
Scala TDD Documentation

Release 1.0

Kaylor, Läufer, and Thiruvathukal

August 02, 2015

CONTENTS

1	Testing Fundamentals	1
1.1	Dependencies	1
1.2	Cloning the Exemplar	1
1.3	The Testing Mindset - Running Tests on Checkout	2
1.4	Assertions	4
1.5	A Basic Example: Rational Arithmetic	7
1.6	Test-driven development with JUnit and Scala	10
1.7	Behavior-Driven Testing using ScalaTest	12
1.8	Discussion	16
2	Testing Environment and the Simple Build Tool (SBT)	19
2.1	Brief History of Build Tools	19
2.2	Installing sbt	20
2.3	Configuring sbt	20
2.4	Rational Numbers Recap	23
2.5	Testing with sbt	24
2.6	Plugin Ecosystem	26
2.7	IDE Option: JetBrains IntelliJ IDEA	26
2.8	IDE Option: Eclipse Scala IDE	28
2.9	IDE Choice: Codio Instant Cloud-Based Environment	30
3	Basic Testing	31
3.1	A Basic Example: Rational Arithmetic	31
3.2	Scala + JUnit to Test Java Rational Class	33
4	Mocking	37
5	Advanced	39
6	Continuous Integration	41
6.1	Continuous Integration Products	41
6.2	Team City	41
6.3	Jenkins	49
6.4	Scaling Continuous Integration	52
6.5	Continuous Integration Frequency	53
6.6	False Positives and Periodic Failure in Computationally or Memory Bound Tests	54
7	UI Testing	57
8	Advanced Topics	59
9	Indices and tables	61

TESTING FUNDAMENTALS

This chapter is focused on the essentials of testing. Because we know that most readers are likely to come to Scala from a Java (or similar) environment based on JUnit, which is one of the one of the xUnit variants, we will begin this discussion with an example of how to use JUnit within Scala, since Java can be used and mixed into Scala programs. We'll then take a look at a couple of introductory (and integrated) Scala examples and how to write them using JUnit and ScalaTest.

1.1 Dependencies

You need to install the Scala Build Tool ¹ - The details of this tool are covered in the [Testing Environment and the Simple Build Tool \(SBT\)](#). There is no need to install Scala separately. What we'll be doing in this chapter uses SBT to install Scala on demand.

You must also install the Git distributed version control system (DVCS) but it is already on your computer if you are running OS X or Linux. Git is available for all major platforms ². Git is one of the preeminent version control systems and is used by GitHub ³, one of the most popular—and we think, best—sites for hosting free and open source projects. We use it for almost all of our public-facing projects.

Note: Many folks often think of Git (the tool) and GitHub (a social coding service built around Git) as being one in the same. You can use Git with or without a hosted service and might feel the need to do so, especially when your project isn't open source or you wish to keep it private. We take advantage of one such service in our own work, Bitbucket ⁴, which offers a number of features for corporate customers and academic software projects. For example, we write our papers in Markdown and LaTeX and often keep them private—at least until they are accepted for publication. Our book is also an example of where we use Git on two systems—one for the book chapters (our publisher wants to sell books) and several for the code examples.

To do the examples in this chapter, you must install both of these. Installation is not covered as it is platform-specific. However, we point you to web materials.

1.2 Cloning the Exemplar

An important defining principle of our book is that the text and examples should remain relevant long after you have decided the book is no longer needed. We keep all of our examples under version control. In addition to our dear readers, countless students, research collaborators, and we, ourselves, depend on the examples continuing to work long after we create them. It might go without saying, but we think there's nothing worse than getting a book where

¹ Scala Build Tool, <http://www.scala-sbt.org/>

² Git, <http://git-scm.com>

³ GitHub, <http://github.com>

⁴ <http://bitbucket.org>

there is a typo or compilation error and having to waste time trying to convince the author to cough up the latest code (or fix it).

So without further ado, let's get the code for this chapter.

```
$ git clone
https://github.com/LoyolaChicagoCode/scala-tdd-fundamentals.git

$ cd scala-tdd-fundamentals
$ ls
```

At this point, it is worth noting that we do most of our work in a Unix style environment. You can also checkout our project using IntelliJ (covered in the next chapter) but we're mostly going to operate at the command line in this chapter. It really does help to illuminate the principles.

Note: It is also worth noting at this point that we are command-line junkies. When it comes to things we are doing in a terminal session (e.g. entering commands and seeing output), at times we will need to break lines up for the purpose of formatting the book. Notably, the `git clone` command is entered entirely on one line without hitting the return (a.k.a. enter) key. Similarly, when we should show you the output of a command, it is entirely possible the output you observe will be slightly different.

1.3 The Testing Mindset - Running Tests on Checkout

As you now have the source code, let's make sure we can compile and run the tests using `sbt`. We consider it a sign of a healthy project where you can immediately test the code as soon as you've downloaded it. We also think it can be part of a growth mindset when it comes to coding. If you think of testing as fun, you're more likely to do it. Not to sound hokey, but when you see others having fun when they checkout your code and gleefully observe the tests passing, you are multiplying the fun!

The following *compiles* the code we just checked out via `git`. `sbt` generates a lot of output, mainly aimed at *helping* you when something goes wrong. Because dependencies are being pulled from the internet for the unit testing frameworks (JUnit and ScalaTest) there is a chance you could see an error message but it is extremely unlikely. We've written the build file in such a way that it will evolve with Scala and its many moving parts.

```
[info] Set current project to SimpleTesting (in build
file:/Users/gkt/Dropbox/Work/LUC/scala-tdd-fundamentals/)
[info] Updating
{file:/Users/gkt/Dropbox/Work/LUC/scala-tdd-fundamentals/}sc
ala-tdd-fundamentals...
[info] Resolving org.scala-lang#scala-library;2.11.4 ...
[info] Resolving com.novocode#junit-interface;0.11 ...
[info] Resolving junit#junit;4.11 ...
[info] Resolving org.hamcrest#hamcrest-core;1.3 ...
[info] Resolving org.scala-sbt#test-interface;1.0 ...
[info] Resolving org.scalatest#scalatest_2.11;2.2.1 ...
[info] Resolving org.scala-lang#scala-reflect;2.11.2 ...
[info] Resolving
org.scala-lang.modules#scala-xml_2.11;1.0.2 ...
[info] Resolving org.scala-lang#scala-compiler;2.11.4 ...
[info] Resolving org.scala-lang#scala-reflect;2.11.4 ...
[info] Resolving
org.scala-lang.modules#scala-parser-combinators_2.11;1.0.2
...
[info] Resolving jline#jline;2.12 ...
[info] Done updating.
[info] Compiling 4 Scala sources to
```

```
/Users/gkt/Dropbox/Work/LUC/scala-tdd-fundamentals/target/sc
ala-2.11/classes...
[warn] there was one feature warning; re-run with -feature
for details
[warn] one warning found
[success] Total time: 3 s, completed Dec 16, 2014 3:41:07 PM
```

The most important line to observe in this output is the one beginning with `[success]`. If for any reason you don't see this line when running it yourself, it is likely that something failed, more than likely a network connectivity problem (or some repository became unavailable, we hope, temporarily).

If you've indeed encountered success in building our code with `sbt`, the next step is to run the tests. This is achieved by running `sbt test`.

```
[info] RationalScalaTestFlatSpecMatchers:
[info] GCD involving 0
[info] - should give y for gcd(0, y)
[info] - should give x for gcd(x, 0)
[info] Initializing
[info] - should reduce 2/4 to 1/2
[info] - should reduce -2/4 to -1/2
[info] - should reduce -3/-6 to 1/2
[info] Arithmetic
[info] - should perform addition
[info] - should perform subtraction
[info] - should perform multiplication
[info] - should perform division
[info] - should perform the reciprocal
[info] - should perform negation
[info] Comparisons
[info] - should perform ==
[info]   + numeric equality (==) works
[info]   + object equality works, e.g. equals()
[info]   + hashCode() works
[info] - should perform <
[info]   + operator < works
[info] - should perform >
[info]   + operator > works
[info] - should perform <= for something < and for
something =
[info]   + operator <= works for < case
[info]   + operator <= works for == case
[info] - should perform >= for something > and for
something =
[info]   + operator >= works for > case
[info]   + operator >= works for == case
[info] Passed: Total 37, Failed 0, Errors 0, Passed 37
[success] Total time: 6 s, completed Dec 16, 2014 3:41:29 PM
```

Again there is a *lot* of output generated, and what you see may differ from what you see in this console output, owing to changes that occur between the time a book is published and other changes that we or the Scala developers may make.

You'll see a number of lines that begin with `[info]`. This indicates that informative messages about tests are being written to the console. For our example, there are various tests being performed on our domain model (rational numbers). We have organized the tests according to the different methods in our implementation. Here, you can observe that we test everything from an internal helper method (the greatest common divisor), to initializer (construction) methods, and the each related groups of methods (for arithmetic, helpers, and equality/hashcode for use with other Scala classes).

While the approach to checking out code and testing it without knowing any details might appear a bit odd at first, it is deliberate on our part. There is a *mindset* to testing. When you do testing right, you should be able to check out someone's code, compile it, and run the tests. We'd also like to hope that when we checkout your code, the first thing we can do is to run tests and see evidence of good software engineering!

Without further ado, let's get started by looking at some code.

1.4 Assertions

A key notion of test-driven development and behavior-driven development is the ability to make a logical assertion about something that generally must hold *true* if the test is to pass. Because this is so fundamental to a *testing mindset*, we cover it separately before diving into various testing approaches.

Todo

Joe - In my experience, the built in language assertions are usually a debug-build only sanity check that wouldn't make sense for the cases where you'd normally use exceptions. There is a strong case for being able to run unit tests in debug, release, and obfuscated release modes to validate your product. Sometimes these language level asserts don't work in these cases.

Assertions are not a standard language feature in Scala. (They are in other languages but often without the other good stuff.) Instead, there are a number of classes that provide functions for assertion handling. In the framework we are using to introduce unit testing (JUnit), a class named `Assert` supports assertion testing (class) methods. When we move to `ScalaTest` (for good), we'll find that the need for explicit assertions is significantly reduced but they can still be useful.

In our tests, we make use of assertion method, e.g. `Assert.assert()`, to determine whether an assertion is successful. If the variable or expression passed to this method is *false*, the assertion fails.

Here are some examples of assertions, JUnit style. Readers should consult ⁵ for details of all supported methods. Although these are Java, we will be using them in Scala (Scala can use any Java class, a salient feature of the language).

Let's take a look at how to test drive the API via `sbt test:console`. The details of `sbt` are covered in the [Testing Environment and the Simple Build Tool \(SBT\)](#) chapter.

Let's fire up the Scala test console:

```
$ sbt test:console
[info] Set current project to SimpleTesting (in build
file:/Users/gkt/Dropbox/Work/LUC/scala-tdd-fundamentals/)
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.11.4 (Java HotSpot(TM) 64-Bit
Server VM, Java 1.8.0_25).
Type in expressions to have them evaluated.
Type :help for more information.
```

Now let's import JUnit and try out the assertion methods.

```
scala> import org.junit.Assert._
import org.junit.Assert._

scala> org.junit.Assert.<tab>
asInstanceOf      assertNotEquals   assertSame        isInstanceOf
assertArrayEquals assertNotNull    assertThat        toString
```

⁵ <http://junit.org>

assertEquals	assertNotSame	assertTrue
assertFalse	assertNull	fail

Perhaps one of the most awesome things in Scala's test console is that you can *discover* the available methods by using tab completion (a long-time favorite of Linux/Unix geeks like us).

Let's try an assertion that we know would be successful.

```
scala> assertTrue(true)
scala> assertFalse(false)
```

When an assertion is *successful*, you will see *no output*. On the other hand, when an assertion is *not successful* or *fails*, you see the dreaded *stack trace*.

```
scala> assertTrue(false)
java.lang.AssertionError
at org.junit.Assert.fail(Assert.java:86)
at org.junit.Assert.assertTrue(Assert.java:41)
at org.junit.Assert.assertTrue(Assert.java:52)
... 43 elided
```

You can see more information about the exception as follows:

```
scala> lastException.printStackTrace
```

Let's look at some of the other assertions. `assertArrayEquals()` is an interesting one!

```
scala> val v1 = Array(1, 2, 3)
v1: Array[Int] = Array(1, 2, 3)

scala> val v2 = Array(1, 2)
v2: Array[Int] = Array(1, 2)

scala> assertArrayEquals(v1, v2)
java.lang.AssertionError: array lengths differed,
expected.length=3 actual.length=2
    at org.junit.Assert.fail(Assert.java:88)
    at org.junit.internal.ComparisonCriteria.assertArraysAreSame
      Length(ComparisonCriteria.java:71)
    at
org.junit.internal.ComparisonCriteria.arrayEquals(Comparison
Criteria.java:32)
    at
org.junit.Assert.internalArrayEquals(Assert.java:473)
    at org.junit.Assert.assertArrayEquals(Assert.java:369)
    at org.junit.Assert.assertArrayEquals(Assert.java:380)
    ... 43 elided

scala> val v3 = Array(1, 2, 3)
v3: Array[Int] = Array(1, 2, 3)

scala> assertArrayEquals(v1, v3)
```

For arrays to be considered equal, they must match in type, length, and content. For this reason, only `v1` and `v3` compare equal as arrays.

If we try to compare two arrays of *different* types, we won't get very far. Consider:

```
scala> val v4 = Array(1.0, 2.0)
v4: Array[Double] = Array(1.0, 2.0)
```

```
scala> assertEquals(v1, v4)
<console>:13: error: overloaded method value assertEquals
with alternatives:
  (x$1: Array[Long], x$2: Array[Long])Unit <and>
  (x$1: Array[Int], x$2: Array[Int])Unit <and>
  (x$1: Array[Short], x$2: Array[Short])Unit <and>
  (x$1: Array[Char], x$2: Array[Char])Unit <and>
  (x$1: Array[Byte], x$2: Array[Byte])Unit <and>
  (x$1: Array[Object], x$2: Array[Object])Unit
cannot be applied to (Array[Int], Array[Double])
      assertEquals(v1, v4)
      ^
```

Remember that we're trying out Scala's assertion methods *interactively* in the REPL. The carat symbol just before `assertEquals(v1, v4)` means that this is a *syntax error*. Why? The problem is that `v1` is a Scala `Array[Int]`, while `v4` is a Scala `Array[Double]`—a problem! So this assertion neither passes nor fails. Luckily, when you write your unit tests normally, they will be within a Scala module (often within a class) and full syntactic and type checking will be performed before any test is run. The lowdown is that Scala requires your assertions to compile cleanly before they are actually executed—even within the REPL.

So we know that `assertEquals(v1, v3)` is successful. Let's look at what happens when we use `assertEquals(v1, v3)`. The results may surprise you!

```
scala> assertEquals(v1, v3)
java.lang.AssertionError: expected:<[I@3533e8ba> but
was:<[I@386e9a04>
    at org.junit.Assert.fail(Assert.java:88)
    at org.junit.Assert.failNotEquals(Assert.java:743)
    at org.junit.Assert.assertEquals(Assert.java:118)
    at org.junit.Assert.assertEquals(Assert.java:144)
    ... 43 elided
```

What's going on here?

