Chapter 9
Scala Classes

9.1 Introduction

This chapter considers the constructs in Scala used to define classes.

9.2 Classes

A class is one of the basic building blocks of Scala. Classes act as *templates* which are used to construct instances. Classes allow programmers to specify the *structure* of an instance (i.e. its instance variables or fields) and the behaviour of an instance (i.e. its methods and functions) separately from the instance itself. This is important, as it would be extremely time-consuming (as well as inefficient) for programmers to define each instance individually. Instead, they define classes and create *instances* of those classes.

9.2.1 Class Definitions

In Scala, a class definition has the following format:

```
class nameOfClass extends SuperClass / Trait {
    scope properties;
     scope methods
     scope functions
}
```

Although you should note that you can mix the order of the definition of properties, functions and methods as required within a single class.

You need not remember this format precisely, as the meaning of the various parts of the class definition is explained later in the book. Indeed, the above is far from complete, but it illustrates the basic features. The following code is an example of a class definition:

```
class Person {
  var age = 0
  var name = ""
}
```

This code defines a class called Person. This class possesses two *properties* (or instance variables) called `name` and `age`. It has no functions nor methods defined.

Notice that the `age` instance variable contains a value of type *Int* (this is a basic data type), while the instance variable `name` possesses an instance of the class `String`. Here Scala is inferring the type of the properties based on the initial assignments made to the `age` and `name`—they are statically typed but that type is determined by the compiler at compile time (not at runtime). Both variables are initialised: `age` to Zero and `name` to the empty string "".

Classes are not just used as templates. They have three further responsibilities: holding methods, providing facilities for inheritance and creating instances.

9.2.2 Developing a Class Definition

Let us return to the class `Person` and explore the definition of this class a bit further. A slightly different version of the definition for this class is presented again below:

```
class Person() {
  var name = ""
  var age = 0
}
```

What does this class actually say? It defines a number of things:

It states that both name and age are read/write properties. This can be determined by the fact that they are **vars** and not **vals**. A *var* is a property (or local variable) that can be written to multiple times. A *val* is a property (or local variable) that can only be set once. In both cases they can be read as many times as required. Thus, in this case both name and age can be read and reset as required by the application. If we had made name a *val*, then it would only be possible to write to it once, after a value has been set it cannot be reset.

This definition also defines what is known as a Zero parameter constructor. These are the '()' after the name of the class. Every class in Scala will have at least one constructor. A constructor is actually used to initialise values within the object created from the class. In this case that constructor does not take any parameters and merely provides a default placeholder. In Scala such definitions are typically optional and this is the case here.

In either case when we create a new instance of the `Person` class, we cannot provide the appropriate name and age until later. Thus a test program for this class might look like:

```
object PersonTest1 extends App {
  val p1 = new Person()
  println(s" ${p1.name} is ${p1.age}")
  p1.name = "John"
  p1.age = 21
  println(s" ${p1.name} is ${p1.age}")
}
```

In this case line 2 causes a new instance of the class `Person` to be created. Note that it is the keyword *new* that is being used here to create a new instance of the class and the '()' which are used to indicate the constructor to execute when that new instance is created. The result of this creation is that the address of this new object is stored in the variable `p1` which is a variable that can hold references to (i.e. the address of) an instance of type `Person`. This could have been written in longhand form as:

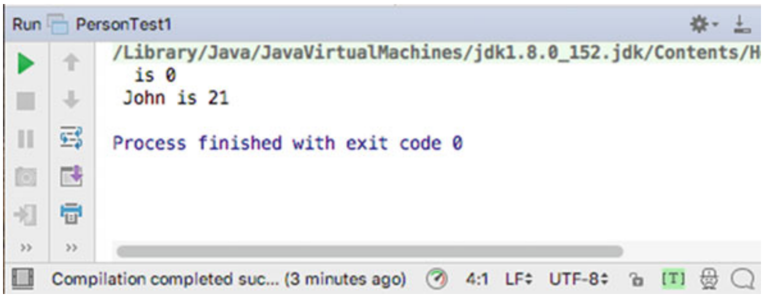
```
val p1: Person = new Person()
```

Here we explicitly specify the type of the variable p1. In the earlier example (in PersonTest1) Scala inferred the type of `p1` for us.

