Assertions and Tests
&
Case Classes and Pattern Matching

Assertions (1)

Assertions in Scala are written as calls of a predefined method assert.

The expression assert(condition) throws an AssertionError if condition does not hold.

There's also a two argument version of assert: assert(condition, explanation)

The type of explanation is Any. The assert method will call to String on it to get a string explanation to place inside the AssertionError.

Assertions (2)

You can add assertion in the method it self like this:

```
def above(that: Element): Element = {
        val this1 = this widen that.width
        val that1 = that widen this.width
        assert(this1.width == that1.width)
        elem(this1.contents ++ that1.contents)
 }
```

Assertions can be enabled and disabled using the JVM's -ea and -da command-line flags.

Uses the code itself as test cases.

Ensuring

The ensuring method can be used with any result type because of an implicit conversion.

```
private def widen(w: Int): Element =
    if (w <= width)
        this
    else {
        val left = elem(' ', (w - width) / 2, height)
        var right = elem(' ', w - width - left.width, height)
        left beside this beside right
    } ensuring (w <= _.width)
```

The _ is a placeholder for the one argument passed to the predicate, the Element result of the widen method.

TESTING IN SCALA (1)

You have many options for testing in Scala, from established Java tools, such as JUnit and TestNG, to tools written in Scala, such as ScalaTest, specs2, and ScalaCheck.

ScalaTest framework is customizable. For example FunSuite style.

```
import org.scalatest.FunSuite
import Element.elem
class ElementSuite extends FunSuite {
        test("elem result should have passed width") {
                val ele = elem('x', 2, 3)
                assert(ele.width == 2)
        }
}
```

TESTING IN SCALA (2)

The central concept in ScalaTest is the suite, a collection of tests.

In ScalaTest, you organize large test suites by nesting Suites inside Suites.

test(“”) is the name of the test. The test code is a function passed as a byname parameter to test, which registers it for later execution.

ScalaTest is integrated into common build tools (such as sbt and Maven) and IDEs (such as IntelliJ IDEA and Eclipse).

TESTING IN SCALA (3)

You can also run a Suite directly via ScalaTest's Runner application or from the Scala interpreter simply by invoking execute on it. Here's an example:

```
scala> (new ElementSuite).execute()
ElementSuite:
        - elem result should have passed width
```

INFORMATIVE FAILURE REPORTS

Informative error message:

```
scala> assert(width == 2)
org.scalatest.exceptions.TestFailedException:
        3 did not equal 2
```

For more information you can use DiagrammedAssertions

```
scala> assert(List(1, 2, 3).contains(4))
org.scalatest.exceptions.TestFailedException:
assert(List(1, 2, 3).contains(4))
           |    |  |  |  |            |
           |    1  2  3  false        4
         List(1, 2, 3)
```

Distinction between actual and expected

```
assertResult(2) {
        ele.width
}
```

With this expression you indicate that you expect the code between the curly braces to result in 2. Otherwise message "Expected 2, but got 3".

If you want to check that a method throws an expected exception, you can use ScalaTest's assertThrows method, like this:

```
assertThrows[IllegalArgumentException] {
        elem('x', -2, 3)
}
```

Intercept method

The intercept method works the same as assertThrows, except if the expected exception is thrown, intercept returns it:

```
val caught =
    intercept[ArithmeticException] {
        1 / 0
    }
assert(caught.getMessage == "/ by zero")
```

TESTS AS SPECIFICATIONS (FlatSpec)

```
class ElementSpec extends FlatSpec with Matchers {
        "A UniformElement" should
        "have a width equal to the passed value" in {
                val ele = elem('x', 2, 3)
                ele.width should be (2)
        }
        it should "have a height equal to the passed value" in {
                val ele = elem('x', 2, 3)
                ele.height should be (3)
        }
        it should "throw an IAE if passed a negative width" in {
                an [IllegalArgumentException] should be thrownBy {
                        elem('x', -2, 3)
                }
        }
    }
```

TESTS AS SPECIFICATIONS (FlatSpec)

Subject then the subsequent clauses.

It will run as a ScalaTest.

```
scala> (new ElementSpec).execute()
A UniformElement
        - should have a width equal to the passed value
        - should have a height equal to the passed value
        - should throw an IAE if passed a negative width
```

Specs2 framework same idea other style and keywords like Scenario, Given , When Then, Pending .

PROPERTY-BASED TESTING (ScalaCheck)

```
import org.scalatest.WordSpec
import org.scalatest.prop.PropertyChecks
import org.scalatest.MustMatchers._
import Element.elem

class ElementSpec extends WordSpec with PropertyChecks {
    "elem result" must {
        "have passed width" in {
            forAll { (w: Int) =>
                whenever (w > 0) {
                    elem('x', w, 3).width must equal (w)
                }
            }
        }
    }
}
```

Case Classes and Pattern Matching

Case classes and pattern matching, twin constructs that support you when writing regular, non-encapsulated data structures.