Well, if you look carefully, there are two different object identifiers here. Although `v1` and `v3` have the same array values, they are by no means equal as in object equality. Further examination reveals what might be going on:

```
scala> v1.hashCode
res13: Int = 892594362

scala> v2.hashCode
res14: Int = 1561484803

scala> v3.hashCode
res15: Int = 946772484
```

Aha! `v1` and `v3` refer to identical arrays, at least in terms of their content, but their hash codes don't match. This is often a sign of *not being equal*, especially in object-oriented thinking.

Luckily, a unit-testing framework allows you to avoid looking at the dreaded stack trace. It will cheerfully intercept the `AssertionError` in the test runner. (This chapter doesn't use the IDE so everyone will have a common basis for thinking about tests that relies only on Scala's standard toolset.)

Todo

Add demonstrations of a few others here, especially those that are useful to our actual test cases.

1.5 A Basic Example: Rational Arithmetic

We begin with a *guiding example* that has been conceived with the following objectives in mind:

- Easy to explain and (likely) familiar to most readers. It relies on mathematical ideas that are taught to us in childhood.
- Has sufficient complexity to make testing necessary.
- Plays to Scala's strengths as a language.
- Allows us to introduce all of the testing styles without getting bogged down by domain-specific details.

This example is also featured in Martin Odersky's seminal introduction to Scala, a.k.a. *Scala by Example*⁶. Harrington and Thiruvathukal also present a version with more complete API and robust unit testing in their introductory CS1 course⁷.

While we're reasonably certain you already know what a rational number is, it is helpful to understand its requirements. Later, we shall see that these requirements can be expressed in the various testing styles—a form of documentation.

1. A rational number is expressed as a quotient of integers with a *numerator* and a *denominator*.
2. The *denominator* must not be zero.
3. The *numerator* and *denominator* are always kept in a reduced form. That is, if the *numerator* is 2 and the *denominator* is 4, you would expect the reduced form to be *numerator* = 1 and *denominator* = 2.
4. Rational numbers must be able to perform the usual rules of arithmetic, including binary operations such as +, -, *, and / (explained below) and various *convenience* operations such as negation and reciprocal.
5. Any two rational numbers that are the same number should compare equally and be treated as the same number anywhere they might be used. For example, if you used a Scala set to keep a set of rational numbers (e.g. 1/2, 2/3, 2/4) you would expect this set to contain 1/2 and 2/3. 2/4 wouldn't be expected to appear in the set.

1.5.1 Implementation

Central to making rational numbers work is a famous algorithm, attributable to Euclid, that computes the greatest common divisor. While one of the most important algorithms, it isn't provided as a standard Scala function. Even so, it makes for kind of an interesting testing example, because in the case of Rational numbers, it really needs to work for positive and negative values (and all possible combinations in the numerator and denominator) and do what is expected.

```
def gcd(x: Int, y: Int): Int = {
  if (x == 0) y
  else if (x < 0) gcd(-x, y)
  else if (y < 0) -gcd(x, -y)
  else gcd(y % x, x)
}
```

We won't rehash the details of how Euclid's algorithm actually does what it does. This is covered extensively in online resources. The following is the complete implementation of the Rational class. We will explore this in a bit of detail to ensure the ensuing discussion of unit testing makes complete sense.

The following is the complete implementation of the Rational class. We will explore this in a bit of detail to ensure the ensuing discussion of unit testing makes complete sense.

We'll present this in pieces. Here is a general outline of how the Rational class is organized:

⁶ Martin Odersky, *Scala by Example*, <http://www.scala-lang.org/docu/files/ScalaByExample.pdf>.

⁷ Andrew N. Harrington and George K. Thiruvathukal, *Introduction to Computer Science with C# and Mono*, <http://introcs.cs.luc.edu>.

```
class Rational(n: Int, d: Int) extends Ordered[Rational] {
```

```
  def gcd(x: Int, y: Int) = ...

  // initialization
  private val g = gcd(n, d)
  val numerator: Int = n / g
  val denominator: Int = d / g

  // arithmetic
  def +(that: Rational) = ...
  def -(that: Rational) = ...
  def *(that: Rational) = ...
  def /(that: Rational) = ...

  // important unary operations
  def reciprocal() = ...
  def negate() = ...

  // comparisons
  def compare(that: Rational) = ...

  // objects
  override def equals(o: Any) = ...
  override def hashCode = ...

  // companion object
  object Rational ...
```

We think the `Rational` class is a great example that plays to Scala’s strengths, especially when it comes to clarity, conciseness, and correctness. Let’s take a look at the implementation of this class in detail.

For readers new to Scala, you can start by thinking of Scala as “Java done more concisely”. The class definition basically gives us all we need to construct instances. A `Rational` number has a numerator and a denominator, n and d , respectively.

```
val num: Int = n / g
val den: Int = d / g

// perform test quotient for possible ArithmeticException
val testQuotient = num / den
```

Initializing *members* is a matter of creating Scala `val` definitions. Our `Rational` implementation is intended to be *side-effect free*. We initialize the *numerator* and *denominator* by dividing out the greatest-common divisor. This has the effect of reducing the fraction

Let’s take a look at the methods that deal with basic arithmetic.

Rational number arithmetic is interesting. One of the things that is fascinating about arithmetic involving rational numbers is that the addition (and subtraction) operations are in some ways *harder* than the multiplication (and division) operations. Why? Well, for one thing, when you do addition, you need to compute the product of the first rational’s numerator and the second rational’s denominator. Then you add this result to the product of second rational’s numerator and the first rational’s denominator. Once done, you divide this result by the product of the two rationals’ denominators. Of course, having seen children actually get confused by this (including myself), suffice it to say, you’re going to want to test it to believe it—even if you understand it.

So more formally, when we have $\frac{a}{b} + \frac{c}{d}$, addition is achieved by doing $\frac{ad+bc}{bd}$ and then using the normal methods (prayer, guessing factors, or using Euclid’s algorithm) to reduce the fraction to a final result.

Now let's look at it in Scala.

```
def +(that: Rational) =
  new Rational(num * that.den + that.num * den, den * that.den)

def -(that: Rational) =
  new Rational(num * that.den - that.num * den, den * that.den)

def *(that: Rational) =
  new Rational(num * that.num, den * that.den)

def /(that: Rational) =
  new Rational(num * that.den, den * that.num)

def reciprocal() =
  new Rational(den, num)

def negate() =
  new Rational(-num, den)
```

Scala makes this particularly beautiful, given its robust support for *operator overloading* (a concept you may have missed when moving from C++ to Java as we did). In Scala, it's just a matter of defining an operator function (+), which we are defining to work *only* with Rational on the left and right hand side in this example (again, with the intent of not driving to far from Martin Odersky's example).

We won't go into the explanation of every operator. Subtraction (-) is essentially the same as addition (+).

Multiplication of rational numbers is super simple by comparison. Multiply the numerators and denominators of each rational number. Reduce. Division follows similarly, where you basically are multiplying the first fraction by the reciprocal of the second fraction. Reduce. Done.

To support the `Ordered[Rational]` trait, we need to implement comparison. This is shown in the following:

```
def compare(that: Rational) = num * that.den - that.num * den
```

It is interesting to think about how rational numbers are properly compared. It is *essentially* the same as subtraction. The difference, however, is that we really don't need to know the denominator. Why is this? A simple example is helpful. For example, if you had 1/4 and 1/4, subtraction gives 0/16. Of course, we only need the numerator of the subtraction result to know that the two fractions compared equally. More importantly, however, it is about not doing work that doesn't need to be done.

This

```
def compare(that: Rational) = numerator * that.denominator - that.numerator * denominator
```

is way more efficient than

```
def compare(that: Rational) = (this - that).numerator
```

Nevertheless, because it is taking advantage of a bit of mathematical cleverness, there is great value to testing whether compare does what we expect. In our forthcoming JUnit and ScalaTest examples, it will be apparent that we took great care to test this, despite our confidence in the underlying mathematical thinking behind comparison.

Lastly, we have some methods that allow Rational instances to be used properly in Scala collections. We'll keep this simple for now by saying that we define object equality on Rational numbers as follows:

- If a Rational is compared to another non-Rational object, the objects are not equal.
- Two Rationals are equal objects if (and only if) they compare equal using the compare method we just discussed. That is, `this.compare(that) == 0`.

```
override def equals(o: Any) = o match {  
  case that: Rational => compare(that) == 0  
  case _ => false  
}  
  
override def hashCode = (num.hashCode, den.hashCode).hashCode
```

We also provide a definition of `hashCode()` by taking the numerator and denominator and placing them in a Scala tuple and then punting the actual hash code computation to the tuple. Although a simple idea, this little session in `sbt test:console` shows how it works:

```
scala> (1, 4).hashCode  
res1: Int = -1116984814  
scala> (1, 4).hashCode  
res2: Int = -1116984814
```

Two different tuples containing the same integers (numerator and denominator) do, indeed, result in the same hash code! Whew!

1.6 Test-driven development with JUnit and Scala

In this section, we're going to take a look at the first of two general ways of thinking about testing: test driven development, or TDD⁸. In test-driven development, you typically think in terms of the tests you need to write and then write the implementation code expressly with the idea of making the tests pass. (In the interests of disclosure, the author wrote the test code and the implementation code somewhat simultaneously after starting with a spartan implementation of the `Rational` class.) TDD is a valuable way of thinking, because for some classes, it is virtually impossible to envision the entire set of interfaces that are needed. If you look at many class libraries (e.g. `collections`), it is clear that there was often an organic process that led to their creation.

Java programmers nowadays write JUnit tests by creating a test class and using the `@Test` annotation for each intended test method. Each method is written by using the various and sundry assertion methods covered earlier in this chapter.

As the intention of this section is to show how you can take what you already know in Java and apply it immediately to Scala, we are assuming that you have previous experience working with JUnit and Java.

Let's start by looking at the tests for our greatest common divisor method.

```
@Test  
def testMathUtilities(): Unit = {  
  assertEquals(3, gcd(3, 6))  
  assertEquals(3, gcd(-3, 6))  
  assertEquals(-3, gcd(-3, -6))  
  assertEquals(5, gcd(0, 5))  
  assertEquals(5, gcd(5, 0))  
  assertEquals(1, gcd(1, 5))  
  assertEquals(1, gcd(5, 1))  
}
```

Our GCD tests are basically all encoded in one test method, because they're all similar and very closely related. What we're primarily trying to test here is whether various combinations of positive and negative numbers result in the expected behavior. It is worth pondering, however, what exactly is *expected* when it comes to Euclid's algorithm. If you've never looked at it closely, the results may surprise (or scare) you.

Let's start with some easy examples. `gcd(3, 6)` should produce 3. It's easy to see why, because 3 divides into both 3 and 6. You'd expect these two numbers, when they appear in a `Rational` number (fraction) to be reduced to 1 and 2, respectively.

⁸ <http://blog.andolasoft.com/2014/06/rails-things-you-must-know-about-tdd-and-bdd.html>

What about when either or both of the numbers is/are negative? $\text{gcd}(-3, -6)$ should do what? Let's start by thinking about what we *hope* it does. Ideally, we'd like a rational with -3 numerator and -6 denominator to be *reduced* to $1/2$. So we *hope* that the greatest common divisor gives us -3. (Rest assured, it does, but we need to test this. It will only help to cure a bad headache later.)

When either of the numbers is negative (but not both), we expect the greatest common divisor to be positive, though it doesn't really matter in this case, because it will only affect whether the sign is negative in the numerator or the denominator. (Cosmetically, we prefer the sign to be in the numerator when writing fractions, but it doesn't much matter to a computer as long as it works properly when doing arithmetic.

Lastly, we need to look at what happens when zero appears in a greatest common divisor calculation. The last two tests $\text{gcd}(0, 5)$ and $\text{gcd}(5, 0)$ both check that the GCD is 5. Is this what we expect? Yes. If I had a rational number $\frac{0}{5}$, it should be reduced to $\frac{0}{1}$.

So assuming all of our greatest common divisor tests work properly, it becomes much easier to think about whether the rest of our class `Rational`'s methods are working.

```
@Test
def testInitialization(): Unit = {
  val r1 = new Rational(2, 4)
  assertEquals(1, r1.num)
  assertEquals(2, r1.den)

  val r2 = new Rational(-3, 6)
  assertEquals(-1, r2.num)
  assertEquals(2, r2.den)

  val r3 = new Rational(-3, -6)
  assertEquals(1, r3.num)
  assertEquals(2, r3.den)
}

@Test(expected = classOf[ArithmeticException])
def testZeroDenominator(): Unit = {
  val r4 = new Rational(1, 0)
  fail("Zero demoninator was accepted " + r4.den)
}
```

Testing whether rational number arithmetic works as expected.

```
@Test
def testArithmetic(): Unit = {
  val r1 = new Rational(47, 64)
  val r2 = new Rational(-11, 64)

  assert(r1 + r2 == new Rational(36, 64))
  assert(r1 - r2 == new Rational(58, 64))
  assert(r1 * r2 == new Rational(47 * -11, 64 * 64))
  assert(r1 / r2 == new Rational(47, -11))
  assert(r2.reciprocal() == new Rational(64, -11))
  assert(r2.negate() == new Rational(11, 64))
}
```

Testing whether the comparisons involving rational numbers work as expected.

```
@Test
def testComparisons() {
  val r1 = new Rational(-3, 6)
  val r2 = new Rational(2, 4)
```

```
val r3 = new Rational(1, 2)
assert(r1 < r2)
assert(r1 <= r2)
assert(r2 > r1)
assert(r2 >= r1)
assert(r2 == r3)
assert(r2 <= r3)
assert(r3 >= r2)
}
```

1.7 Behavior-Driven Testing using ScalaTest

The following is an example of one of the ScalaTest styles, known as FlatSpec. FlatSpec is an example of *behavior-driven development* or BDD. You write FlatSpec tests with the idea of *describing* the *requirements* and/or expected behavior. As software engineers, behavior-driven development allows us to ensure that the stated requirements match the expected behavior. When we think in terms of BDD as opposed to TDD, we can express our tests at a much higher level, often in plain (English) language. As we consider each of the of the following groups of tests, therefore, we'll go back to what we *said* a rational number should *do*.

One thing we *said* that rational numbers should *do* is to maintain their representation in a reduced form. When combined with other rational numbers (via arithmetic), we also expect this behavior to be observed.

So let's start with looking at how this style applies to the greatest-common divisor itself.

What do we expect of the greatest common divisor algorithm? Well, our friend Euclid had something to say about this.

- When 0 is involved, e.g. $\text{gcd}(x, 0)$ or $\text{gcd}(0, y)$, we expect x or y to be the result.
- When some multiple of a reduced fraction's numerator and denominator are given, we expect that multiple to be the result. A good example is $\frac{1}{3} \cdot \frac{3}{3}$. We'd expect the greatest common divisor to be this multiple.

