Testing in Scala

@dhinojosa

https://github.com/dhinojosa/testing-scala

ScalaTest

ScalaTest

- Created by “Venerable” Bill Venners
- JUnit, TestNG Integration
- Various Specs depending on need
- http://www.scalatest.org/
- Runs on Ant, SBT, Maven

ScalaTest TestNG

```
package com.xyzcorp.scala.study
import org.testng.annotations.Test
import org.scalatest.testng.TestNGSuite
import org.scalatest.matchers.MustMatchers
class EmployeeTestWithTestNG extends TestNGSuite with
      MustMatchers {

      @Test()
      def testCreationOfEmployeeObjectAndProperties() {
          val emp = new Employee("Lex", "Luthor")
          emp.firstName must equal ("Lex")
          emp.firstName must include regex ("L.x")
          emp.lastName must include regex ("L.*r")// success!
      }
}
```

ScalaTest TestNG

```
package com.xyzcorp.scala.study
import org.testng.annotations.Test
import org.scalatest.testng.TestNGSuite
import org.scalatest.matchers.MustMatchers
class EmployeeTestWithTestNG extends TestNGSuite with
      MustMatchers {

      @Test()
      def testCreationOfEmployeeObjectAndProperties() {
          val emp = new Employee("Lex", "Luthor")
          emp.firstName must equal ("Lex")
          emp.firstName must include regex ("L.x")
          emp.lastName must include regex ("L.*r")// success!
      }
}
```

ScalaTest TestNG

```
package com.xyzcorp.scala.study
import org.testng.annotations.Test
import org.scalatest.testng.TestNGSuite
import org.scalatest.matchers.MustMatchers
class EmployeeTestWithTestNG extends TestNGSuite with
      MustMatchers {

      @Test()
      def testCreationOfEmployeeObjectAndProperties() {
          val emp = new Employee("Lex", "Luthor")
          emp.firstName must equal ("Lex")
          emp.firstName must include regex ("L.x")
          emp.lastName must include regex ("L.*r")// success!
      }
}
```

ScalaTest TestNG

```
package com.xyzcorp.scala.study
import org.testng.annotations.Test
import org.scalatest.testng.TestNGSuite
import org.scalatest.matchers.MustMatchers
class EmployeeTestWithTestNG extends TestNGSuite with
      MustMatchers {

      @Test()
      def testCreationOfEmployeeObjectAndProperties() {
          val emp = new Employee("Lex", "Luthor")
          emp.firstName must equal ("Lex")
          emp.firstName must include regex ("L.x")
          emp.lastName must include regex ("L.*r")// success!
      }
}
```

ScalaTest TestNG Results

```
[info] == com.evolutionnext.scala.study.EmployeeTestWithTestNG ==
[info] EmployeeTestWithTestNG:
[TestNG] Running:
  Command line suite

[info] EmployeeTestWithTestNG:
[info] - testCreationOfEmployeeObjectAndProperties

===============================================
Command line suite
Total tests run: 1, Failures: 0, Skips: 0
===============================================

[info] == com.evolutionnext.scala.study.EmployeeTestWithTestNG ==
```

ScalaTest JUnit

```
package com.xyzcorp.scala.study
import org.junit.Assert._
import org.junit.Test
import org.junit.Before
import org.scalatest.junit.{JUnitSuite, AssertionsForJUnit}
import org.scalatest.matchers.MustMatchers
class EmployeeTestWithJUnit4 extends JUnitSuite with
      MustMatchers {
      @Test def testCreationOfEmployeeObjectAndProperties() {
         val emp = new Employee("Lex", "Luthor")
         emp.firstName must equal("Lex")
         emp.firstName must include regex ("L.x")
         emp.lastName must include regex ("L.*r") // Success!
      }
}
```

ScalaTest JUnit

```
package com.xyzcorp.scala.study
import org.junit.Assert._
import org.junit.Test
import org.junit.Before
import org.scalatest.junit.{JUnitSuite, AssertionsForJUnit}
import org.scalatest.matchers.MustMatchers
class EmployeeTestWithJUnit4 extends JUnitSuite with
      MustMatchers {
      @Test def testCreationOfEmployeeObjectAndProperties() {
         val emp = new Employee("Lex", "Luthor")
         emp.firstName must equal("Lex")
         emp.firstName must include regex ("L.x")
         emp.lastName must include regex ("L.*r") // Success!
      }
}
```

ScalaTest JUnit

```
package com.xyzcorp.scala.study
import org.junit.Assert._
import org.junit.Test
import org.junit.Before
import org.scalatest.junit.{JUnitSuite, AssertionsForJUnit}
import org.scalatest.matchers.MustMatchers
class EmployeeTestWithJUnit4 extends JUnitSuite with
      MustMatchers {
      @Test def testCreationOfEmployeeObjectAndProperties() {
         val emp = new Employee("Lex", "Luthor")
         emp.firstName must equal("Lex")
         emp.firstName must include regex ("L.x")
         emp.lastName must include regex ("L.*r") // Success!
      }
}
```

ScalaTest JUnit Results

```
[info] == com.evolutionnext.scala.study.EmployeeTestWithJUnit4 ==
[info] EmployeeTestWithJUnit4:
[info] - testCreationOfEmployeeObjectAndProperties
[info] == com.evolutionnext.scala.study.EmployeeTestWithJUnit4 ==
```