Line 3 then accesses the current values of name and age for the instance referenced by p1 and prints them to the console (standard output) of your IDE. It uses string interpolation to embed the current value of the name and age fields into a string. The interpolation is indicated by the 's' preceding the string itself. Scala will then look inside the string for values preceded by a '$' such as $p1 or for expression surrounded by a ${..} such as ${p1.age}. It will then evaluate the variable or expression and inline the result into the string being processed.

In lines 4 and 5 we actually assign the values and we want to name and age properties of p1. After line 8 p1 now represents John who is 49. Line 9 the reprints this data out.

We can see the effect of running this program on the Console in the IDE, for example,



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/H
  is 0
 John is 21

Process finished with exit code 0
```

As can be seen from this, the effect of the first print out is that the empty string and Zero are first printed—which seems a little confusing. The second print out then shows ‘John’ and 21, which appears to make more sense.

9.2.3 Classes and Messages

When a message is sent to an instance of a class, it is not the instance which possesses the method but the class. This is for efficiency reasons: if each object possessed a copy of all the methods defined for the class, then there would be a great deal of duplication. Instead, only the class possesses the method definitions. Thus, when an object receives a message, it searches its class for a method with the name in the message. If its own class does not possess a method with the appropriate name, it goes to the superclass and searches again. This search process continues up the class hierarchy until either an appropriate method is found or the class hierarchy terminates (with the class `Any`). If the hierarchy terminates, an error is raised.

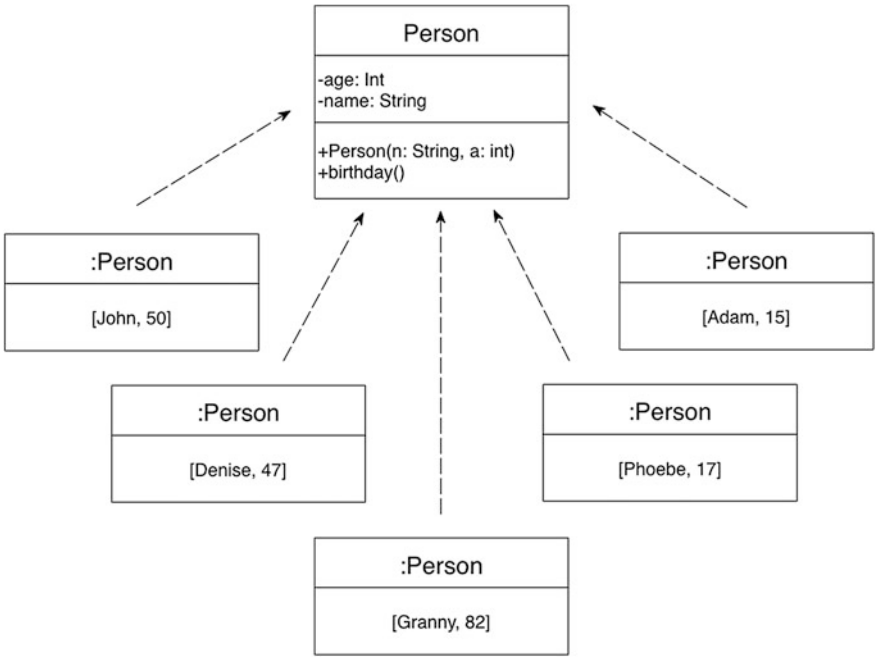
If an appropriate method is found, then it executes *within the context of the object,* although the definition of the method resides in the class. Thus, different objects can execute the same method at the same time without conflict.

Do not confuse methods with the data held by a property. Each instance possesses its own copy of the data (as each instance possesses its own state). Following Figure illustrates this idea more clearly.

9.2.4 Instances and Instance Variables

In Scala, an *instance* is an example of a *class*. All instances of a class share the same responses to messages (methods or functions), but they contain different data (i.e. they possess a different “state”). For example, the instances of class `Point` all respond in the same way to messages inquiring about the value of the x-coordinate, but they may provide different values.

The class definition consists of variable declarations and method definitions. The state of each instance is maintained in one or more instance variables/properties (also known as fields).



The above figure contains five instances of the class Person. Each instance contains copies of the instance variable definitions for name and age, thus enabling them to have their own values for these instance variables. In contrast, each instance references the single definition for the method birthday, which is held by the class.

9.2.5 Classes and Inheritance

It is through classes that an object inherits facilities from other types of object. That is, a subclass inherits properties from its superclass. For example, the Person definition above is a subclass of AnyRef which in turn is a subclass of Any. Therefore, Person inherits all the methods and instance variables that were defined in AnyRef and Any (except those that were overwritten in Person).

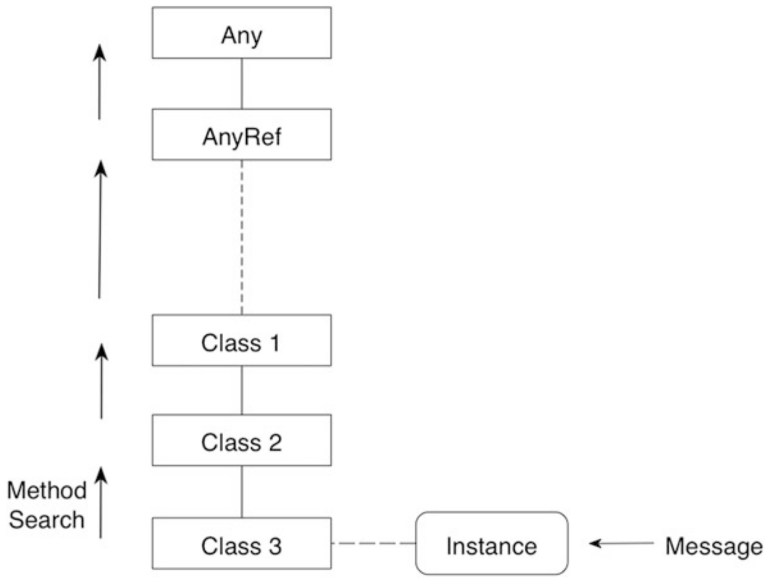
Subclasses are used to refine the behaviour and data structures of a superclass. It should be noted that Scala supports single inheritance (as does Java and C#) while some Object-Oriented languages (most notably C++) support multiple inheritance.

Multiple inheritance is where a subclass can inherit from more than one superclass. However, difficulties can arise when attempting to determine where methods are executed. Scala introduces the concept of *traits* to overcome one of the most significant problems with single inheritance. However, the discussion of Scala traits comes later in this book.