In a technical sense, this might be more of a pure testing style than behavior driven. However, the way we have expressed this is in terms of how we expect GCD to behave.

As this is the first example of a FlatSpec style test, let's introduce a few things. To create a FlatSpec style test that uses the features we'll be showing in the remaining discussion, you need to put your tests in a Scala class:

```
import org.scalatest._

class RationalScalaTestFlatSpecMatchers extends FlatSpec
  with Matchers {
  // Tests go here
}
```

You then write your tests (behavioral examples):

Each test is written as follows:

```
"Example" should "do something" in {
  // details of what it should do
}
```

or

```
it should "do something" in {
  // details of what it should do
}
```


When you write something like “Example”, the idea is to indicate that we are describing the behavior of a particular aspect under testing. When we use *it*, we are referring to the same aspect but breaking out a separate case. After seeing a few examples, it will become apparent how cool this is.

Back to the actual GCD, here it is:

```
"GCD involving 0" should "give y for gcd(0, y)" in {
  gcd(0, 5) should be(5)
}

it should "give x for gcd(x, 0)" in {
  gcd(0, 5) should be(5)
}

"GCD not involving 0" should "be 3" in {
  gcd(3 * 1, 3 * 3) should be(3)
}

it should "be 5" in {
  gcd(5 * 1, 5 * 5) should be(5)
}
```

As you can see, the code more or less follows the descriptions already given. We basically look at the different cases as written (GCD involving 0 and GCD not involving 0).

Behavior-driven development, as mentioned earlier, tends to eschew (but not prohibit) explicit assertions. So we see this:

instead of:

We think both have value, but it is clear that one has a more *literate* style than the other.

```
"Initializing" should "reduce fractions (+,+)" in {
  val r1 = new Rational(2, 4)
  r1.num should be(1)
  r1.den should be(2)
}

it should "reduce fractions (-,+)" in {
  val r1 = new Rational(-2, 4)
  r1.num should be(-1)
  r1.den should be(2)
}

it should "reduce fractions (-,-)" in {
  val r1 = new Rational(-3, -6)
  r1.num should be(1)
  r1.den should be(2)
}

it should "prohibit zero denominator" in {
  a [ArithmeticException] should be thrownBy {
    new Rational(3, 0)
  }
}
```

Armed with the knowledge the GCD is working, testing initialization is fairly straightforward. We basically want to ensure that initialization of rational instances with any combination of positive and negative denominators results in a reduced fraction.

Todo

We could also ensure that whole numbers represented as fractions have a one in the denominator. We could also ensure that 0 with any denominator reduces to 0/1. Need to add this to the code. Might also be a good exercise for the reader.

One of the interesting tests is to “not allow a zero denominator”. This shows easy it is to test for exceptions in Scala sans the familiar (dreaded?) try/catch syntax found in Java and other languages. The test fails if the `ArithmeticException` (actually, `java.lang.ArithmeticException`) is not successfully intercepted.

The tests of Rational arithmetic are largely what we’d expect.

```
"Arithmetic" should "perform addition" in {
  val r1 = new Rational(47, 64)
  val r2 = new Rational(-11, 64)
  r1 + r2 should == (new Rational(36, 64))
  r1 + r2 should be (new Rational(36, 64))
}

it should "perform subtraction" in {
  val r1 = new Rational(47, 64)
  val r2 = new Rational(-11, 64)
  r1 - r2 should be (new Rational(58, 64))
}

it should "perform multiplication" in {
  val r1 = new Rational(47, 64)
  val r2 = new Rational(-11, 64)
  r1 * r2 should be (new Rational(47 * -11, 64 * 64))
}

it should "perform division" in {
  val r1 = new Rational(47, 64)
  val r2 = new Rational(-11, 64)
  r1 / r2 should be (new Rational(47, -11))
}

it should "perform the reciprocal" in {
  val r1 = new Rational(47, 64)
  r1.reciprocal() should be (new Rational(64, 47))
}

it should "perform negation" in {
  val r1 = new Rational(47, 64)
  val r2 = new Rational(-11, 64)
  r1.negate() should be (new Rational(-47, 64))
}
```

We won’t go through every single one of these as they are largely similar, but let’s take a look at what is possible with the be-matching logic, thanks to our support (in `Rational`) for proper object equality.

Clearly, `r1 + r2` do not result in exactly the same object as `new Rational(36, 64)`. In Scala, as in Java, and many other object-oriented language, it is important to know that equality is not a given. It depends on having defined equality. Because we’ve done this in the implementation of our `Rational` class, we are able to use the be matcher to write the test rather concisely. (In fact, we can use it in any situation where we want to assert equality, but having proper equality really comes in handy for be-matchers in `ScalaTest FlatSpec` style.)

Let’s look at how we test the comparison operations.

```

"Comparisons" should "perform ==" in {
  val r1 = new Rational(2, 4)
  val r2 = new Rational(1, 2)
  info("numeric equality (==) works")
  assert(r1 == r2)
  info("object equality works, e.g. equals()")
  assert(r1.equals(r2))
  info("hashCode() works")
  assert(r1.hashCode() == r2.hashCode())
}

it should "perform <" in {
  val r1 = new Rational(-3, 6)
  val r2 = new Rational(2, 4)
  info("operator < works")
  assert(r1 < r2)
}

it should "perform >" in {
  val r1 = new Rational(-3, 6)
  val r2 = new Rational(2, 4)
  info("operator > works")
  assert(r2 > r1)
}

it should "perform <= for something < and for something =" in {
  val r1 = new Rational(-3, 6)
  val r2 = new Rational(2, 4)
  val r3 = new Rational(1, 2)
  info("operator <= works for < case")
  assert(r1 <= r2)
  info("operator <= works for == case")
  assert(r2 <= r3)
}

it should "perform >= for something > and for something =" in {
  val r1 = new Rational(-3, 6)
  val r2 = new Rational(2, 4)
  val r3 = new Rational(1, 2)
  info("operator >= works for > case")
  assert(r2 >= r1)
  info("operator >= works for == case")
  assert(r2 >= r3)
}

```

For testing equality, we test a number of different dimensions:

- Does == work as expected? We must always be able to compare two rational numbers, even when we are not thinking about object-oriented programming?
- Does object equality, `equals()`, work as expected?
- Does the object hash, `hashCode()`, work as expected?

Knowing the answer to the first is particularly important to scientific programming types (one of us being among them) who want to know whether `Rational` behaves largely like a built-in datatype. Knowing the answer to the second and third questions is important for using `Rational` in non-computational situations, e.g. within a Scala collection (more on that shortly).

This also shows another aspect of how BDD style testing goes *beyond* basic TDD testing. The `info()` method can

be called to show how a test is breaking out different cases (the three cases above, in fact). The setup is largely the same for each, so we don't want to repeat ourselves, because all three of these questions are addressing different ways of looking at equality. Taken as a group, we want all of them to work so equality is meaningful in both a numeric and object-oriented sense.

For the rest of our tests, we expect them to work, because we defined our `Rational` class to extend `Ordered[Rational]`. Nevertheless, a characteristic of good testing is to *test the obvious*. As we did with the equality tests, the `<=` and `>=` are tested to ensure that they work for `<` and `=` (for `<=`) and `>` and `=` (for `>=`).

Recalling the points we made about collections, the following shows how we test whether `Rational` works properly in a collection. We chose a Scala `Set`, because this set uses equality to determine whether or not a member should be included.

```
"Within collections" should "work" in {  
  info("on Set[Rational]")  
  val r1 = new Rational(-3, 6)  
  val r2 = new Rational(2, 4)  
  val r3 = new Rational(1, 2)  
  val s = Set(r1, r2, r3)  
  assert(s.size == 2)  
}
```

If this test works as expected, `Rational(2, 4)` and `Rational(1, 2)` (two different objects, but both of which will have the numerator and denominator by virtue of having been reduced) will result in only one entry being added to the set. The second entry will be `Rational(-3, 6)`. So the cardinality of the resulting `Set[Rational]` should be two (2).

Todo

Add pattern-matching case? I'm still a bit concerned that this is making the example too complicated for a first chapter.

1.8 Discussion

In the interests of following the tradition of Packt books and providing highly hands-on treatment from the beginning, we focused our energy in this chapter on introducing the basics of testing by introducing a fairly well-known OO example, Rational numbers, and elaborating it completely by writing tests in TDD style (using JUnit, a fairly low-level testing framework) and BDD style (using one of the ScalaTest frameworks, known as FlatSpec). While TDD will continue to have a place in your toolset, we expect that you'll find BDD the more compelling alternative, especially when you want to show your customers that you have incorporated direct testing their requirements in your code.

It is worth taking a few moments to consider what sort of things we might want to test in practice. Three rather general questions tend to guide the way we think of testing:

- Does the code work?
- Does the code work well?
- How could the code be improved?

In the general theory of testing, these three questions tend to be informed by some general ideas of software engineering, some of which can get rather theoretical.

1.8.1 Correctness

The first of our three questions definitely falls into this category. When we talk about correctness, we are fundamentally talking about whether the code (algorithms) are correct in a formal sense. While there is still hope that formal

verification systems will eventually be able to prove any piece of code correct, testing is a *pragmatic* approach to correctness. With the work we have put into testing the Rational number class, we are confident that it is correct. We've tested every dimension that a math-inclined person would expect us to test.

But it is interesting to ponder: Have we *exhaustively* tested Rational yet? The answer is firmly in the negative. We've tested various combinations of things that should (and should not) appear in the numerator and denominator. But we haven't thrown random data both to know whether anything breaks. More importantly, we haven't done anything that appears to break performance.

1.8.2 Performance

We all are authors with interests in high-performance computing and related genres (e.g. embedded systems, mobile computing, covered later in this book). One thing our tests in this chapter haven't covered is related to our second question, "Does the code run well?" For something as simple as Rational, we hope the answer is yes, but these tests do nothing to inform us of same.

Performance testing is a huge challenge in general and something beyond the scope of this book to cover in depth. But we still need to know in many situations. If we were implementing a collection class, we'd like to know if the running time of certain methods tracks the known theoretical running time. For example, if we were inserting an item into a (mutable) set, we'd expect the performance to track the running time of the underlying structure. If this were a red/black tree, we'd hope to see time proportional to $O(\log(n))$. We'll look at these aspects later in the book by testing and exploring built-in collection classes, which provide a great case study for performance testing that doesn't become unwieldy or overly domain-specific.

1.8.3 Lifecycle Testing

At present, we are taking a rather limited view of testing in this first chapter. We're very much focused on unit testing and some aspects of acceptance testing. Complex software engineering includes other in-process testing, including integration testing (do all pieces of code work when put together?) to system and acceptance testing. When testing web and mobile applications, you interact with forms that accept data and interact in a black-box way with domain objects. We're going to do a bit of user-interface testing in this book, especially when it comes to Android examples, but there are presentation-level tiers that we're unlikely to cover, owing to the fact that we're not writing those tiers in Scala (e.g. a web app, which is probably using some sort of JavaScript library).

1.8.4 Code Coverage

The third of our general questions about testing is to know "How could the code be improved?" For this question, we may need to look at code coverage tools. This can help us to understand whether the tests we have written really coverage all of the code in our domain. For example, do we honestly know that every single method and line of code in the Rational class has been *touched* by the tests we have written. We don't know that by looking at JUnit or FlatSpec output directly. We'll look at tools that can help us with this later in the book.

1.8.5 Coupling

Test driven development is an important part of incremental development. Because code is continuously changing as new requirements are introduced or refined, refactoring is also an important part of incremental development.

Two important metrics to determine how easy it is to refactor code are afferent and efferent coupling. Afferent coupling describes the number of types that refer to a class. Efferent coupling describes the number of types a class refers to. Stable classes have a high ratio of afferent coupling to efferent coupling. These types of classes are considered 'stable'. A ratio of 10:1 or 15:1 is commonly considered stable. When refactoring, it is simpler to refactor less stable classes and more challenging to refactor more stable classes.

When writing unit tests, be sure to pay attention to how they effect efferent coupling. In our experience we've seen cases where tests will raise the stability from a 20:1 ratio to over a 100:1 ratio. It is our recommendation that if unit tests are introducing excessive efferent coupling that you introduce a factory or builder pattern for your tests to make use of. This will allow you to preserve the production stability of your code and make refactoring and incremental development more productive.

Note: The role of automated testing in the development process (see Fowler) - testing - refactoring - CI/CD

TESTING ENVIRONMENT AND THE SIMPLE BUILD TOOL (SBT)

Todo

Consider evaluating and possibly discussing Typesafe Activator (sbt-based!).

Note: The main reference source for this chapter is <http://www.scala-sbt.org/0.13/docs/>, especially <http://www.scala-sbt.org/0.13/docs/Testing.html>

In this chapter, we'll discuss your choices for setting up an effective development and testing environment for Scala. The main thing to keep in mind is that proper testing almost always involves dependencies on external libraries. Even if you are comfortable working with the Scala command-line tools and a text editor, you are responsible for setting the dreaded classpath. This can quickly become unwieldy even when only simple dependencies are involved, so this is not something you would usually want to do manually.

Therefore, you will benefit greatly from upgrading to Scala's Simple Build Tool (sbt), and the rest of this book relies heavily on this. After switching to sbt, you can continue to use your favorite text editor. And if you prefer to use an integrated development environment (IDE), you are in luck as well: JetBrains IntelliJ IDEA and Eclipse, both very popular IDEs for Scala, integrate well with sbt.

2.1 Brief History of Build Tools

In general, build tools support the build process in several ways:

1. structured representation of the project dependency graph
2. management of the build lifecycle (compile, test, run)
3. management of external dependencies

Some well-known build tools include:

Unix make, Apache ant These earlier tools manage the build lifecycle but not external dependencies.

Apache maven This tool introduced several innovative capabilities. It supports convention over configuration in terms of project layout in the file system and build lifecycle. In addition, it automatically manages external dependencies by downloading them from centralized repositories. It relies on XML-based configuration files.

```
<dependency>
  <groupId>org.restlet</groupId>
  <artifactId>org.restlet.ext.spring</artifactId>
  <version>${restlet.version}</version>
</dependency>
```

Apache ivy, Gradle, sbt, etc. These newer tools emphasize convention over configuration in support of agile development processes. sbt is compatible with ivy and designed primarily for Scala development. For example, ivy uses a structured but lighter-weight format:

```
<dependency org="junit" name="junit" rev="4.11"/>
```

We will focus on sbt in the remainder of this book, though the concepts equally apply to similar build systems and IDEs.

2.2 Installing sbt

The main prerequisite to Scala development is having a Java runtime, version 1.6 or later, installed on your system. We recommend that you install the [latest available Oracle Java 7 Development Kit](#). While you can work with OpenJDK and other VM implementations to run Scala, we believe that the best experience and performance comes from the latest stable release of the Java 7 Platform.

Equally important is having an installation of sbt itself on your system. The exact way of doing so depends on your platform.

