new EditorView()
  editor.startDisplay
}
```

  If you define such a method, but the class is not the root of the application, it is ignored. This makes it a very useful way of providing a *test* harness for a given class.
- *Answering enquiries about the class* This role can provide generally useful objects, frequently derived from companion object variables. For example, they may return the number of instances of this class that have been created using a factory method.
- *Instance management* In this role, companion object methods control the number of instances created. For example, a class only allows a ten instance to be created. Instance management methods may also be used to access an instance (e.g. randomly or in a given state).

- *Examples* Occasionally, companion object methods are used to provide helpful examples which explain the operation of a class.
- *Testing* companion object methods can be used to support the testing of an instance of a class. You can use them to create an instance, perform an operation and compare the result with a known value. If the values are different, the method can report an error. However, test frameworks are generally a better approach.
- *Support for one of the above roles.*

Any other tasks should be performed by a method (of function) defined in the class.

14.3.2 A Object or an Instance

In some situations, you may only need to create a single instance of a class and reference it wherever it is required. A continuing debate ponders whether it is worth creating such an instance or whether it is better to define the required behaviour in a companion object. The answer to this is not straightforward as there are several factors that should be taken into account including:

- The use of an object in Scala guarantees you a singleton instance within the current JVM. This means that it also limits you to a single instance in the current JVM and over time this may be a problem.
- The creation of an instance has a very low overhead. This is a key feature in Scala and it has received extensive attention.
- You may be tempted to treat the object as a global reference. This suggests that the implementation has been poorly thought out.

In deciding whether to use a Scala object or a Scala class to hold data and/or behaviour or functionality you need to consider the context in which it will be used, how you expect to develop the concept and whether there will ever be a need for more than one instance of that concept.

References

Gang of Four Design Patterns Book

- Gamma E, Helm R, Johnson R, Vlissades J (1995) Design Patterns: elements of reusable Object-Oriented software, Addison-Wesley

Quick reference list of Design Patterns

- http://www.oodesign.com/

  Introductory descriptions and examples of patterns

- http://sourcemaking.com/design_patterns

  List of patterns based sites

- http://hillside.net/patterns/patterns-catalog