Case Classes

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr) extends Expr
```

Case modifier adds Factory method (no NEW keyword)

All arguments in the parameter list of a case class implicitly get a val prefix.

Implementations of methods toString, hashCode, equalsto and Copy

```
 scala> val v = Var("x")
scala> v.name, op.left
op.right == Var("x")
op.copy(operator = "-")
```

Pattern matching (1)

```
UnOp("-", UnOp("-", null))  => null   // Double negation
BinOp("+", null, Number(0)) => null   // Adding zero
BinOp("*", null, Number(1)) => null   // Multiplying by on

 scala> simplifyTop(UnOp("-", UnOp("-", Var("x"))))
 res4: Expr = Var(x)

def simplifyTop(expr: Expr): Expr = expr match {
        case UnOp("-", UnOp("-", e))  => e   // Double negation
        case BinOp("+", e, Number(0)) => e   // Adding zero
        case BinOp("*", e, Number(1)) => e   // Multiplying by one (e match)
        case _ => expr                       // matches every value
}
```

Pattern matching (2)

Match is an expression in Scala (i.e., it always results in a value).

Scala's alternative expressions never "fall through" into the next case.

if none of the patterns match, an exception named MatchError is thrown.

```
expr match {
        case BinOp(op, left, right) =>       println(expr + " is a binary operation")
        case _ =>
}
```

KINDS OF PATTERNS (1)

Wildcard patterns (_)

```
expr match {
        case BinOp(_, _, _) =>
                println(expr + " is a binary operation")
        case _ => println("It's something else")
}
```

Constant patterns

```
def describe(x: Any) = x match {
        case 5 => "five"
        case true => "truth"
        case _ => "something else"
}
```

KINDS OF PATTERNS (2)

Variable patterns

```
import math.{E, Pi}
scala> E match {
        case Pi => "strange math? Pi = " + Pi
        case _ => "OK"
}
res11: String = OK
```

```
scala> val pi = math.Pi
scala> E match {
        case pi => "strange math? Pi = " + pi
        (case _ => "OK") unreachable code pi is variable pattern and will match all inputs.
}
res12: String = strange math? Pi = 2.718281828459045
```

You can use `pi`, will be interpreted as a constant, not as a variable.

KINDS OF PATTERNS (3)

Constructor patterns

```
expr match {
        case BinOp("+", e, Number(0)) => println("a deep match")
        case _ =>
}
```

Checks if the top-level object is a BinOp, third constructor parameter is a Number, and that the value field of that number is 0.

Sequence patterns

A pattern that checks for a three-element list starting with zero

```
expr match {
        case List(0, _, _) => println("found it")
        case _ =>
}
```

Matches any list that starts with zero, regardless of how long the list is.

```
expr match {
        case List(0, _*) => println("found it")
        case _ =>
}
```

Tuple patterns

```
def tupleDemo(expr: Any) =
expr match {
        case (a, b, c)  =>  println("matched " + a + b + c)
        case _ =>
}

scala> tupleDemo(("a ", 3, "-tuple"))
matched a 3-tuple
```

Typed patterns

You can use a typed pattern as a convenient replacement for type tests and type casts.

```
def generalSize(x: Any) = x match {
        case s: String => s.length
        case m: Map[_, _] => m.size
        case _ => -1
}
scala> generalSize("abc")   (non-null)
res16: Int = 3

if (x.isInstanceOf[String]) {
        val s = x.asInstanceOf[String]
        s.length
} else ...
```

Typed patterns (Type erasure)

Scala uses the erasure model of generics, just like Java does.

You can’t do this:

```
scala> def isIntIntMap(x: Any) = x match {
        case m: Map[Int, Int] => true    // Could not know which type is declared
        case _ => false
}
```

You can do this:

```
scala> def isStringArray(x: Any) = x match {
        case a: Array[String] => "yes"    // only array
        case _ => "no"
}
```

Variable binding

```
expr match {
        case UnOp("abs", e @ UnOp("abs", _)) => e
        case _ =>
}
```

SEALED CLASSES

To cover all the cases you need to make superclass of your case classes sealed. A sealed class cannot have any new subclasses added except the ones in the same file.

```
sealed abstract class Expr
```

It means you only need to worry about the subclasses you already know about.

To remove warnings add @unchcked

```
def describe(e: Expr): String = (e: @unchecked) match {...
```

PATTERNS EVERYWHERE (Patterns in variable definitions)

In other parts of Scala

scala> val myTuple = (123, "abc")

scala> ***val (number, string)*** *= myTuple*

Or

val exp = new BinOp("", Number(5), Number(1))*

val BinOp(op, left, right) = exp

PATTERNS EVERYWHERE (Case sequences as partial functions)

A sequence of cases (i.e., alternatives) in curly braces can be used anywhere a function literal can be used.

```
val withDefault: Option[Int] => Int = {
        case Some(x) => x
        case None => 0
}
scala> withDefault(Some(10))  res28: Int = 10
scala> withDefault(None)  res29: Int = 0
```

Partial function (1)

Use PartialFunction to test whether the function is defined at a particular value.

A function works for every argument of the defined type. In other words, a function defined as (Int) => String takes any Int and returns a String.
A Partial Function is only defined for certain values of the defined type. A Partial Function (Int) => String might not accept every Int.
isDefinedAt is a method on PartialFunction that can be used to determine if the PartialFunction will accept a given argument.

Partial function (2)

```
scala> val one: PartialFunction[Int, String] = {
          case 1 => "one"
}

scala> one.isDefinedAt(1)
res0: Boolean = true

scala> one.isDefinedAt(2)
res1: Boolean = false

scala> one(1)
res2: String = one
```

PATTERNS EVERYWHERE (Patterns in for expressions)

```
scala> for ((country, city) <- capitals)
println("The capital of " + country + " is " + city)
The capital of France is Paris
The capital of Japan is Tokyo
```