- On the Mac, the recommended way is to use Homebrew

```
$ brew install sbt
```

or MacPorts:

```
$ sudo port install sbt
```

- On Linux, the recommended way is to use the installer for your platform available in the sbt setup instructions. For Linux:

```
$ wget https://dl.bintray.com/sbt/native-packages/sbt/0.13.7/sbt-0.13.7.tgz
$ sudo tar xzf sbt-0.13.7.tgz -C /opt
```

- On Windows, there is an MSI (Microsoft Installer) file. Download and install as you normally would any installer on Windows.
- On Windows and Linux, the recommended way is to use the installer for your platform available in the [sbt setup instructions](#).

2.3 Configuring sbt

In the simplest cases, sbt does not require any configuration and will use reasonable defaults. The project layout is the same as the one Maven uses:

- Main application or library code goes in `src/main/scala`.
- Test code goes in `src/test/scala`.

In practice, however, we will want to include some automated testing in the build process, and this typically requires at least one build dependency, as we will see shortly.

sbt supports two configuration styles, one based on a simple Scala-based domain-specific language, and one based on the full Scala language for configuring all aspects of a project.

2.3.1 build.sbt format

A minimal sbt `build.sbt` file looks like this. The empty lines are required, and the file must be placed in the top-level root folder of your project.

```
name := "integration-scala"

version := "0.2"
```

Additional dependencies can be specified either one at a time

```
libraryDependencies += "org.scalatest" %% "scalatest" % "2.2.2" % Test
```

or as a group

```
libraryDependencies += Seq(
  "org.scalatest" %% "scalatest" % "2.2.2" % Test,
  "org.mod4j.org.apache.commons" % "logging" % "1.0.4"
)
```

The dependency format follows the same structure as the Maven example above:

- organization ID
- artifact ID
- version (of the artifact)
- configuration (within the sbt build lifecycle)

In particular, the `Test` configuration indicates that this dependency is required to compile and run the tests for this project but to compile or run the main project code itself.

Furthermore, some dependencies are “cross-built” against different versions of Scala. For example, the `ScalaTest` library comes in the form of two artifacts, `scalatest_2.10` and `scalatest_2.11`, for use with the corresponding versions of Scala. When we use `%%` between organization ID and artifact ID, sbt automatically appends an underscore and the Scala version to the artifact ID.

For example, if our Scala version is 2.10, then

```
"org.scalatest" %% "scalatest" % "2.2.2" % Test
```

is equivalent to

```
"org.scalatest" % "scalatest_2.10" % "2.2.2" % Test
```

This allows us to rely on the default Scala version or indicate our choice in a single place, e.g.:

```
scalaVersion := "2.11.4"
```

2.3.2 Build.scala and multi-project formats

Though you are generally encouraged to use the `build.sbt` format, some complex projects may require build files that use the full Scala syntax. The main build file should be named `Build.scala`. It and other Scala-based build files must be placed in the `project` subfolder of your project root. Further details are available in the [.scala build definition](#) section of the sbt reference manual.

The new [multi-project .sbt build definition](#) format combines the strengths of the other two flavors and is recommended for complex projects instead of the `.scala` flavor.

2.3.3 Finding libraries to depend on

The default place where Maven and its descendents, including sbt, find their dependencies is Maven’s *Central Repository*, which you can search via <http://search.maven.org>. This search interface will allow you to drill down into the desired artifact and ultimately show you exactly what to add to the `libraryDependencies` in your build file.

For example, searching for `scalatest` results in a long list, of which we show only the top.

The screenshot shows the Maven Central Repository search interface. The search bar contains 'scalatest'. Below the search bar, there are links for 'New: About Central', 'Advanced Search', 'API Guide', and 'Help'. The search results are displayed in a table with columns: GroupId, ArtifactId, Latest Version, Updated, and Download. The results show several artifacts, including 'org.scalatest:scalatest', 'org.scalatest:scalatest-maven-plugin', and 'org.scalatest:scalatest_2.10'.

GroupId	ArtifactId	Latest Version	Updated	Download
org.scalatest	scalatest	1.4.RC2 all (14)	26-Apr-2011	pom jar javadoc.jar scaladoc.jar sources.jar test-sources.jar tests.jar
org.scala-tools.testing	scalatest	0.9.5 all (2)	05-Mar-2009	pom jar tests.jar
org.scalatest	scalatest-maven-plugin	1.0 all (5)	03-Jun-2014	pom jar
org.scalatest	scalatest_2.10	3.0.0-SNAP3 all (84)	11-Dec-2014	pom jar javadoc.jar sources.jar
org.scalatest	scalatest_2.11	3.0.0-SNAP3 all (19)	11-Dec-2014	pom jar javadoc.jar sources.jar

Once we drill into the specific artifact `scalatest_2.10`, we see the available versions of this artifact. (The non-cross-built artifact `scalatest` without the added Scala version corresponds to much older versions of this framework.)

The screenshot shows the Maven Central Repository search interface with a more specific search query: 'g:"org.scalatest" AND a:"scalatest_2.10"'. The search results are displayed in a table with columns: GroupId, ArtifactId, Version, Updated, and Download. The results show several versions of the 'org.scalatest:scalatest_2.10' artifact, including '3.0.0-SNAP3', '3.0.0-SNAP2', '3.0.0-SNAP1', '2.2.3-SNAP2', '2.2.3-SNAP1', '2.2.2', '2.2.1', and '2.2.1-M3'.

GroupId	ArtifactId	Version	Updated	Download
org.scalatest	scalatest_2.10	3.0.0-SNAP3	11-Dec-2014	pom jar javadoc.jar sources.jar
org.scalatest	scalatest_2.10	3.0.0-SNAP2	29-Oct-2014	pom jar javadoc.jar sources.jar
org.scalatest	scalatest_2.10	3.0.0-SNAP1	18-Oct-2014	pom jar javadoc.jar sources.jar
org.scalatest	scalatest_2.10	2.2.3-SNAP2	30-Sep-2014	pom jar javadoc.jar sources.jar
org.scalatest	scalatest_2.10	2.2.3-SNAP1	22-Aug-2014	pom jar javadoc.jar sources.jar
org.scalatest	scalatest_2.10	2.2.2	18-Aug-2014	pom jar javadoc.jar sources.jar
org.scalatest	scalatest_2.10	2.2.1	01-Aug-2014	pom jar javadoc.jar sources.jar
org.scalatest	scalatest_2.10	2.2.1-M3	06-Jul-2014	pom jar javadoc.jar sources.jar

Now we can choose the desired version of this artifact. For learning and production development, it is usually best to choose the latest released version, in this case, `2.2.2`. Once we select this version, we can go to the “dependency information” section of the page, select the “Scala SBT” tab, and will see the exact dependency definition we can copy and paste into our build file.

Project InformationGroupId: ArtifactId: Version: **Dependency Information**

Apache Maven

Apache Buildr

Apache Ivy

Groovy Grape

Gradle/Grails

Scala SBT

`"org.scalatest" % "scalatest_2.10" % "2.2.2"`

Leiningen

To use any dependencies not in the central repo, you need to add custom resolvers (preferred) or perform a local install (discouraged).

2.3.4 What Scala test framework are available for Scala?

Concretely, sbt provides a common test interface that these main Scala testing frameworks support directly:

- ScalaTest
- specs2
- ScalaCheck

What this means is that no additional support for the test interface is needed, and you can simply add the desired testing framework(s) as managed dependencies (`libraryDependencies`) in sbt.

If you want to use JUnit or TestNG, however, you will also need to pull in a separate adapter for either of these to work with sbt's common interface. For JUnit, this is a simple additional dependency in the build file:

```
"com.novocode" % "junit-interface" % "0.10" % Test
```

For TestNG, there is an `sbt plugin` that requires a couple of extra steps to add to your project.

In this book, we will start you out with plain JUnit because we assume that most readers are familiar with it. Then we will focus on ScalaTest, which also supports some techniques from ScalaCheck. In the advanced chapter, we will also take advantage of specs2.

2.4 Rational Numbers Recap

Before we discuss how to use sbt in more detail, recap our example from chapter [Testing Fundamentals](#). Our subject under test (SUT) is a Scala class that represents rational numbers. The complete code is available at <https://github.com/LoyolaChicagoCode/scala-tdd-fundamentals>.

The example also includes JRational, a similar class implemented in Java, as a second SUT and various test suites for each of these implementations:

- JavaRationalJUnitTests
- JavaRationalScalaTestFlatSpecMatchers
- RationalJUnitTests
- RationalScalaTestFlatSpecFixtures
- RationalScalaTestFlatSpecMatchers

2.5 Testing with sbt

In this section, we'll continue our sbt explorations with a bit more explanation. As we indicated in chapter 1-Testing Fundamentals, we used sbt with the idea of tying up loose ends in this chapter. This approach works ideally for tools like sbt, which emphasizes convention over configuration, so you can just use sbt without knowing all of the (sometimes gory) details. So please make sure you have checked out the `scala-tdd-fundamentals` repository.

In `ScalaTest`, a test is an atomic unit of testing that is either executed during a particular test run or it is not; a test is usually a method or other program element that stands for a method. A test suite is a collection of zero or more tests. (In `JUnit`, a test class corresponding to a test suite in `ScalaTest`, and a test suite is a collection of test classes.) The sbt testing tasks correspond to this test organization hierarchy. In general, to run one or more sbt tasks `task1`, `task2`, ... `taskN`, we either specify them on the sbt command line in the desired order separated by spaces

```
$ sbt task1 task2 ... taskN
```

or we launch sbt's interactive mode and then enter the tasks one by one

```
$ sbt
...some output...
> task1
...some more output...
> task2
...etc...
```

The following are the most important sbt tasks for testing: The test task This task runs all tests in all available test suites. In our example, it would simply run the six tests (two times three).

```
$ sbt test
[info] Loading global plugins from /Users/lauffer/.sbt/0.13/plugins
[info] Loading project definition from /Users/lauffer/Cloud/Dropbox/lauffer/Work/scala-tdd-examples/scal
[info] Set current project to SimpleTesting (in build file:/Users/lauffer/Cloud/Dropbox/lauffer/Work/s
[info] RationalScalaTestFlatSpecFixtures:
[info] GCD involving 0
[info] - should give y for gcd(0, y)
[info] - should give x for gcd(x, 0)
[info] GCD not involving 0
[info] - should be 3
[info] - should be 5
...more output...
[info] RationalScalaTestFlatSpecMatchers:
[info] GCD involving 0
[info] - should give y for gcd(0, y)
[info] - should give x for gcd(x, 0)
[info] GCD not involving 0
[info] - should be 3
[info] - should be 5
```

```
...lots more output...
[info] ScalaTest
[info] Run completed in 657 milliseconds.
[info] Total number of tests run: 62
[info] Suites: completed 3, aborted 0
[info] Tests: succeeded 62, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[info] Passed: Total 73, Failed 0, Errors 0, Passed 73
[success] Total time: 2 s, completed Mar 6, 2015 4:46:56 PM
```

with the most important information on the third-last line: all tests have passed.

For each ScalaTest-based test suite, we see a heading for that suite and output indicating the result of each test, even when the test has passed. The fourth through sixth lines from the bottom indicate how many ScalaTest suites and tests ran and what results they produced.

For each JUnit-based suite, we only see output if there is a failure or an error. Because all of them passed, the only indication that they ran is the higher total number of tests on the second-last line.

The testOnly task

During development, to save time, we may want to run only a subset of the available tests. The testOnly task allows us to specify zero or more test classes to run. For example, we can run only the test with the square function:

```
$ sbt 'testOnly RationalScalaTestFlatSpecMatchers'
...some output...
[info] Run completed in 445 milliseconds.
[info] Total number of tests run: 22
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 22, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[info] Passed: Total 22, Failed 0, Errors 0, Passed 22
```

This task also supports wildcards, so

```
$ sbt 'testOnly *Fix*'
```

will run only RationalScalaTestFlatSpecFixtures, while

```
$ sbt 'testOnly *Java*'
```

will run both JavaRationalJUnitTests and JavaRationalScalaTestFlatSpecMatchers.

The testQuick task

This task is similar to testOnly in giving you the option to select the matching tests to run. In addition, it runs only those tests that meet at least one of the following conditions: the test failed in the previous run the test has not been run before the tests has one or more transitive dependencies that have been recompiled The test:console task This task allows you to enter an interactive Scala REPL (read-eval-print loop), just like sbt console, but with the test code and its library dependencies for the Test configuration (along with any transitive dependencies) conveniently on the class path. This is useful when you want to explore any code in src/test/scala or the library dependencies for the Test configuration interactively. The test: prefix This prefix is optional for the other tasks we discussed above because their names are unambiguous. There are various other tasks, however, that also apply to the main sources. In those cases, the test: prefix will allow you to disambiguate. For example,

```
$ sbt test:compile
```

will compile the test sources along with the main sources, while

```
$ sbt compile
```

will compile only the main sources. Similarly, if you have a main program in your test sources, you can run it with

```
$ sbt test:run
```

or

```
$ sbt test:runMain
```

2.6 Plugin Ecosystem

sbt includes a rich and growing plugin community-based ecosystem. Plugins extend the capabilities of sbt, and you can install them per project or globally. More details are available in the [sbt reference](#).

In addition to the [sbt-testng-interface](#) mentioned above, here are some useful examples relevant to testing:

- [sbt-scoverage](#): uses Scoverage to produce a test code coverage report
- [sbt-dependency-graph](#): creates a visual representation of library dependency tree
- [ls-sbt](#): browse available libraries on GitHub using ls.implicit.ly
- [sbt-updates](#): checks central repos for dependency updates

2.7 IDE Option: JetBrains IntelliJ IDEA

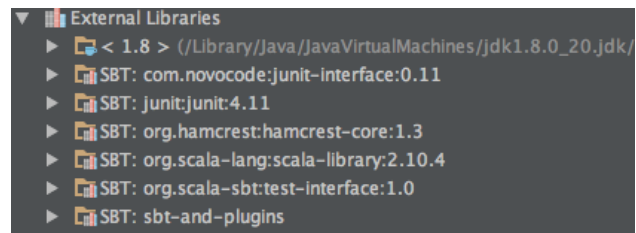
Many developers and students prefer an Integrated Development Environment (IDE) because of code completion and easier code comprehension for complex projects.

Our preferred IDE is IntelliJ IDEA, which has had a lot of traction in the open-source and agile development communities for a long time. You can get the current version of IntelliJ IDEA Community edition for free from the following URL and then install the Scala plugin through the plugin manager.

- <http://www.jetbrains.com/idea/download>

When you install the Scala plugin through the plugin manager, you will automatically get the version that matches that of IDEA. This plugin has become quite mature and usable as of December 2014. In particular, compilation (and execution of Scala worksheets) has become much faster.