9.2.5.1 An Example of Inheritance

To illustrate how single inheritance works, consider the following Figure. There are several classes: `Class1` is a subclass of `AnyRef`, `Class2` is a subclass of `Class1` and `Class3` is a subclass of `Class2`. Note that `AnyRef` is a direct subclass of `Any`. In Scala `Any` is the root of all types.

When an instance of `Class3` is created, it contains all the instance variables defined in Classes 1–3 and class `AnyRef`. If any instance variable has the same name as an instance variable in a higher class, then the `Class3` instance uses the instance variable definition from the nearest class. That is, `Class3` definitions take priority over `Class2,` and `Class2` definitions take priority over `Class1`.

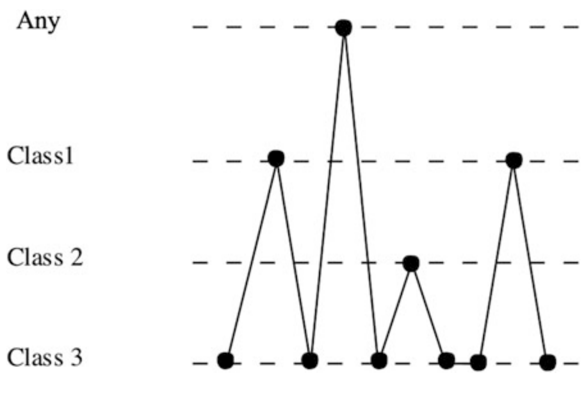


We can send an instance of `Class3` a message requesting that a particular method is executed. Remember that methods are held by classes and not by instances. This means that the system first finds the class of the instance (in this case `Class3`) and searches it for the required method. If the method is found, then the search stops and the method is executed. However, if the method is not found, then the system searches the superclass for `Class3`, in this case `Class2`. This process is repeated until the method is found. Eventually, the search through the super classes may reach the class `Any` (which is the root class in the Scala system). If the required method is not found here, then the search process terminates and the `doesNotUnderstand`: method in the class `Any` is executed instead. This method raises an exception stating that the message sent to the original instance is not understood.

This search process is repeated every time a message is sent to the instance of `Class3`. Thus, if the method that matches the original message sends a message to itself (i.e. the instance of `Class3`), then the search for that method starts again in `Class3` (even if it was found in `Class1`).

9.2.5.2 The Yo-Yo Problem

The process described above can pose a problem for a programmer trying to follow the execution of the system by tracing methods and method execution. This problem is known as the Yo-Yo problem (see the figure below) because, every time you encounter a message that is sent to “this” (the current object), you must start searching from your own class. This may result in jumping up and down the class hierarchy.



The problem occurs because you know that the execution search starts in the current instance’s class, even if the method which sends the message is defined in a superclass of the current class. In the above Figure, the programmer starts the search in `Class3` but finds the method definition in `Class1`; however, this method sends a message to “this” which means that the programmer must restart the search in `Class3`. This time, the method definition is found in the class `Any`, etc. Even with the browsing tools provided, this can still be a tedious and confusing process (particularly for those new to Scala).

9.2.6 Instance Creation

A class creates an instance in response to a request, which is handled by a constructor. The request is represented by the keyword *new* followed by the name of class. The constructor to be invoked once the class is created is indicated by the parameters that follow the name of the class in parentheses (with the empty parameter list often being referred to as the no parameter constructor).

A programmer requests a new instance of a Scala class using the following construct:

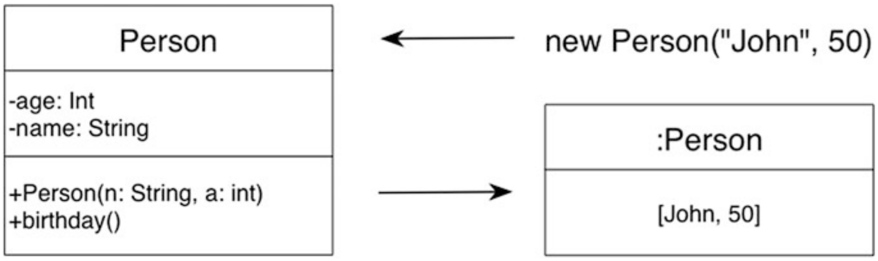
```
new ClassName()
```

Any parameters which need to be passed into the constructor can be placed between the parentheses. They are then passed onto an appropriate constructor. Constructors are special as every class has a primary constructor and can optionally have one or more auxiliary constructors (which must eventually invoke the primary constructor). As previously mentioned, constructors are used to initialise a new instance of the class in an appropriate manner (you do not need to know the details of the process).

The whole of this process is referred to as *instantiation*. An example of instantiating the class `Person` is presented below:

```
new Person("John", 50)
```

The class `Person` receives the message `new` which causes the Scala to generate a new instance of the class, with its own copy of the instance variables `age` and `name` (see the following Figure). The name property of the instance is “John” and the age property is set to 50.



9.2.7 Constructors

A constructor is not a method, but a special operation that is executed when a new instance of a class is created. Depending on the arguments passed to the class when the instance is generated, a different constructor can be called. For example, a class may need to have three fields initialised when it is instantiated. However, the programmer may allow the user of the class to provide one, two or three values. They can do this by defining auxiliary constructors that take one or two in addition to the primary three parameter constructors.

The syntax for a constructor is:

```
class classname(.. primary constructor parameters ..) {
  auxiliary constructors ( … parameters …) {
   … statements …
  }
}
```

By default every class has a single primary constructor. It is up to the programmer to decide if this is a no parameter constructor or one that takes a set of parameters. It is defined following the class name in the class definition. An example of a primary constructor for the class Person is shown below:

```
class Person(var name: String = "Anyone",
             var age: Int = 0)
```

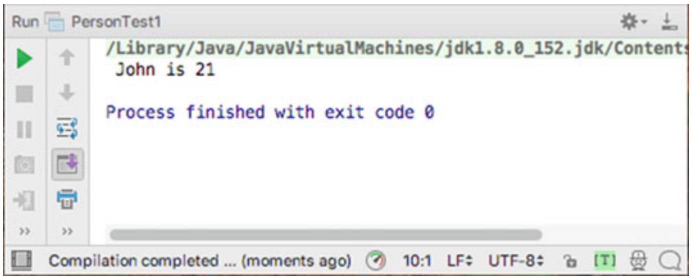
The above is complete as a class definition. It defines a class *Person* with a primary constructor that takes two parameters and defines no additional methods (beyond the defaults provided). As such no braces are needed unless you want to add any code, fields or methods to the body of the class.

The primary constructor defines two properties for name and age and also provides default values for these properties. Note that both properties are *vars,* and therefore both readers and writers are generated for them by Scala.

The main advantage of this version of the class `Person` over those defined earlier is that the `name` and `age` can be provided at the same time as the class is instantiated, rather than having to create the instance and then set the name and age separately. Thus we can now write:

```
object PersonTest1 extends App {
  val p1 = new Person("John", 21)
  println(s" ${p1.name} is ${p1.age}")
}
```

Now a `Person` type object can be created with a `name` and an `age` which seems more meaningful and understandable. The output generated from this application is illustrated below:


Run PersonTest1

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Content
 John is 21

Process finished with exit code 0
```

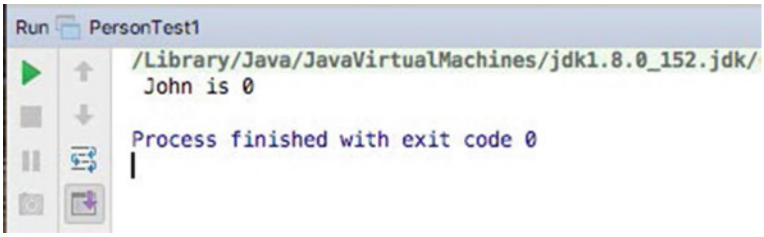
Compilation completed ... (moments ago) 10:1 LF: UTF-8:

Rather than having a null string and 0 printed out we have "John" and 21 right from the start.

However, because we have provided default values for both the name and the age, they are in fact optional. If we omit the age and only provide the name as in:

```
object PersonTest1 extends App {
  val p1 = new Person("John")
  println(s" ${p1.name} is ${p1.age}")
}
```

Then the result is that we set the age to Zero, and thus if we run this example, we would find that the output is:


Run PersonTest1

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/
 John is 0

Process finished with exit code 0
```

Similarly, we can omit the name and the age:

```
val p3 = new Person()
```

Which would result in the name being set to the string "Anyone" and the age to 0.

Note that we cannot omit the name and just provide the age as Scala would try and bind an integer (e.g. 18) to the name field which is invalid. However, as we are not passing in any parameters we can omit the brackets, e.g.:

```
val p0 = new Person
```

9.2.8 Auxiliary Constructors

Every class in Scala has a primary constructor; however, optionally any class in Scala can also have one or more *auxiliary constructors*.

If we wanted to allow a Person to be instantiated with just an age, then one way to do it would be to define an auxiliary constructor. An auxiliary constructor must be called *this* and must call another constructor. It can either call the primary constructor or another auxiliary constructor defined within the same class (you can have any number of auxiliary constructors) as their *first* action. They *cannot* simply call the superclass's constructor explicitly or implicitly as they can in Java. This ensures that the primary constructor is the sole point of entry to the class.

The following example defines an auxiliary constructor for the Person class that takes an integer to use for a Person's age without the need to define the person's name:

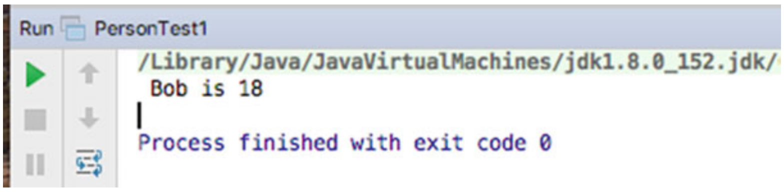
```
class Person (var name: String="", var age:Int=0) {
  def this(a: Int) = this("Bob", a)
}
```

Note that the only thing this auxiliary constructor does is to call the primary constructor providing a default name for all *unnamed* person. This is a very common idiom for auxiliary constructors to use.

We can now use this constructor to construct a new instance of the Person class using only an age (but who will be default be called Bob):

```
object PersonTest1 extends App {
  val p1 = new Person(18)
  println(s" ${p1.name} is ${p1.age}")
}
```

The result of executing this program is:



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/
 Bob is 18

Process finished with exit code 0
```

As you can see the name has been set to Bob (by the auxiliary constructor) and the age to 18.

