Manual
Scala Training
Xomnia

Assertions

ScalaTest makes three assertions available by default in any style trait. You can use:

- `assert` for general assertions;
- `assertResult` to differentiate expected from actual values;
- `intercept` to ensure a bit of code throws an expected exception.

In any Scala program, you can write assertions by invoking `assert` and passing in a `Boolean` expression:

```
val left = 2

val right = 1

assert(left == right)
```

If the passed expression is `true`, `assert` will return normally. If `false`, Scala's `assert` will complete abruptly with an `AssertionError`. This behavior is provided by the `assert`method defined in object `Predef`, whose members are implicitly imported into every Scala source file.

ScalaTest provides a domain specific language (DSL) for expressing assertions in tests using the word `should`. ScalaTest matchers provides five different ways to check equality, each designed to address a different need. They are:

```
result should equal(3) // can customize equality

result should ===(3) // can customize equality and enforce type constraints

result should be(3) // cannot customize equality, so fastest to compile

result shouldEqual 3 // can customize equality, no parentheses required

result shouldBe 3       // cannot customize equality, so fastest to compile
, no parentheses required
```

Come on, your turn:

Classes

Classes in Scala are static templates that can be instantiated into many objects at runtime. Here is a class definition which defines a class Point:

```
class Point(x: Int, y: Int) {

  override def toString(): String = "(" + x + ", " + y + ")"

}
```

The class defines two variables `x` and `y` and one method `toString`.

Classes in Scala are parameterized with constructor arguments. The code above defines two constructor arguments, `x` and `y`; they are both visible in the whole body of the class. In our example they are used to implement `toString`.

Classes are instantiated with the `new` primitive, as the following example will show:

```
object Classes {

  def main(args: Array[String]) {

    val pt = new Point(1, 2)

    println(pt)

  }

}
```

The program defines an executable application `Classes` in the form of a top-level singleton object with a `main` method. The `main` method creates a new `Point` and stores it in value `pt`.

Options

If you have worked with Java at all in the past, it is very likely that you have come across a `NullPointerException` at some time (other languages will throw similarly named errors in such a case). Usually this happens because some method returns null when you were not expecting it and thus not dealing with that possibility in your client code. A value of `null` is often abused to represent an absent optional value.

Scala tries to solve the problem by getting rid of `null` values altogether and providing its own type for representing optional values, i.e. values that may be present or not: the `Option[A]` trait.

`Option[A]` is a container for an optional value of type `A`. If the value of type `A` is present, the `Option[A]` is an instance of `Some[A]`, containing the present value of type `A`. If the value is absent, the `Option[A]` is the object `None`.

Tuples

Scala tuple combines a fixed number of items together so that they can be passed around as a whole. They are one-indexed. Unlike an array or list, a tuple can hold objects with different types but they are also immutable. Here is an example of a tuple holding an integer, a string, and the console:

```
val t = (1, "hello", Console)
```

Which is syntactic sugar (short cut) for the following:

```
val t = new Tuple3(1, "hello", Console)
```

Lists

Scala Lists are quite similar to arrays, which means all the elements of a list have the same type - but there are two important differences. First, lists are immutable, which means elements of a list cannot be changed by assignment. Second, lists represent a linked list whereas arrays are flat. The type of a list that has elements of type `T` is written as `List[T]`.

Pattern Matching

Scala has a built-in general pattern matching mechanism. It allows to match on any sort of data with a first-match policy. Here is a small example which shows how to match against an integer value:

```
object MatchTest1 extends App {

  def matchTest(x: Int): String = x match {

    case 1 => "one"

    case 2 => "two"

    case _ => "many" // case _ will trigger if all other cases fail.

  }

  println(matchTest(3)) // prints "many"

}
```

The block with the `case` statements defines a function which maps integers to strings. The `match` keyword provides a convenient way of applying a function (like the pattern matching function above) to an object.

Scala's pattern matching statement is most useful for matching on algebraic types expressed via `case classes`. Scala also allows the definition of patterns independently of case classes, using `unapply` methods in extractor objects.

Case classes

Scala supports the notion of *case classes*. Case classes are regular classes which export their constructor parameters and which provide a recursive decomposition mechanism via pattern matching.

Here is an example for a class hierarchy which consists of an abstract superclass `Term` and three concrete case classes `Var`, `Fun`, and `App`:

```
abstract class Term

case class Var(name: String) extends Term

case class Fun(arg: String, body: Term) extends Term

case class App(f: Term, v: Term) extends Term
```

This class hierarchy can be used to represent terms of the untyped lambda calculus. To facilitate the construction of case class instances, Scala does not require that the `new` primitive is used. One can simply use the class name as a function.

Here is an example:

```
Fun("x", Fun("y", App(Var("x"), Var("y"))))
```

The constructor parameters of case classes are treated as public values and can be accessed directly.

```
val x = Var("x")

println(x.name)
```

For every case class the Scala compiler generates an `equals` method which implements structural equality and a `toString` method. For instance,

```
val x1 = Var("x")

val x2 = Var("x")

val y1 = Var("y")

println("" + x1 + " == " + x2 + " => " + (x1 == x2))

println("" + x1 + " == " + y1 + " => " + (x1 == y1))
```

will print

```
Var(x) == Var(x) => true

Var(x) == Var(y) => false
```

It only makes sense to define case classes if pattern matching is used to decompose data structures.

Traits

Similar to interfaces in Java, traits are used to define object types by specifying the signature of the supported methods. Unlike Java, Scala allows traits to be partially implemented; i.e. it is possible to define default implementations for some methods. In contrast to classes, traits may not have constructor parameters.

Here is an example:

```
trait Similarity {

  def isSimilar(x: Any): Boolean

  def isNotSimilar(x: Any): Boolean = !isSimilar(x)

}
```

This trait consists of two methods `isSimilar` and `isNotSimilar`.
While `isSimilar` does not provide a concrete method implementation (it is abstract in the terminology of Java), method `isNotSimilar` defines a concrete implementation. Consequently, classes that integrate this trait only have to provide a concrete implementation for `isSimilar`. The behavior for `isNotSimilar` gets inherited directly from the trait.