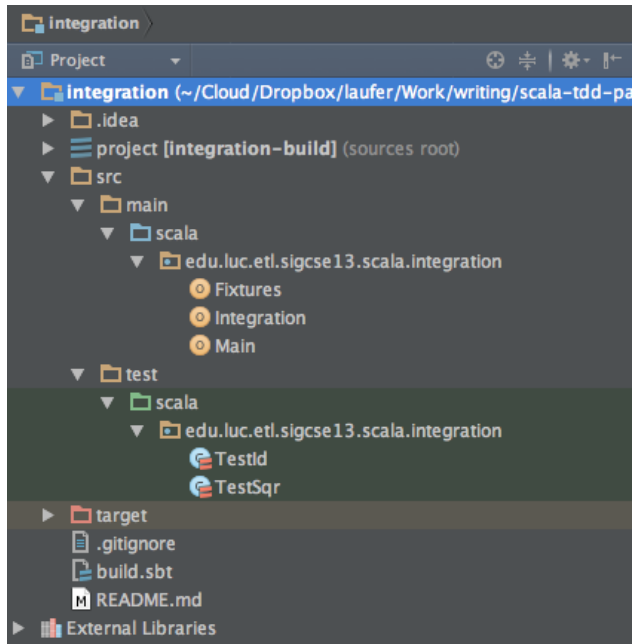
The IntelliJ IDEA Scala plugin also integrates directly with sbt. Instead of *importing* an sbt-based project, you simply *open* it. When you make any changes to the sbt build file(s), IDEA reloads your project and updates the classpath and other IDEA-specific settings accordingly.



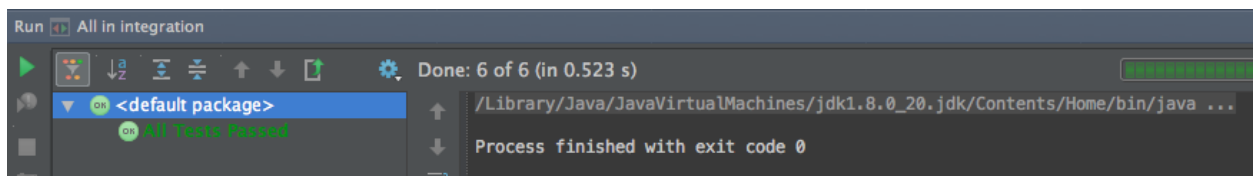
2.7.1 Testing in IntelliJ IDEA

IntelliJ IDEA gives you several options for running tests:

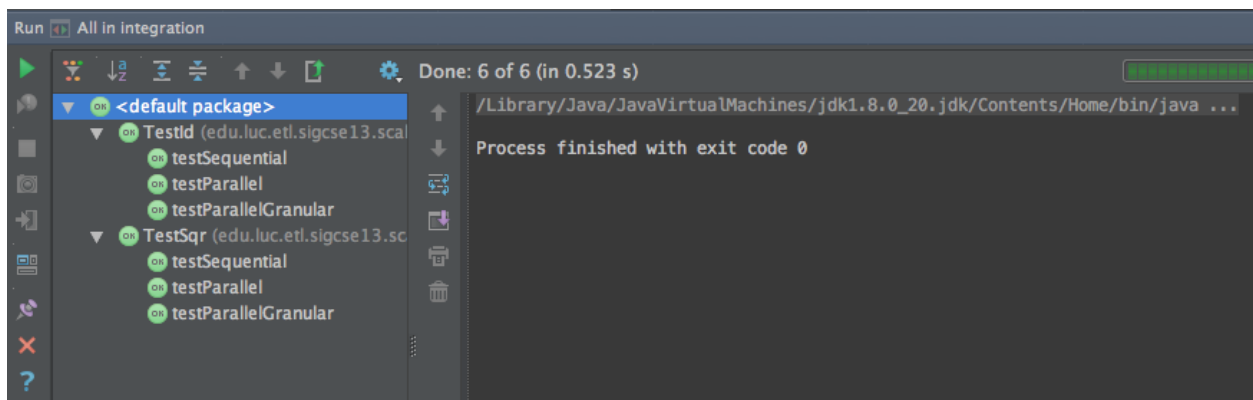
- To run all available tests, you can pop up the context menu (Windows and Linux: right-click, Mac: Control-click) for the project root node or `src/test/scala` and select “Run All Tests”.
- To run an individual test class, pop up the context menu for that test and run it.
- To run two or more specific tests, you can select them, pop up the context menu, and then run them.



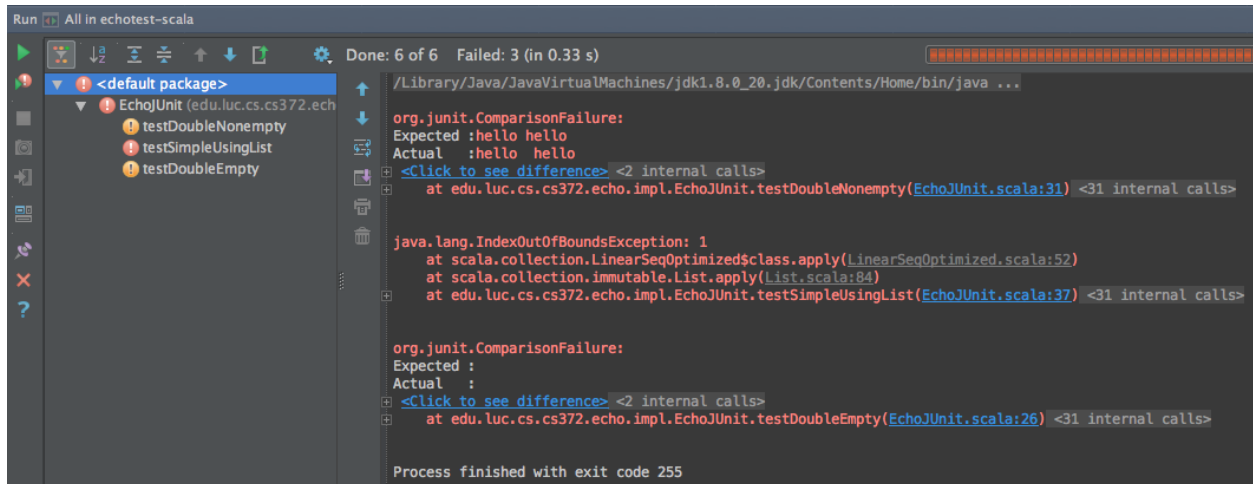
After you run the tests and they all passed, you will usually see a condensed view with the passed tests hidden.



This is because the leftmost button, “hide passed”, is enabled by default. You can turn this option off and drill into the tests.



Also, failed tests automatically show up in expanded fashion.



On these images, we recognize the three possible outcomes of a test from the fundamentals chapter [REF]:

- pass: green circle with the word “OK”
- fail: orange circle with an exclamation mark
- error: red circle with an exclamation mark

2.7.2 Tips

- IntelliJ IDEA has a built-in native terminal for your OS. This allows you to use, say, hg or sbt conveniently without leaving IDEA.

View > Tool Windows > Terminal

- To practice Scala in a light-weight, exploratory way, you can use Scala worksheets in IntelliJ IDEA. These will give you an interactive, console-like environment, but your work is saved and can be put under version control.

File > New > Scala Worksheet

You can even make your worksheets test-driven by sprinkling assertions throughout them.

2.8 IDE Option: Eclipse Scala IDE

The official Scala IDE is provided as an Eclipse bundle that has Scala already installed, based on the current Luna release. It will work on all platforms with very minor differences and provides similar functionality to IntelliJ IDEA. The following link will take you there.

<http://scala-ide.org/download/sdk.html>

If you are already using Eclipse, you can add the Scala IDE as a plugin to your existing installation. The steps for this are described here.

<http://scala-ide.org/docs/current-user-doc/gettingstarted/index.html>

Perhaps the key difference between IntelliJ IDEA and Eclipse from a Scala developer’s point of view is the support for sbt. While IntelliJ IDEA can directly open sbt-based projects in most cases, Eclipse requires you to first generate an Eclipse project from sbt on the command line and then import this into Eclipse as an existing project. Importing an sbt project into Eclipse To enable sbt to generate an Eclipse project definition, we add the sbteclipse plugin to your project by placing the following line into our project/plugins.sbt file (or installing it globally in \$HOME/.sbt/0.13/plugins/build.sbt, where \$HOME stands for your home directory.)


```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "3.0.0")
```

This defines the sbt task `eclipse`, which we can use as follows:

```
$ sbt eclipse [info] Loading global plugins from /Users/lauffer/.sbt/0.13/plugins [info] Updating
{file:/Users/lauffer/.sbt/0.13/plugins/}global-plugins... [info] Loading project definition from
/Users/lauffer/Cloud/Dropbox/lauffer/Work/scala-tdd-examples/scala-tdd-fundamentals/project [info] Updat-
ing {file:/Users/lauffer/Cloud/Dropbox/lauffer/Work/scala-tdd-examples/scala-tdd-fundamentals/project/}scala-
tdd-fundamentals-build... [info] Set current project to SimpleTesting (in build
file:/Users/lauffer/Cloud/Dropbox/lauffer/Work/scala-tdd-examples/scala-tdd-fundamentals/) [info] About to cre-
ate Eclipse project files for your project(s). ... [info] Done updating. [info] Successfully created Eclipse project files
for project(s): [info] SimpleTesting
```

We are now ready to import the project into Eclipse in three steps. We first choose `File > Import` from the top-level menu and tell Eclipse that we want to import an existing project. We then navigate to the root folder of our project. Finally, we confirm the import, making sure our desired existing project shows up in the list and has been checked off.

You should now see your project tree in the Eclipse Package Explorer view to the left. In particular, the Referenced Libraries section will include the library dependencies from your `build.sbt` along with their direct and indirect dependencies (transitive closure).

TIP: To make sure you always have a working command-line configuration of your project, you should consider sbt's `build.sbt` the "single source of truth" and make any configuration changes there. Then, every time you do make a change, re-run `sbt eclipse` and then refresh your project within Eclipse to pick up the changes. For example, if you added `scalacheck` as a library dependency, it should now show up in Eclipse's Package Explorer view under Referenced Libraries.

Testing in Eclipse We can now run all JUnit-based test suites through the top-level menu

```
Run > Run As... > Scala JUnit Test
```

or the desired sub-hierarchy of our test through context menu (obtained by right-clicking on the desired node in the Package Explorer)

```
Run As... > Scala JUnit Test
```

The test(s) then run in the graphical Eclipse JUnit test runner.

To run `ScalaTest`-based test suites, we have to run them individually through the context menu

```
Run As... > ScalaTest - File
```

The selected test suite will then run in the graphical Eclipse `ScalaTest` runner. Initially, Eclipse will show you the `ScalaTest` console.

Once you switch to the `ScalaTest` tab, you will see the graphical `ScalaTest` runner displaying a hierarchical view of your test suite.

TIP: Eclipse is pickier than sbt or IntelliJ IDEA about requiring the source folder hierarchy under test to match the logical package structure. If they don't match, then Eclipse will not discover the JUnit tests therein.

IDE Choice: `Typesafe Activator` `Typesafe Activator` is a lightweight IDE implemented as a browser-based front end to a local installation of sbt provided by Typesafe, the company behind Scala and related technologies. You can obtain `Activator` from here:

<https://typesafe.com/get-started>

For our needs, the "mini package" with no bundled dependencies is actually sufficient.

When you launch `Activator` from your file explorer or command line, the `Activator` home screen will show up in your default browser. From there, you can navigate to an existing sbt-based project using the navigation view on the right.

TIP: Your sbt-based project must contain a `project/build.properties` file containing at least the single line

```
sbt.version=0.13.7
```

Otherwise Activator will not recognize your project as valid.

You can now use the menu in the far left column to edit, run, and test your code. There is a file navigation menu on the near left and a syntax-directed editor as the main content of the window.

Once you run the tests, you will see a flat list of results. Failures and/or errors will show up as a problem count in the far left column and within the flat list. So far, there is no navigation from the flat list directly to the source editor.

2.9 IDE Choice: Codio Instant Cloud-Based Environment

Todo

Konstantin needs to help me get all of the figures into the remaining sections.

Codio is an entirely cloud-based hosted environment and enables you to develop without installing any software locally. The free starter tier provides modest but sufficient computational resources and unlimited public and/or private repositories. You can access Codio here:

<https://codio.com>

From your personal Codio home screen, you can start creating a project. The first thing you will need to do is choose a suitable solution stack for Scala development:

Then you can create a new empty project or, as we are showing here, import an existing one from any Git repository.

Using the “Filetree” view on the left, you can navigate within your source files and open them in the syntax-directed editor.

To run or test your code, you will use Tools > Terminal to open a Linux terminal where you can run `sbt test` or other commands.

At the time of this writing, Codio provides you with a persistent virtual Linux environment based on the Ubuntu 12.04.5 LTS (long-term support) release. This environment does not give you root access or access to the standard package manager, `apt`. Instead, it provides you with Codio’s own package manager called `parts`, which lets you add packages from a much more restricted list maintained by Codio.

Codio has emerged as our preferred and recommended way to get started with general Scala console app and web app/service development because it requires no local installation yet is fully sbt-based and integrates with Git. In addition, we have been able to create a custom Codio stack for Android development in Scala as well as Java. Summary In this chapter, we discussed how to test Scala code in various ways. We first covered command-line testing using the simple build tool (sbt) and its test-specific tasks. We then covered several popular and emerging IDE choices, including Codio, our recommended choice to get started in the cloud without any local installation.

BASIC TESTING

- almost like a meta-framework that supports a broad range of testing styles
- great way to get started with Scala as a team
- nonfunctional code, similar to writing build scripts in, say, Groovy (as in Gradle)
- agile way to do Java!
- status of [ScalaTest shell](<http://www.artima.com/weblogs/viewpost.jsp?thread=326389>)?

3.1 A Basic Example: Rational Arithmetic

In the previous chapter, we discussed a *guiding example* (Rational numbers) that was written in Scala and tested using JUnit (Java implementation of xUnit) and ScalaTest. This begs an interesting question:

Can we actually write a Java based version of the same class and test it using Scala?

Of course, the answer is an emphatic yes. By now, it should be apparent that Scala and Java can, in fact, be used together or apart. In our own work, we rely extensively on the ability to make use of existing Java frameworks/libraries, some of which may not be given the Scala touch for awhile.

3.1.1 JRational - the Java version of Rational

We're mostly going to present this class *as is* with the following two assumptions:

- You're already familiar with Java and have working knowledge of it.
- You can easily go back to chapter one to understand the details.

We'll cover the key differences, which will probably make you love Scala more than Java after reading this chapter. We're going to present each of the fragments of code, which have been commented so you can see how the Scala and Java versions differ.

Let's start with the GCD implementation:

```
public static int gcd(int x, int y) {  
    if (x == 0)  
        return y;  
    else if (x < 0)  
        return gcd(-x, y);  
    else if (y < 0)  
        return -gcd(x, -y);  
    else  
        return gcd(y % x, x);  
}
```

Here we have a static class method as opposed to a standalone Scala def. The logic for computing the GCD is identical to the Scala version.