To summarise auxiliary constructors:

- Any class can have any number of additional, auxiliary constructors
- The first statement within an auxiliary constructor must be a call to another auxiliary constructor, or to the primary constructor
- Thus, every object creation eventually ends up at the primary constructor

9.2.9 Class Initialisation Behaviour

It may seem somewhat strange, but you can place free-standing code anywhere within the body of the class. Here free-standing code means executable statements that are not part of a method, function or constructor but are defined within the scope of the class as a whole. Such free-standing code is executed after a new instance has been created, and after any values have been assigned to any properties defined within the primary constructor. Such code can be treated as part of the initialisation process and may be treated as the way in which you can define your own initialisation behaviour. Free-standing code is typically used for validation checks, additional processing and auditing functions, for example,

```
class Currency(a: Long, d: String) {

  // Auxiliary Constructor
  def this(a: Long) = this(a, "GBP")

  // Validates the data being received
  // If not met will throw a
  // java.lang.IllegalArgumentException
  require(a > 0)
  // Runs when the object is first created
  // Can't use toString as not set yet
  println("Created: " + d + " " + a)

}
```

In the above example, we have a class `Currency,` which possesses:

- A primary constructor with two parameters.
- An auxiliary constructor with one parameter and which defaults the type of the currency to "GBP".
- A validation statement that checks that the parameter 'a' is greater than Zero. If it is not then an illegal argument exception is thrown (a type of error).
- A logging (`println`) statement that indicates what has been created.
- The require statement and the `println` statements are free-standing statements and are therefore part of the initialisation routine of the class.

An important point to note is that the initialisation behaviour (the free-standing code) runs before the code in the auxiliary constructor. This is because it is associated with the instantiation of the class and allows behaviour to be defined for the primary construction process. The auxiliary constructor can then override this if it needs to. This is achieved by requiring the auxiliary constructor to always call another constructor (e.g. the primary constructor) as the first thing that it does.

9.2.10 Review of Classes and Constructors

The syntax of a class is

```
class ClassName(constructor parameters) {body}
```

The ClassName should begin with a capital letter and be in Camel Case to follow Scala conventions. The constructor is indicated by the parenthesis '()' following the class name, and there must always be a constructor defined (no hidden constructors as in Java) even though it may take no parameters. Within the constructor there can be specific meanings for the parameters:

- A **var** parameter will cause a field, getter and setter to be included.
- Setter and getter methods can be redefined inside the method.
- A **val** parameter will create a field and a getter, but no setter.
- A parameter with neither **val** nor **var** does not create a field or any methods, but it can be used within the body of the class—it is a local field to the class.
- However, note that if a case class is used, then parameters to the primary constructor default to **val** (see below).

When a new instance of a class is defined, the fields are created, the methods are defined and any "loose" code (not within a `def`) is executed.

To make some of the terminology clearer here are some definitions:

- *Class* acts as a template for defining the structure and behaviour of a type of thing.
- *Instance* is an example of a class that maintains its own state (copy of the data held within it).

- *Instance variables* are defined in the class, but a copy is maintained in each instance, which has its own value.
- *Instance methods* are defined in the class, with a single copy maintained in the class, but they are executed within the context of an object.
- *Constructors* are used to initialise properties once the instance has been constructed in memory, but before any other code has access to the instance.
- *Auxiliary constructors* can be used to extend the functionality of the base constructor but must either directly or indirectly invoke the base constructor.

9.3 Case Classes

There is another construct that can be used when defining a class, this is a `case` class. A `case` class is defined in the same way as a normal class, with an additional keyword placed in front of the class keyword. This keyword is `case`. For example,

```
case class Person (name: String="Denise", var age:Int=0)
```

The effect of this is that a number of additional features are provided for your class. The first obvious difference is that it appears that you no longer have to use the keyword *new* to create a new instance of a class; instead you can apparently just use the name of the class, for example,

```
object PersonTest1 extends App {
  val p1 = new Person("John", 21)
  println(s" ${p1.name} is ${p1.age}")
}
```

Actually a factory facility has been created that is named after the class and hides the use of the *new* keyword (although logically it is still *new* that is being used to construct the instance). A factory is a recurring software pattern that is used to produce instances of things (in the same way that a physical car factory produces cars).

Actually, the use of the `case` keyword provides a number of enhancements to the basic class definition including:

1. A default factory creation facility—no need to use *new*.
2. All arguments to the constructor are `val` by default, no need to state that they are *vals* (although you can override with a *var*).
3. Default implementation of `toString` method. This method is used to convert the object into a printable string format. The normal default provides the name of the class from which the instance was created and an indication of where it resides in memory.

