

Chapter 14

Assertions and Tests

Assertions and tests are two important ways you can check that the software you write behaves as you expect. In this chapter, we'll show you several options you have in Scala to write and run them.

14.1 ASSERTIONS

Assertions in Scala are written as calls of a predefined method `assert`.^[1] The expression `assert(condition)` throws an `AssertionError` if `condition` does not hold. There's also a two-argument version of `assert`: The expression `assert(condition, explanation)` tests `condition` and, if it does not hold, throws an `AssertionError` that contains the given explanation. The type of explanation is `Any`, so you can pass any object as the explanation. The `assert` method will call `toString` on it to get a string explanation to place inside the `AssertionError`. For example, in the method named "above" of class `Element`, shown in Listing 10.13 here, you might place an `assert` after the calls to `widen` to make sure that the widened elements have equal widths. This is shown in Listing 14.1.

```
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)  
}
```

Listing 14.1 - Using an assertion.

Another way you might choose to do this is to check the widths at the end of the `widen` method, right before you return the value. You can accomplish this by storing the result in `aval`, performing an assertion on the result, then mentioning the `val` last so the result is returned if the assertion succeeds. However, you can do this more concisely with a convenience method in `Predef` named `ensuring`, as shown in Listing 14.2.

The `ensuring` method can be used with any result type because of an implicit conversion. Although it looks in this code as if we're invoking `ensuring` on `widen`'s result, which is type `Element`, we're actually invoking `ensuring` on a type to which `Element` is implicitly converted. The `ensuring` method takes one argument, a predicate function that takes a result type and returns `Boolean`, and passes the result to the predicate. If the predicate returns `true`, `ensuring` will return the result; otherwise, `ensuring` will throw an `AssertionError`.

In this example, the predicate is `"w <= _.width"`. The underscore is a placeholder for the one argument passed to the predicate, the `Element` result of the `widen` method. If the width passed as `w` to `widen` is less than or equal to the width of the result `Element`, the predicate will result in `true`, and `ensuring` will result in the `Element` on which it was invoked. Because this is the last expression of the `widen` method, `widen` itself will then result in the `Element`.

```

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)

```

Listing 14.2 - Using ensuring to assert a function's result.

Assertions can be enabled and disabled using the JVM's `-ea` and `-da` command-line flags. When enabled, each assertion serves as a little test that uses the actual data encountered as the software runs. In the remainder of this chapter, we'll focus on the writing of external tests, which provide their own test data and run independently from the application.

14.2 TESTING IN SCALA

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. For the remainder of this chapter, we'll give you a quick tour of these tools. We'll start with ScalaTest.

ScalaTest is the most flexible Scala test framework: it can be easily customized to solve different problems. ScalaTest's flexibility means teams can use whatever testing style fits their needs best. For example, for teams familiar with JUnit, the FunSuite style will feel comfortable and familiar. Listing 14.3 shows an example.

```

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)
  }
}

```

Listing 14.3 - Writing tests with FunSuite.

The central concept in ScalaTest is the suite, a collection of tests. A test can be anything with a name that can start and either succeed, fail, be pending, or canceled. Trait Suite is the central unit of composition in ScalaTest. Suite declares "lifecycle" methods defining a default way to run tests, which can be overridden to customize how tests are written and run.

ScalaTest offers style traits that extend Suite and override lifecycle methods to support different testing styles. It also provides mixin traits that override lifecycle methods to address particular testing needs. You define test classes by composing Suite style and mixin traits, and define test suites by composing Suite instances.

FunSuite, which is extended by the test class shown in Listing 14.3, is an example of a testing style. The "Fun" in FunSuite stands for function; "test" is a method defined in FunSuite, which is invoked by the primary constructor of ElementSuite. You specify the name of the test as a string between the parentheses and the test code itself between curly braces. The test code is a function passed as a by-name 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). 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
```

All ScalaTest styles, including FunSuite, are designed to encourage the writing of focused tests with descriptive names. In addition, all styles generate specification-like output that can facilitate communication among stakeholders. The style you choose dictates only how the declarations of your tests will look. Everything else in ScalaTest works consistently the same way no matter what style you choose.[2]

14.3 INFORMATIVE FAILURE REPORTS

The test in Listing 14.3 attempts to create an element of width 2 and assert that the width of the resulting element is indeed 2. Were this assertion to fail, the failure report would include the filename and line number of the offending assertion, and an informative error message:

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

To provide descriptive error messages when assertions fail, ScalaTest analyzes the expressions passed to each assert invocation at compile time. If you prefer to see even more detailed information about assertion failures, you can use ScalaTest's DiagrammedAssertions, whose error messages display a diagram of the expression passed to assert:

```
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)
```

ScalaTest's assert methods do not differentiate between the actual and expected result in error messages. They just indicate that the left operand did not equal the right operand, or show the values in

a diagram. If you wish to emphasize the distinction between actual and expected, you can alternatively use `ScalaTest's assertResult` method, like this:

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

With this expression you indicate that you expect the code between the curly braces to result in 2. Were the code between the braces to result in 3, you'd see the message, "Expected 2, but got 3" in the test failure report.

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)  
}
```

If the code between the curly braces throws a different exception than expected, or throws no exception, `assertThrows` will complete abruptly with a `TestFailedException`. You'll get a helpful error message in the failure report, such as:

```
Expected IllegalArgumentException to be thrown,  
but NegativeArraySizeException was thrown.
```

On the other hand, if the code completes abruptly with an instance of the passed exception class, `assertThrows` will return normally. If you wish to inspect the expected exception further, you can use `intercept` instead of `assertThrows`. 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")
```

In short, `ScalaTest's` assertions work hard to provide useful failure messages that will help you diagnose and fix problems in your code.

14.4 TESTS AS SPECIFICATIONS

In the *behavior-driven development* (BDD) testing style, the emphasis is on writing human-readable specifications of the expected behavior of code and accompanying tests that verify the code has the specified behavior. `ScalaTest` includes several traits that facilitate this style of testing. An example using one such trait, `FlatSpec`, is shown in Listing 14.4.

```
import org.scalatest.FlatSpec  
import org.scalatest.Matchers  
import Element.elem
```

```

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)
      }
    }
}

```

Listing 14.4 - Specifying and testing behavior with a ScalaTest FlatSpec.

In a FlatSpec, you write tests as *specifier clauses*. You start by writing a name for the *subject* under test as a string ("A UniformElement" in Listing 14.4), then should (or must or can), then a string that specifies a bit of behavior required of the subject, then in. In the curly braces following in, you write code that tests the specified behavior. In subsequent clauses you can write it to refer to the most recently given subject. When a FlatSpec is executed, it will run each specifier clause as a ScalaTest test. FlatSpec (and ScalaTest's other specification traits) generate output that reads like a specification when run. For example, here's what the output will look like if you run ElementSpec from Listing 14.4 in the interpreter:

```

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

```

Listing 14.4 also illustrates ScalaTest's *matchers* domain-specific language (DSL). By mixing in trait Matchers, you can write assertions that read more like natural language. ScalaTest provides many matchers in its DSL, and also enables you to define new matchers with custom failure messages. The matchers shown in Listing 14.4 include the "should be" and "an [...] should be thrownBy { ... }" syntax. You can alternatively mix in MustMatchers if you prefer must to should. For example, mixing in MustMatchers would allow you to write expressions such as:

```

result must be >= 0
map must contain key 'c'

```

If the last assertion failed, you'd see an error message similar to:

```

Map('a' -> 1, 'b' -> 2) did not contain key 'c'

```

The [specs2 testing framework](#), an open source tool written in Scala by Eric Torreborre, also supports the BDD style of testing but with a different syntax. For example, you could use specs2 to write the test shown in Listing 14.5:

```
import org.specs2._
import Element.elem

object ElementSpecification extends Specification {
  "A UniformElement" should {
    "have a width equal to the passed value" in {
      val ele = elem('x', 2, 3)
      ele.width must be_==(2)
    }
    "have a height equal to the passed value" in {
      val ele = elem('x', 2, 3)
      ele.height must be_==(3)
    }
    "throw an IAE if passed a negative width" in {
      elem('x', -2, 3) must
        throwA[IllegalArgumentException]
    }
  }
}
```

Listing 14.5 - Specifying and testing behavior with the specs2 framework.

Like [ScalaTest](#), [specs2](#) provides a matchers DSL. You can see some examples of specs2 matchers in action in Listing 14.5 in the lines that contain "must be _==" and "must throwA".^[3] You can use specs2 standalone, but it is also integrated with [ScalaTest](#) and [JUnit](#), so you can run specs2 tests with those tools as well.

One of the big ideas of BDD is that tests can be used to facilitate communication between the people who decide what a software system should do, the people who implement the software, and the people who determine whether the software is finished and working. Although any of [ScalaTest's](#) or [specs2's](#) styles can be used in this manner, [ScalaTest's FeatureSpec](#) in particular is designed for it. Listing 14.6 shows an example:

```
import org.scalatest._

class TVSetSpec extends FeatureSpec with GivenWhenThen {

  feature("TV power button") {
    scenario("User presses power button when TV is off") {
      Given("a TV set that is switched off")
      When("the power button is pressed")
      Then("the TV should switch on")
      pending
    }
  }
}
```

Listing 14.6 - Using tests to facilitate communication among stakeholders.

FeatureSpec is designed to guide conversations about software requirements: You must identify specific features, then specify those features in terms of scenarios. The Given, When, and Then methods (provided by trait GivenWhenThen) can help focus the conversation on the specifics of individual scenarios. The pending call at the end indicates that neither the test nor the actual behavior has been implemented—just the specification. Once all the tests and specified behavior have been implemented, the tests will pass and the requirements can be deemed to have been met.

14.5 PROPERTY-BASED TESTING

Another useful testing tool for Scala is ScalaCheck, an open source framework written by Rickard Nilsson. ScalaCheck enables you to specify properties that the code under test must obey. For each property, ScalaCheck will generate data and execute assertions that check whether the property holds. Listing 14.7 shows an example of using ScalaCheck from a ScalaTest WordSpec that mixes in trait PropertyChecks.

```
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)
        }
      }
    }
  }
}
```

Listing 14.7 - Writing property-based tests with ScalaCheck.

WordSpec is a ScalaTest style class. The PropertyChecks trait provides several forAll methods that allow you to mix property-based tests with traditional assertion-based or matcher-based tests. In this example, we check a property that the elem factory should obey. ScalaCheck properties are expressed as function values that take as parameters the data needed by the property's assertions. This data will be generated by ScalaCheck. In the property shown in Listing 14.7, the data is an integer named w that represents a width. Inside the body of the function, you see this code:

```
whenever (w > 0) {
  elem('x', w, 3).width must equal (w)
}
```

The whenever clause indicates that whenever the left hand expression is true, the expression on the right must hold true. Thus in this case, the expression in the block must hold true whenever w is greater than 0. The right-hand expression in this case will yield true if the width passed to the elem factory is the same as the width of the Element returned by the factory.

With this small amount of code, ScalaCheck will generate possibly hundreds of values for `wand` and test each one, looking for a value for which the property doesn't hold. If the property holds true for every value ScalaCheck tries, the test will pass. Otherwise, the test will complete abruptly with a `TestFailedException` that contains information including the value that caused the failure.

14.6 ORGANIZING AND RUNNING TESTS

Each framework mentioned in this chapter provides some mechanism for organizing and running tests. In this section, we'll give a quick overview of ScalaTest's approach. To get the full story on any of these frameworks, however, you'll need to consult their documentation.

In ScalaTest, you organize large test suites by nesting Suites inside Suites. When a Suite is executed, it will execute its nested Suites as well as its tests. The nested Suites will in turn execute their nested Suites, and so on. A large test suite, therefore, is represented as a tree of Suite objects. When you execute the root Suite in the tree, all Suites in the tree will be executed.

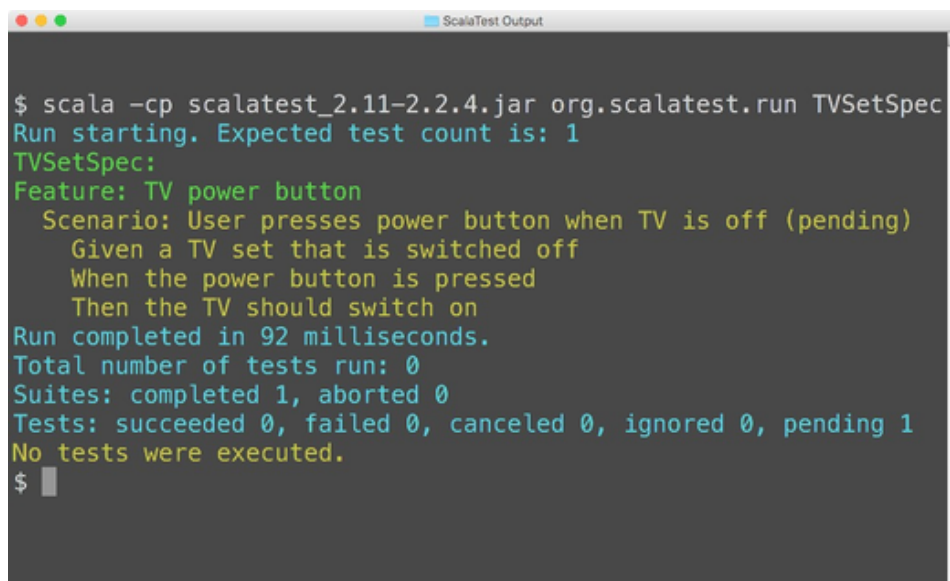
You can nest suites manually or automatically. To nest manually, you either override the `nestedSuites` method on your Suites or pass the Suites you want to nest to the constructor of class `Suites`, which ScalaTest provides for this purpose. To nest automatically, you provide package names to ScalaTest's Runner, which will discover Suites automatically, nest them under a root Suite, and execute the root Suite.

You can invoke ScalaTest's Runner application from the command line or via a build tool, such as `sbt`, `maven`, or `ant`. The simplest way to invoke Runner on the command line is via the `org.scalatest.run` application. This application expects a fully qualified test class name. For example, to run the test class shown in Listing 14.6, you must compile it with:

```
$ scalac -cp scalatest.jar TVSetSpec.scala
```

Then you can run it with:

```
$ scala -cp scalatest.jar org.scalatest.run TVSetSpec
```

```
$ scala -cp scalatest_2.11-2.2.4.jar org.scalatest.run TVSetSpec
Run starting. Expected test count is: 1
TVSetSpec:
  Feature: TV power button
    Scenario: User presses power button when TV is off (pending)
      Given a TV set that is switched off
      When the power button is pressed
      Then the TV should switch on
Run completed in 92 milliseconds.
Total number of tests run: 0
Suites: completed 1, aborted 0
Tests: succeeded 0, failed 0, canceled 0, ignored 0, pending 1
No tests were executed.
$
```

Figure 14.1 - The output of `org.scalatest.run`.

With `-cp` you place ScalaTest's JAR file on the class path. (When downloaded, the JAR file name will include embedded Scala and ScalaTest version numbers.) The next token, `org.scalatest.run`, is the fully qualified application name. Scala will run this application and pass the remaining tokens as command line arguments. The `TVSetSpec` argument specifies the suite to execute. The result is shown in Figure 14.1.

14.7 CONCLUSION

In this chapter you saw examples of mixing assertions directly in production code, as well as writing them externally in tests. You saw that as a Scala programmer, you can take advantage of popular testing tools from the Java community, such as JUnit and TestNG, as well as newer tools designed explicitly for Scala, such as ScalaTest, ScalaCheck, and specs2. Both in-code assertions and external tests can help you achieve your software quality goals. We felt that these techniques are important enough to justify the short detour from the Scala tutorial that this chapter represented. In the next chapter, however, we'll return to the language tutorial and cover a very useful aspect of Scala: pattern matching.

Footnotes for Chapter 14:

- [1] The `assert` method is defined in the `Predef` singleton object, whose members are automatically imported into every Scala source file.
- [2] More detail on ScalaTest is available from <http://www.scalatest.org/>.
- [3] You can download specs2 from <http://specs2.org/>.