Initialization (construction) is similar to the Scala version. The only difference is that in Java all of the initialization code goes into a constructor method. In Scala, you initialize instances outside of any method. (There are no constructors.)

```
public JRational(int initialNumerator, int initialDenominator) throws ArithmeticException {
    g = gcd(initialNumerator, initialDenominator);
    numerator = initialNumerator / g;
    denominator = initialDenominator / g;
    testQuotient = initialNumerator / initialDenominator;
}
```

Arithmetic is where we see a few differences, owing to Java's lack of support for operator overloading. So where you previously saw `+`, `-`, `*`, and `/`, here you see `add()`, `subtract()`, `multiply()`, and `divide()`.

```
public JRational add(JRational that) {
    return new JRational(numerator * that.denominator + that.numerator * denominator,
        denominator * that.denominator);
}

public JRational subtract(JRational that) {
    return new JRational(numerator * that.denominator - that.numerator * denominator,
        denominator * that.denominator);
}

public JRational multiply(JRational that) {
    return new JRational(numerator * that.numerator, denominator * that.denominator);
}

public JRational divide(JRational that) {
    return new JRational(numerator * that.denominator, denominator * that.numerator);
}

public JRational reciprocal() {
    return new JRational(denominator, numerator);
}

public JRational negate() {
    return new JRational(-numerator, denominator);
}
```

The comparison logic is similar to the Scala version.

```
@Override
public int compareTo(JRational that) {
    return numerator * that.denominator - that.numerator * denominator;
}
```

To make comparison work in Java, we implement `Comparable<JRational>` and provide a definition that is virtually identical to the Scala version.

Lastly, and similar to our intentions in the Scala version, we provide methods for allowing JRational instances to work properly in object containers, e.g. Java Collections.

```

@Override
public int hashCode() {
    int[] pair = { numerator, denominator};
    return Arrays.hashCode(pair);
}

@Override
public boolean equals(Object that) {
    if (that instanceof JRational) {
        return compareTo( (JRational) that) == 0;
    } else
        return false;
}

@Override
public String toString() {
    return "Rational(" + numerator + "/" + denominator + ";" + (numerator * g) + "/" + (denominator * g) + ")";
}

```

Object equality, e.g. `equals()`, basically requires us to use `isinstanceof` instead of a Scala pattern match expression.

For computing `hashCode()`, we simulate a Scala tuple by putting the numerator and denominator into an array and using the convenient utility `Arrays.hashCode()` to compute the hash value based on the content of the `int[]` array.

We wanted to be sure we believed it, so we fired up the `sbt console` to check `hashCode()` interactively.

```

scala> import scalatddpackt.JRational
import scalatddpackt.JRational

scala> val r1 = new JRational(1, 2)
r1: scalatddpackt.JRational = scalatddpackt.JRational@3e2

scala> r1.hashCode
res0: Int = 994

scala> val r2 = new JRational(3, 6)
r2: scalatddpackt.JRational = scalatddpackt.JRational@3e2

scala> r2.hashCode
res1: Int = 994

scala> r1.equals(r2)
res3: Boolean = true

```

Sure enough, it works as expected. $\frac{1}{2}$ and $\frac{3}{6}$ are indeed equal and have the same hash codes.

3.2 Scala + JUnit to Test Java Rational Class

Here are the reworked versions to test our Java Rational (`JRational`) using JUnit and Scala!

Aside from a few differences, the tests look virtually identical to how we test the Scala Rational.

```

@Test
def testInitialization(): Unit = {
    val r1 = new JRational(2, 4)
    assertEquals(1, r1.getN)
}

```

```
assertEquals(2, r1.getD)

val r2 = new JRational(-3, 6)
assertEquals(-1, r2.getN)
assertEquals(2, r2.getD)

val r3 = new JRational(-3, -6)
assertEquals(1, r3.getN)
assertEquals(2, r3.getD)
}
```

One key difference is related to Java style vs. Scala style. For our mathematical utility, `gcd()`, which is a Java class (static) method, we cannot make a reference to the static member. Luckily, we can make use of these members by using a little Scala magic to import the method for use:

```
import JRational.gcd
```

Once done, our JUnit code to test `gcd()` is identical to the Scala version.

```
@Test
def testInitialization(): Unit = {
  val r1 = new JRational(2, 4)
  assertEquals(1, r1.getN)
  assertEquals(2, r1.getD)

  val r2 = new JRational(-3, 6)
  assertEquals(-1, r2.getN)
  assertEquals(2, r2.getD)

  val r3 = new JRational(-3, -6)
  assertEquals(1, r3.getN)
  assertEquals(2, r3.getD)
}
```

For testing initialization, our code is identical with the following notable difference: accessing private state in the Java class requires some magic. We introduce *getters* to access the numerator via `getN()` and denominator via `getD()`.

When working with Java classes, only the *public* members can be accessed within Scala.

Arithmetic works largely as expected with the only notable difference being that the Java version doesn't support operators. So we need to turn infix expressions into the less-palatable form.

That is, each occurrence of

```
r1 + r2
r1 - r2
r1 * r2
r1 / r2
```

becomes

```
r1.add(r2)
r1.subtract(r2)
r1.multiply(r2)
r1.divide(r2)
```

The unary operators `reciprocal()` and `negate()` are the same in both of our implementations (that is, they don't define an operator).

```
@Test
def testArithmetic(): Unit = {
  val r1 = new JRational(47, 64)
  val r2 = new JRational(-11, 64)

  assert(r1.add(r2) == new JRational(36, 64))
  assert(r1.subtract(r2) == new JRational(58, 64))
  assert(r1.multiply(r2) == new JRational(47 * -11, 64 * 64))
  assert(r1.divide(r2) == new JRational(47, -11))
  assert(r2.reciprocal() == new JRational(64, -11))
  assert(r2.negate() == new JRational(11, 64))
}
```

As for testing comparisons, our code simply needs to test what `compareTo()` does. Given that there's no operator overloading in Java, we don't need to test each of the operators and various combinations involving equality.

```
@Test
def testComparisons() {
  val r1 = new JRational(-3, 6)
  val r2 = new JRational(2, 4)
  val r3 = new JRational(1, 2)
  assert(r1.compareTo(r2) < 0)
  assert(r2.compareTo(r1) > 0)
  assert(r2.compareTo(r3) == 0)
}
```

Roadmap for the rest of this chapter

- A different example (so readers don't get bored)
- `ScalaTest` and `JRational`
- More `ScalaTest` (looking at other `ScalaTest` styles)
- `ScalaTest` console?

MOCKING

A Mockito example is at <https://bitbucket.org/lucoodevcourse/shapes-android-scala-solution/src/default/src/androidTest/scala/ui/DrawTest.scala>

Joe comments that we need to talk about the good and not-so-good uses of mocking.

ADVANCED

Details to follow.

CONTINUOUS INTEGRATION

In this chapter, we will discuss your choices for establishing an effective continuous integration system for your Scala Software. We will explore how to best configure a continuous integration server for two different continuous integration products. We will also explore best practices for writing unit tests to minimize false positives in your automated testing.

6.1 Continuous Integration Products

The two products we will present in this chapter are JetBrains™, TeamCity™ and Jenkins. TeamCity™ is developed by the same company that develops IntelliJ IDEA and other products in the Java eco-system. Jenkins is an award winning product that has a long history with the Java community. Both are good options for implementing a continuous integration system.

6.2 Team City

6.2.1 Installation

First you will need to install TeamCity™. The following instructions are for Ubuntu. You will need to install the OpenJDK package from the ubuntu package manager as a prerequisite.

```
$ wget http://download.jetbrains.com/teamcity/TeamCity-9.0.2.tar.gz
$ tar -xzf TeamCity-9.0.2.tar.gz
```

Then, to configure TeamCity™ to launch on startup, you can save the following script into the /etc/init.d folder with the file name `teamcity`. This script assumes you are going to run the server under the user account also named `teamcity`:

```
#!/bin/bash
case $1 in
start)
    cd /home/teamcity/TeamCity/bin
    su teamcity -c "./teamcity-server.sh start"
;;
stop)
    PID=`ps aux | gawk '{printf("%s %s\n", $1, $2);}' | grep -i "teamcity" | gawk '{printf("%s\n", $1);}'`
    kill $PID
;;
esac
```

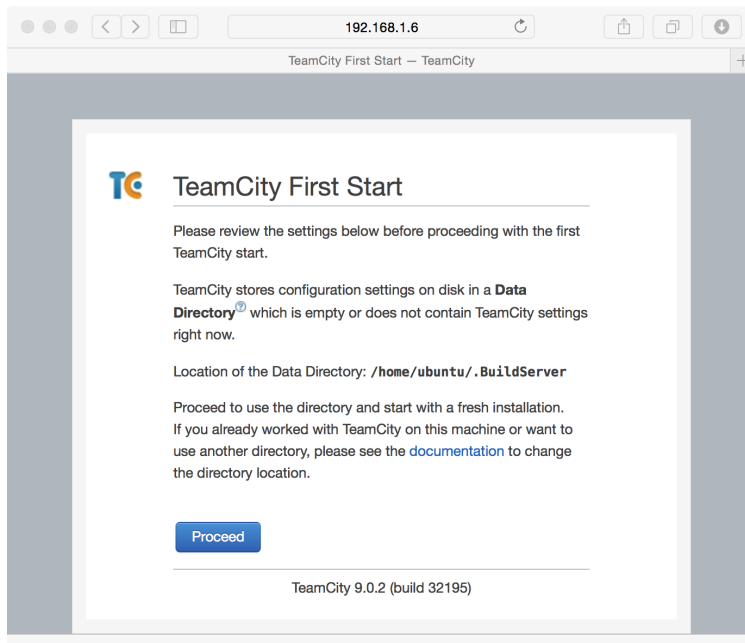
After creating this script, you can run the following command to configure Ubuntu to start TeamCity™ on startup.

```
$ sudo chmod +x /etc/init.d/teamcity
$ sudo update-rc.d teamcity defaults
```

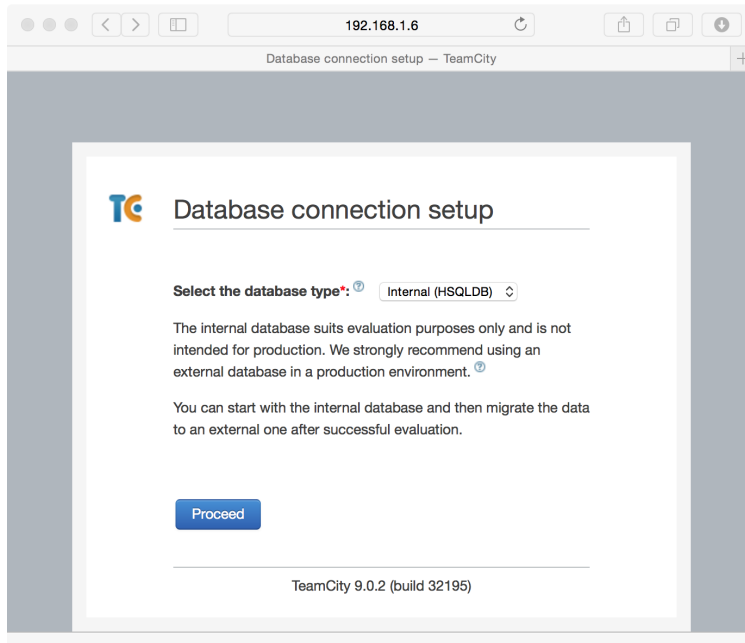
Then, after rebooting the server, you should have a working installation of TeamCity™. You can access the server at <http://server:8111/>

6.2.2 Initial Configuration

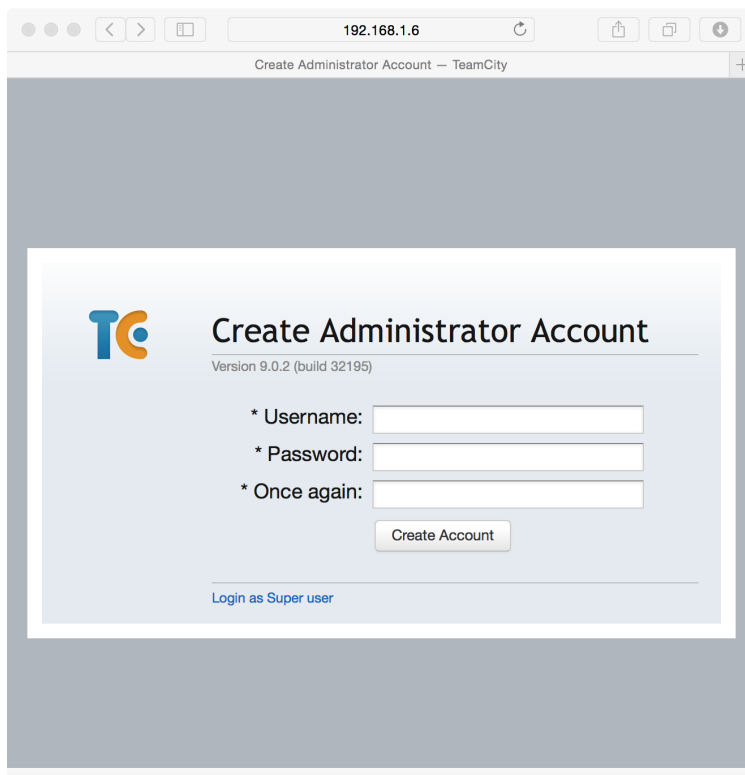
After the initial installation, you will need to perform some configuration for security and the TeamCity™ server database. Start by directing your browser to your TeamCity™ installation. You will be first asked for where your local data directory will be stored. This will be a folder where configuration settings will be stored and should be regularly backed up.



On the next page, you will select your database provider. If your site only has a few developers, HSQLDB will suffice, but if you are planning on supporting more than 10 developers, one of the other database providers is recommended.



After you select your provider, you will have to accept the TeamCity™ license agreement. On the following page, you will be able to create an Administrator account for TeamCity™.