4. Default value-based implementation of the `equals` method (used by ==). That is, equality is based on the values held in the parameters to the primary constructor.
5. Default *copy* method to create a `copy` of an object.
6. Default implementation of the `hashcode` method. This is an unique code used to represent an unique instance in memory and which is suitable for use in a hash map data structure (which relies on a key to value mapping).

The use of a case class also allows some additional comparison tests, which we will look at when we consider pattern matching in Scala.

Case classes can be used to represent data-oriented classes (although Value Types are a better option). They are also very useful when writing more function-oriented code as they avoid a lot of extra syntaxes.

Note that in older versions of Scala (pre-Scala 2–11) the number of parameters to a case class constructor was restricted to 22. Since Scala 2–11, this limitation has been removed; although if you find yourself with such large numbers of parameters, then consideration should probably be given to your design.

9.3.1 A Sample Class

Let us bring together the concepts that we have looked at so far in another version of the class `Person`. A new version of the class `Person` is shown below:

```scala
class Person(val name:String, var age: Int) {

  def birthday(): Unit = {
    print("Happy Birthday, you were " + age)
    age = age + 1
    println("today you are now " + age)
  }

  def isPensioner(): Boolean = age > 65

  def isToddler = age > 0 && age < 3

  override def toString() =
    "Person[" + name + ", " + age + "]"
}
```

This class exhibits several features we have seen already and expands a few others:

- The class has two parameter constructors that take a String and an Int.
- It defines two properties a read-only name and a read/write age (i.e. a val and a var) as part of the constructor definition.
- It redefines the `toString` method so that the details of the person object can be used in the string representation of an instance of the class.
- It defines three methods `birthday`, `isPensioner` and `isToddler`.
- The method `isTodder` does not include '()' and thus can only be invoked without brackets.
- The method `birthday()` returns `Unit` (i.e. it does not return a value) and is comprised of three statements, two print statements and an assignment, and the method body must therefore be placed in curly brackets `{…}`.
- `isPensioner` and `isToddler` represent shorthand forms written in a single line.
- `isPensioner` returns a `Boolean` value (i.e. one that returns true or false).
- Scala infers the value returned by `isToddler` which will also be a `boolean`.

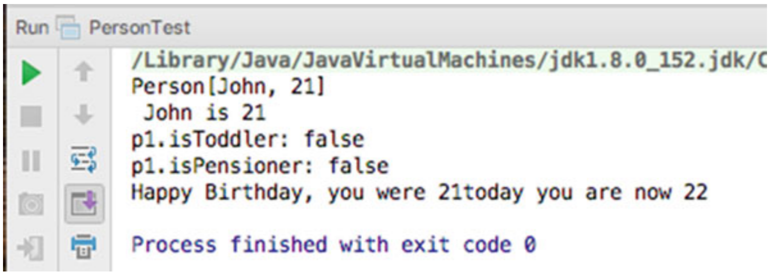
It also illustrates a few other ideas:

- The test '>' is a Boolean operator which returns a true of false value depending upon the left- and right-hand values
- The && represents another Boolean operation, this time the 'and' operation that will return true if and only if both the right-hand expression (age > 0) and the left-hand expression (age < 3) are true.

These Boolean operators will be explored in more detail later in the book.
An example application using this class is given below:

```scala
object PersonTest extends App {
  val p1 = new Person("John", 21)
  println(p1)
  println(s" ${p1.name} is ${p1.age}")
  println(s"p1.isToddler: ${p1.isToddler}")
  println(s"p1.isPensioner: ${p1.isPensioner()}")
  p1.birthday()
  p1.age = 18
  // p1.name = "Bob"
}
```

This application creates an instance of the `Person` class using the values "John" and 21. It then prints out `p1` using `println` (which will automatically call the `toString()` method on the instances passed to it). It then accesses the values of name and age properties and prints these. Following this it calls the `isToddler` and `isPensioner` methods and prints the results returned. It then calls birthday. Finally it assigns a new value to the `age` property (this is allowed as it is a var). However it is not possible to reassign a value to the `name` field as it is defined as a val. The output from this application is given below:



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/C
Person[John, 21]
 John is 21
p1.isToddler: false
p1.isPensioner: false
Happy Birthday, you were 21today you are now 22

Process finished with exit code 0
```