6.2.3 Installing a Build Agent

To install a build agent on Ubuntu, you will need to have the Java runtime and Scala SBT installed as prerequisites. The following code snippet shows how to download and install a TeamCity™ build agent.

```
$ mkdir TeamCityBuildAgent
$ cd TeamCityBuildAgent
$ wget http://server:8111/update/buildAgent.zip
$ unzip buildAgent.zip
$ cd bin
$ chmod +x install.sh
$ ./install.sh http://server:8111
```

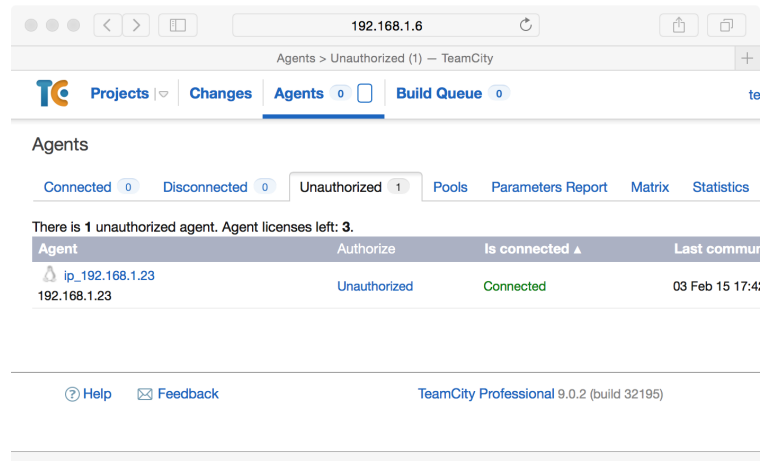
To enable the build agent on startup, you can create the following script under `/etc/init.d/teamcity-buildagent`:

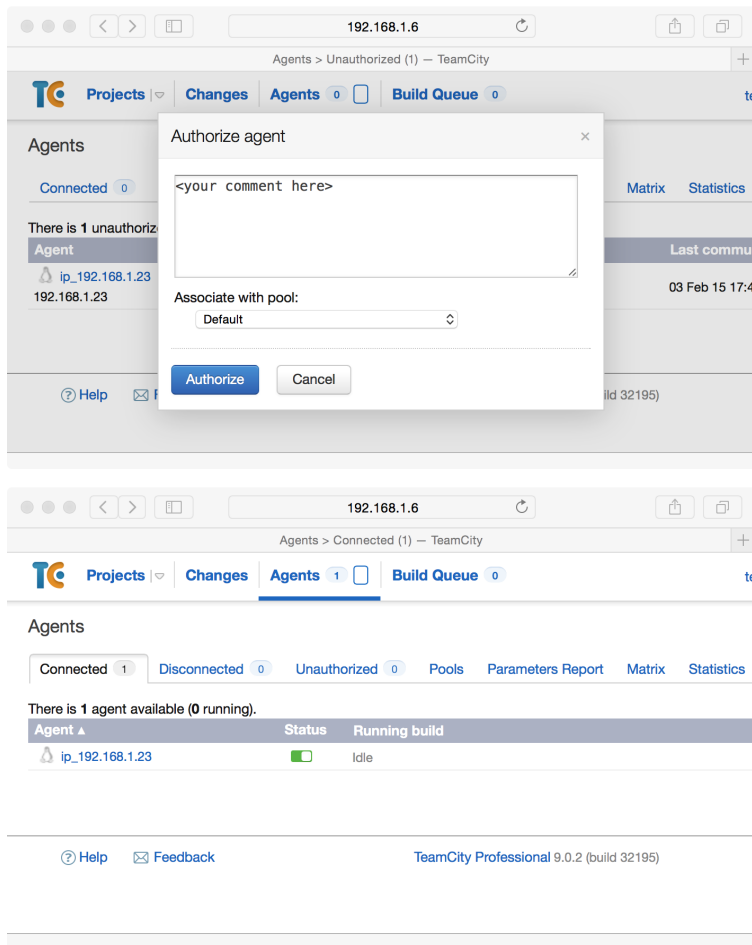
```
#!/bin/bash
case $1 in
start)
    cd /home/teamcity/TeamCityBuildAgent/bin
    su teamcity -c "./agent.sh start"
;;
stop)
    PID=`ps aux | gawk '{printf("%s %s\n", $1, $2);}' | grep -i "teamcity" | gawk '{printf("%s\n", $1);}'`
    kill $PID
;;
esac
```

After creating the script, you will need to refresh your system services:

```
$ sudo chmod +x /etc/init.d/teamcity-buildagent
$ sudo update-rc.d teamcity-buildagent defaults
```

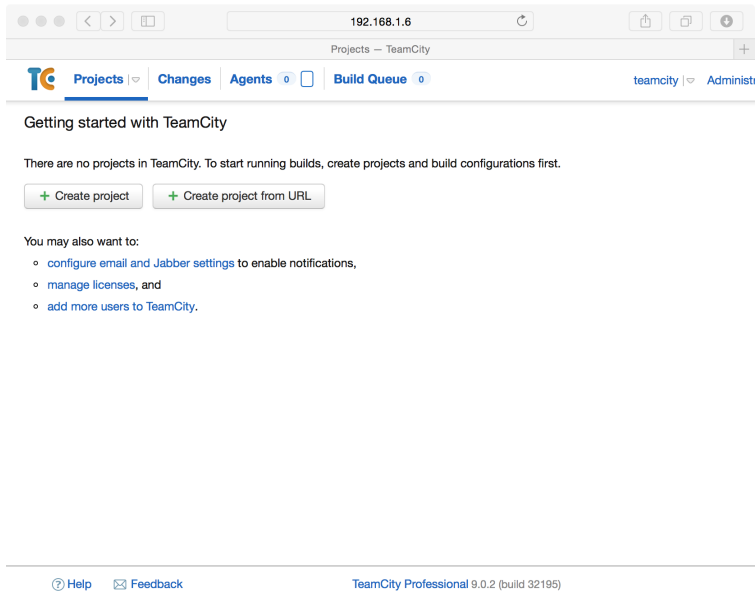
Once the build agent is up and running, you will need to authorize it through the TeamCity™ site. The agent should show up under the Agents tab and Unauthorized tab. To authorize it, click on `unauthorize` and then click `authorize`. After a few seconds, the build agent should show up under the connected tab.



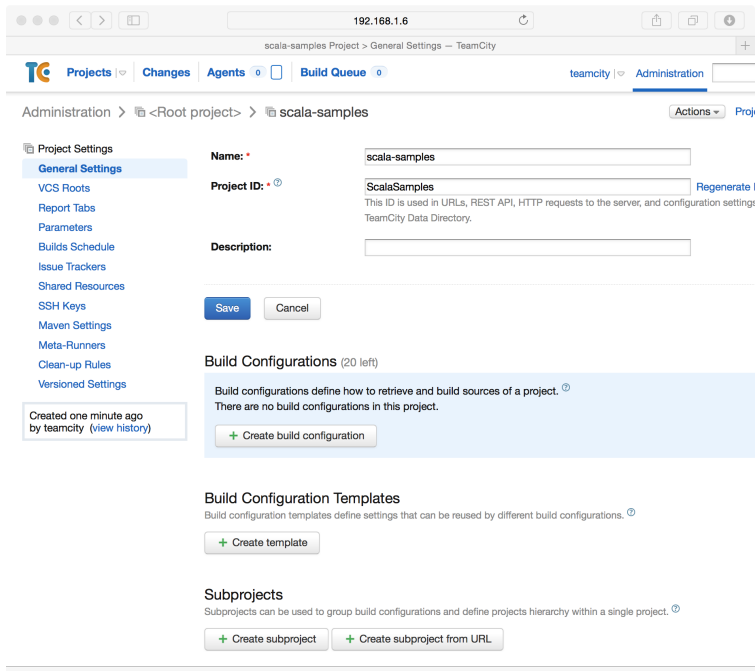


6.2.4 Creating a Project and Build Configuration

Now you should be able to create projects in TeamCity™. TeamCity™ projects are organizational concepts that allow you to reuse version control connections, and to control more fine grained permissions. Every build configuration belongs to a project. To create your first project, you will need to log into your TeamCity™ server and click `Create project`.



After creating the project, you will be able to create a build configuration. The free version of TeamCity™ allows for a maximum of 20 build configurations. To create the build configuration, you will click the `Create build configuration` button.



Create Build Configuration — TeamCity

Administration > <Root project> > scala-samples > Create Build Configuration

Name:

Build configuration ID:
This ID is used in URLs, REST API, HTTP requests to the server, and configuration settings in the TeamCity Data Directory.

Description:

[Help](#) [Feedback](#) TeamCity Professional 9.0.2 (build 32195) [License agreeer](#)

After creating the build configuration, you will be asked to supply the version control information for your project. TeamCity™ supports many of the version control systems available today. Here we will use the book’s sample code which is available on Git Hub

New VCS Root — TeamCity

Administration > <Root project> > scala-samples > Version Control Settings > New VCS Root

Build configuration successfully created. You can now configure VCS roots.

Type of VCS

Type of VCS:

Repository URL and Authentication

Repository URL:
A VCS repository URL. Supported formats: [http\(s\)://](#), [svn://](#), [ssh://git@](#), [git://](#), etc. as well as URLs in Maven format. [?](#)

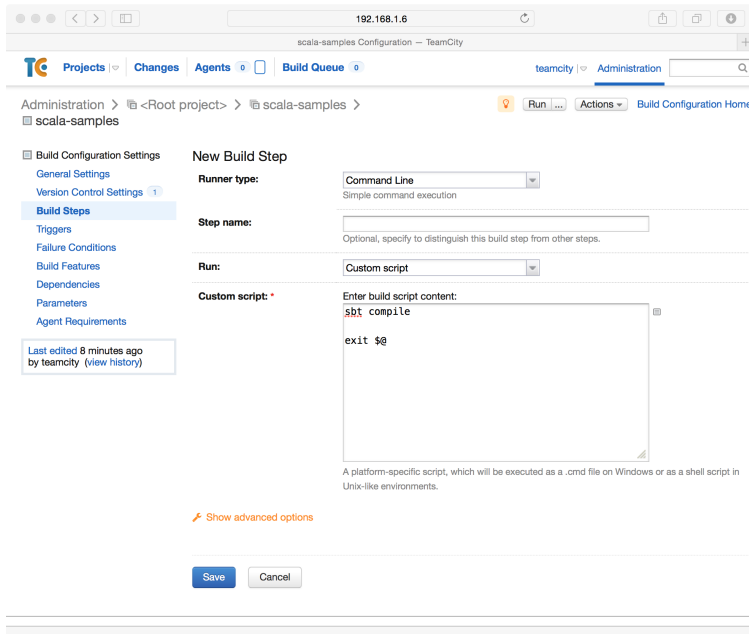
Username:
Provide username if access to the repository requires authentication

Password:
Provide password if access to the repository requires authentication

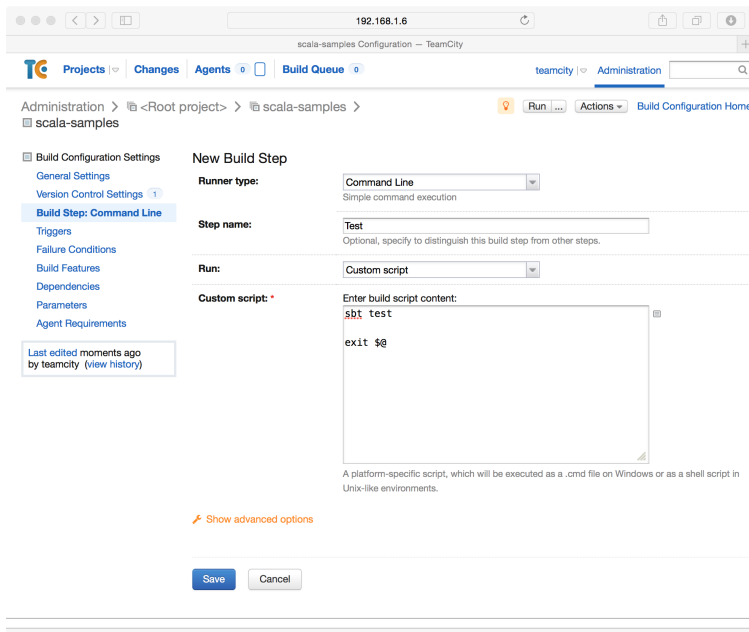
[Show advanced options](#)

[Help](#) [Feedback](#) TeamCity Professional 9.0.2 (build 32195) [License agreeer](#)

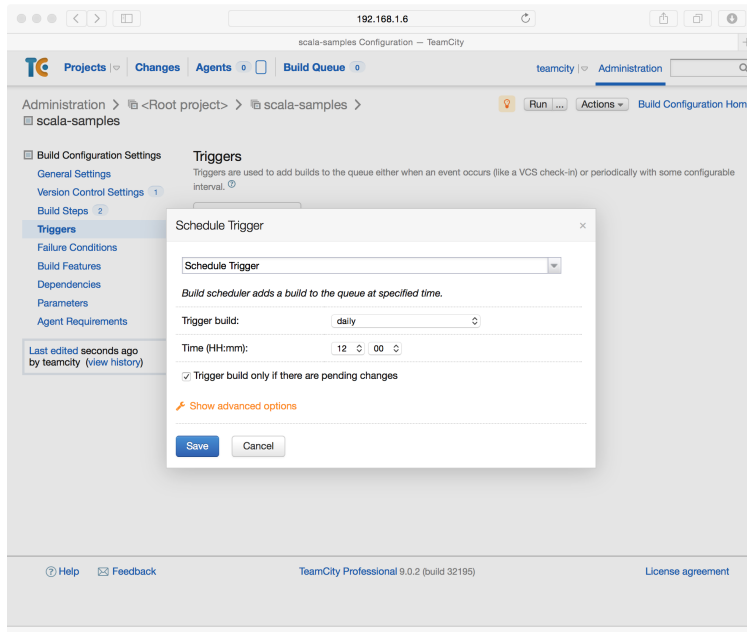
The next step will be to create the build script for the project. This script is fairly simple. The exit statement at the end of the script is there to communicate back to the build server whether the previous call to `sbtc` succeeded or not.



After creating the compilation step, you will next create the step to run all of your unit tests. This script is also very straight forward:



The final step will be to create build triggers. These can be added to trigger builds every time a change is checked into version control or can be configured to build periodically. Later in this chapter, we will discuss how to choose between these two.



Todo

agent configuration

6.3 Jenkins

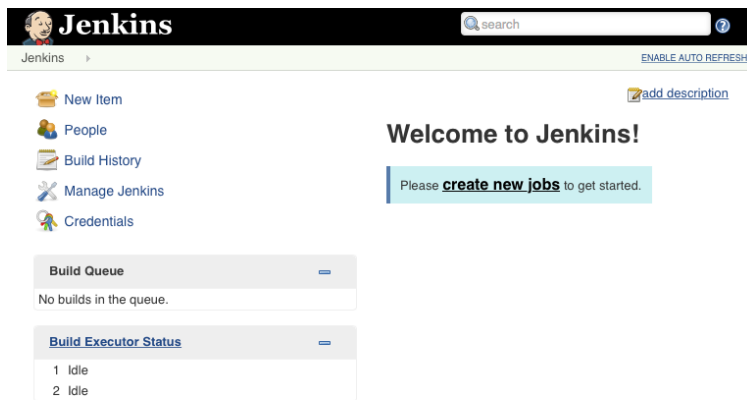
6.3.1 Installation

First you will need to install Jenkins. The following instructions are for Ubuntu. The first step is to install the Jenkins package.

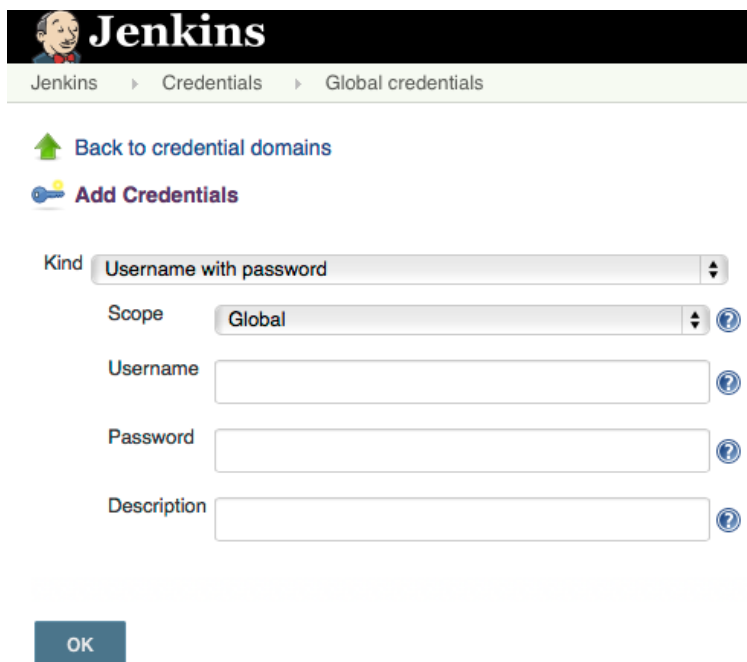
```
$ wget -q -O - https://jenkins-ci.org/debian/jenkins-ci.org.key | sudo apt-key add -
$ sudo sh -c 'echo deb http://pkg.jenkins-ci.org/debian binary/' > /etc/apt/sources.list.d/jenkins.l
$ sudo apt-get update
$ sudo apt-get install jenkins
```

6.3.2 Configuring Security

After this step you should be able to log into your jenkins server. Next we will add a user that can log into Jenkins. To do this, you will open the Jenkins menu and click on Credentials. On the next screen you will select Global Credentials and then add credentials.

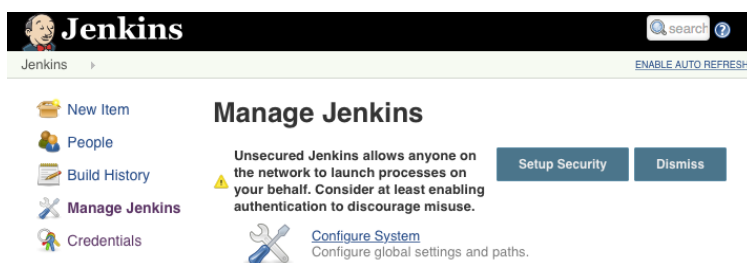


The Jenkins Welcome Screen features a dark header with the Jenkins logo and a search bar. Below the header, a sidebar on the left contains links for 'New Item', 'People', 'Build History', 'Manage Jenkins', and 'Credentials'. The main content area displays 'Welcome to Jenkins!' with a message to 'Please create new jobs to get started.' and a button to 'add description'. At the bottom, there are two expandable sections: 'Build Queue' (showing 'No builds in the queue.') and 'Build Executor Status' (showing two 'Idle' executors).



The 'Add Credentials' form in Jenkins has a dark header with the Jenkins logo. Below the header, a breadcrumb trail shows 'Jenkins > Credentials > Global credentials'. A green arrow points to a 'Back to credential domains' link. The form itself is titled 'Add Credentials' and includes a 'Kind' dropdown set to 'Username with password', a 'Scope' dropdown set to 'Global', and input fields for 'Username', 'Password', and 'Description'. Each input field has a help icon. An 'OK' button is located at the bottom left of the form.

Next you will have to setup security. To do this, click on the Jenkins menu and click Manage Jenkins. You will be presented with a screen where you can click Setup Security. On this screen, for demo purposes we will select Logged-in users can do anything and Jenkins' own user database



The 'Manage Jenkins' screen in Jenkins has a dark header with the Jenkins logo. Below the header, a sidebar on the left contains links for 'New Item', 'People', 'Build History', 'Manage Jenkins', and 'Credentials'. The main content area displays 'Manage Jenkins' with a warning message: 'Unsecured Jenkins allows anyone on the network to launch processes on your behalf. Consider at least enabling authentication to discourage misuse.' Below the warning are two buttons: 'Setup Security' and 'Dismiss'. At the bottom, there is a 'Configure System' link with a wrench icon and the text 'Configure global settings and paths.'

Jenkins

Jenkins > Configure Global Security

Configure Global Security

☒ Enable security

TCP port for JNLP slave agents ☐ Fixed : ☒ Random ☐ Disable

Disable remember me ☐

Access Control

Security Realm

☐ Delegate to servlet container

☒ Jenkins' own user database

☐ LDAP

☐ Unix user/group database

Authorization

☐ Anyone can do anything

☐ Legacy mode

☒ Logged-in users can do anything

☐ Matrix-based security

☐ Project-based Matrix Authorization Strategy

After configuring security, you can begin to create users. At the homepage, you can click on `Sign Up` on the upper-right hand of the page.

6.3.3 Adding Build Configurations

After you log into Jenkins, you will be able to click `Create Item`. From this page you will be able to create a new build configuration. An example of the `scala-tdd-fundamentals` build configuration can be seen below. In this example, we're using GitHub so we've installed the `GitBucket` plugin into Jenkins.

The screenshot shows the Jenkins configuration page for a project named 'scala-tdd-fundamentals'. The left sidebar contains navigation links: Back to Dashboard, Status, Changes, Workspace, Build Now, Delete Project, Configure, GitBucket, and GitBucket Hook Log. The main configuration area is divided into several sections:

- Project name:** scala-tdd-fundamentals
- Description:** [Escaped HTML] [Preview](#)
- ☐ Discard Old Builds
- GitBucket:**
 - URL: <https://github.com/LoyolaChicagoCode/scala-tdd-fundamentals>
 - ☐ Enable hyperlink to the issue
 - ☐ This build is parameterized
 - ☐ Disable Build (No new builds will be executed until the project is re-enabled.)
 - ☒ Execute concurrent builds if necessary
- Build Triggers:**
 - ☐ Trigger builds remotely (e.g., from scripts)
 - ☐ Build after other projects are built
 - ☒ Build periodically
 - Schedule: `H H * * *`
Would last have run at Sunday, January 18, 2015 6:56:13 PM CST; would next run at Monday, January 19, 2015 6:56:13 PM CST.
 - ☒ Build when a change is pushed to GitBucket
 - Pass-through Git commit ☐
 - ☐ Poll SCM
- Build:**
 - Execute shell:**
Command: `sbt compile`
`exit $?`
[See the list of available environment variables](#) [Delete](#)
 - Execute shell:**
Command: `sbt test`
`exit $?`
[See the list of available environment variables](#) [Delete](#)

After you have saved this configuration, Jenkins will be able to watch your version control system for changes, perform builds, and run all of your unit tests on each checkin. Everyone working on the software will be able to see that all of the changes that are checked in, that they compile, and that tests pass.

Todo

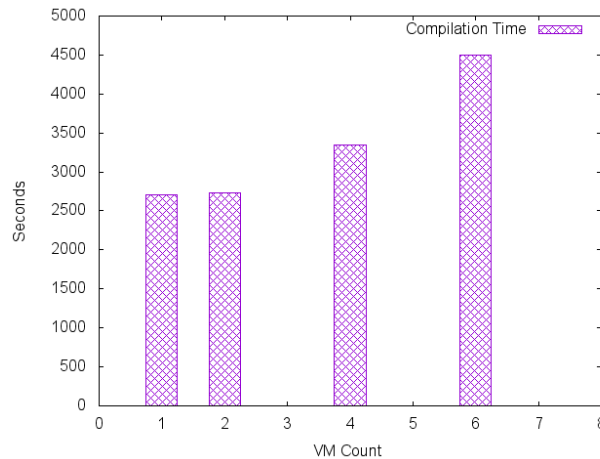
IntelliJ IDEA plugin

6.4 Scaling Continuous Integration

Many build systems and test frameworks are single threaded systems. With modern servers that have multiple cores and RAID based storage systems that support parallel I/O, build servers are often under utilized.

A demonstration of this can be seen with a compilation of the Linux kernel. In this test, a system with 24 logical cores and a RAID-10 SSD storage system was used. A test was performed with one, two, four, and six virtual machines with four virtual cores each. Each VM ran a single build of the Linux kernel. In the figure below, we can see that there is

no noticable difference between having one build server and having two. Also, performance only decreases by about 20% when the VM count is increased to four VMs. A greater loss of performance of about 40% occurs with six virtual machines.



We recommend that when scaling your continuous integration system to include more build servers, to consider the current utilization of the existing physical servers. In many cases, modern hardware is able to support more than one build server per physical server.

6.5 Continuous Integration Frequency

Tests have different execution performance and purpose. Some tests execute quickly and verify one unit of code. Other tests execute more slowly and cover a larger segment of code. Still other tests will involve third party technologies such as web services or databases. All of these tests are important to run in your continuous integration system, but some of these types of tests have challenges that must be addressed.

For this discussion, we divide unit tests into three categories. The first category is tests that are computationally or memory bound. The second category are I/O bound tests and tests that interact heavily with operating system services. The third category of tests are those that work with third party technology and database systems.

An important goal for a continuous integration system is achieve rapid and continuous feedback for project contributors. Running tests with larger run times, tests that don't scale well on the continuous integration system, or tests that have false positive failures interfere with this goal. Of the three categories mentioned above, tests that are computationally and/or memory bound are an excellent fit for this case. On modern systems, computational and memory performance scale quite well. As a suite of tests grows, performance for such tests should remain reasonable.

The two other categories: I/O bound tests and tests involving third party technologies are a bit more complex to consider. For I/O bound tests it is important to consider issues of scale. If several hundred tests that are I/O bound run on a continuous integration system that supports 25 developers with a team average of 75 checkins per day, it does not take much to be running around several hundred thousand I/O bound tests in a day. These tests will often cause builds to queue in a continuous integration system and work against the goal of getting rapid feedback.

For the third category, tests that interact with third party technologies and database systems, there are additional considerations. The first consideration is the issue of periodic failures in third party technologies. Many third party systems have per-call failure rates of 0.1%. Most mature software will build in code around these technologies to react to failures and intelligently retry requests. In a production environment this approach is typically sufficient. In continuous integration, you may experience a different level of scaling. For example, your production software might make occasional requests to a third party web service. To provide coverage to this code, you might write 20 or 30 unit tests to make sure your usage of this service is consistent with its behavior. When run in a continuous integration

environment, these 20 or 30 unit tests could translate to tens or even a few hundreds of calls to the service over a short duration. With other builds running in parallel in the continuous integration system, the third party service may experience bursts of several hundred requests in a short period of time when it was only designed for tens of requests for production. When services like these don't scale up, they can create difficult to reproduce false positive failures in your continuous builds.

So, what's the solution for these two categories? Our recommended solution is to put these tests into a continuously running rolling build instead of a per-checkin build that faster tests run in. Such a build could be triggered every thirty minutes, or be queued each time the previous one completes. The advantage to this approach is that it places an upper limit on the number of tests making use of I/O and/or third party technologies in a frame of time. Whether 10 checkins were made or 1 checkin was made in the last thirty minutes, the same number of slower running tests will be run regardless. This approach reduces the load put on third party services and on the continuous integration system in general. Also, by running these tests continuously, you will be able to see results from these tests several times a day.

6.6 False Positives and Periodic Failure in Computationally or Memory Bound Tests

In computationally and/or memory bound tests, there are a few categories of periodic failures that need to be considered. These categories include tests that involve time, multi-threading, and the order stability of collections and results from computations. For each of these categories we will explore ways to write assertions to be tolerant of reasonable differences in individual executions of a unit test.

6.6.1 Order Stability in Tests

In many languages, and Scala is no exception, there are algorithms that do not preserve order stability. It is not uncommon to see sparse data structures like hash tables, built in sort algorithms, and others inconsistently manage stability. In the construction of algorithms the property of stability is sometimes important. For example, radix sort would not work correctly if its sorting subroutine was not itself a stable sort.

When making assertions, make sure to note when your test is implying an order and whether that order is truly needed. For example, you may wish to assert that two items are in a list. One approach is to assert that the first element in the list is the first item and the second element in the list is the second item. A second approach is to assert for each item that the item is contained somewhere in the list. This second assertion does not depend on the stability of the algorithm that produces the list.

This kind of behavior is also common for hash tables. When the default hash is the internal or managed memory address of an object, two different runs can produce two separate orders of items in the hash table. Where hashes are more deterministic, this is not the case.

Todo

add examples for these kinds of assertions

6.6.2 Multi Threading in Tests

Writing simple, correct, and efficient multi-threaded code requires a good deal of thought and attention to detail. When problems occur in multi-threaded code, they can be difficult to reproduce or rare to occur. Sometimes, a multi-threaded program will work just fine on a 2-CPU system, but run into trouble when it is put on a 8-CPU system. These kinds of issues are often encountered in a continuous integration environment.

There are often differences between a developer's computer, a customer's computer, and a build server in a continuous integration system. These differences can be seen in the number of processor cores, the sizes of caches, the available

memory and the software environment on these systems. Many of these factors can lead to quite different execution timing in multi-threaded code. Sometimes it is the case that a test will fail in continuous integration 10% of the time, but never fail on a developer's machine. These kinds of failures can be quite frustrating to figure out.

There are two important things to consider when testing such code. One is whether or not to test your code in a multi threaded execution environment. It is possible to test its individual components in one thread each to verify each component without assembling them all together for a more integrated test in a multi-threaded environment. There is value in both types of tests. Another consideration is how you respond to failure in the multi-threaded tests that run on the continuous integration environment. A great advantage of these tests running on a continuous integration environment is that they get executed often. So, multi-threading problems that will only reliably occur 0.1% of the time will show up as failures at least a few times over the course of a few days. While this is not the immediate feedback we'd like in continuous builds, it does give us a larger sample size and a larger number of permutations of the multi-threaded execution. The longer a test is running successfully, the more confident we can be in the correctness of our programs.

6.6.3 Tests and Code Concerning Time

In your application you may need to interact with library functions to retrieve the current clock time from the operating system. When this type of code is tested, there are some special considerations to make. Before proceeding, it is important to discuss the behavior of clocks on different systems. One example is the difference between the scheduler quantum on server and client operating systems. Often it is the case that the scheduler quantum is longer, on the order of 100ms, for server operating systems, and is much shorter on mobile devices, and desktop computers, typically on the order of 10-20ms. When the time is retrieved from the operating system, if there is other code that makes a call to the operating system before your assertions, there can be differing behavior on user system and server systems.

For example, you may be able to assert that a recently retrieved time value is the same as a subsequently retrieved time value and get a positive result 99% of the time on a desktop system. However, when the timing changes on a server system, which may be the system you're using in your continuous integration system, these assertions may break down.

When possible, it is a best practice to work with fixed time values in your tests. If you can pass into your code under test a fixed time value, then you can be sure that your tests' assertions will always be valid.

UI TESTING

ADVANCED TOPICS

- Testing Web Apps/Services (Play/spray)

Some initial explorations of this topic are available [here](#). This example uses these libraries:

- [Dispatch](#) as an HTTP client for interacting with the server mock.
- [specs2](#) for its nice matchers (especially JSON).

- Testing Concurrent/Parallel/Distributed Code
- Test and Coupling
- Performance Testing?

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`