ScalaTest WordSpec

```
package com.xyzcorp.scala.study
import org.scalatest.matchers.MustMatchers
import org.scalatest.{WordSpec, Spec}
class EmployeeTestWithWordSpecBDD extends WordSpec with
      MustMatchers {
   "An Employee" should {
     "return the same first name and last given to its
      constructor" in {
         val emp = new Employee("Lex", "Luthor")
         emp.firstName must equal("Lex")
         emp.lastName must equal("Luthor")
      }
      "throw an IllegalArgumentException if
       an empty string is given as a first name" in {
         intercept[IllegalArgumentException] {
         new Employee("", "Luthor")
      }
   }
}
```

ScalaTest WordSpec

```
package com.xyzcorp.scala.study
import org.scalatest.matchers.MustMatchers
import org.scalatest.{WordSpec, Spec}
class EmployeeTestWithWordSpecBDD extends WordSpec with
      MustMatchers {
   "An Employee" should {
     "return the same first name and last given to its
      constructor" in {
         val emp = new Employee("Lex", "Luthor")
         emp.firstName must equal("Lex")
         emp.lastName must equal("Luthor")
      }
      "throw an IllegalArgumentException if
       an empty string is given as a first name" in {
         intercept[IllegalArgumentException] {
         new Employee("", "Luthor")
      }
   }
}
```

ScalaTest WordSpec

```
package com.xyzcorp.scala.study
import org.scalatest.matchers.MustMatchers
import org.scalatest.{WordSpec, Spec}
class EmployeeTestWithWordSpecBDD extends WordSpec with
      MustMatchers {
   "An Employee" should {
     "return the same first name and last given to its
      constructor" in {
         val emp = new Employee("Lex", "Luthor")
         emp.firstName must equal("Lex")
         emp.lastName must equal("Luthor")
      }
      "throw an IllegalArgumentException if
       an empty string is given as a first name" in {
         intercept[IllegalArgumentException] {
         new Employee("", "Luthor")
      }
   }
}
```

ScalaTest WordSpec

```
package com.xyzcorp.scala.study
import org.scalatest.matchers.MustMatchers
import org.scalatest.{WordSpec, Spec}
class EmployeeTestWithWordSpecBDD extends WordSpec with
      MustMatchers {
   "An Employee" should {
     "return the same first name and last given to its
      constructor" in {
         val emp = new Employee("Lex", "Luthor")
         emp.firstName must equal("Lex")
         emp.lastName must equal("Luthor")
      }
      "throw an IllegalArgumentException if
       an empty string is given as a first name" in {
         intercept[IllegalArgumentException] {
         new Employee("", "Luthor")
      }
   }
}
```

ScalaTest WordSpec

```
package com.xyzcorp.scala.study
import org.scalatest.matchers.MustMatchers
import org.scalatest.{WordSpec, Spec}
class EmployeeTestWithWordSpecBDD extends WordSpec with
      MustMatchers {
   "An Employee" should {
     "return the same first name and last given to its
      constructor" in {
         val emp = new Employee("Lex", "Luthor")
         emp.firstName must equal("Lex")
         emp.lastName must equal("Luthor")
      }
      "throw an IllegalArgumentException if
       an empty string is given as a first name" in {
         intercept[IllegalArgumentException] {
         new Employee("", "Luthor")
      }
   }
}
```

ScalaTest WordSpec Results

```
[info] == com.evolutionnext.scala.study.EmployeeTestWithWordSpecBDD ==
[info] EmployeeTestWithWordSpecBDD:
[info] An Employee
[info] - should return the same first name and last given to it's constructor
[info] - should throw IllegalArgumentException if an empty string is given as
a first name
[info] == com.evolutionnext.scala.study.EmployeeTestWithWordSpecBDD ==
```

ScalaTest FreeSpec

```
package com.evolutionnext.scala.study

import org.scalatest.matchers.MustMatchers
import org.scalatest.FreeSpec

class EmployeeTestWithFreeSpec extends FreeSpec with MustMatchers {
  "An Employee" - {
    "is a curious thing" - {
      "they do what you tell them to, up to a certain point, and
           that point is 5PM" - {
         "but I digress, if you give it a firstName and lastName,
                the fullName should be
                the concactenation of both" in {
          val emp = new Employee("Lex", "Luthor")
          emp.firstName must equal("Lex")
          emp.lastName must equal("Luthor")
        }
      }
    }
  }
}
```

ScalaTest FreeSpec

```
package com.evolutionnext.scala.study

import org.scalatest.matchers.MustMatchers
import org.scalatest.FreeSpec

class EmployeeTestWithFreeSpec extends FreeSpec with MustMatchers {
  "An Employee" - {
    "is a curious thing" - {
      "they do what you tell them to, up to a certain point, and
           that point is 5PM" - {
         "but I digress, if you give it a firstName and lastName,
                the fullName should be
                the concactenation of both" in {
          val emp = new Employee("Lex", "Luthor")
          emp.firstName must equal("Lex")
          emp.lastName must equal("Luthor")
        }
      }
    }
  }
}
```

ScalaTest FreeSpec

```
package com.evolutionnext.scala.study

import org.scalatest.matchers.MustMatchers
import org.scalatest.FreeSpec

class EmployeeTestWithFreeSpec extends FreeSpec with MustMatchers {
  "An Employee" - {
    "is a curious thing" - {
      "they do what you tell them to, up to a certain point, and
           that point is 5PM" - {
         "but I digress, if you give it a firstName and lastName,
                the fullName should be
                the concactenation of both" in {
          val emp = new Employee("Lex", "Luthor")
          emp.firstName must equal("Lex")
          emp.lastName must equal("Luthor")
        }
      }
    }
  }
}
```

ScalaTest FreeSpec

```
package com.evolutionnext.scala.study

import org.scalatest.matchers.MustMatchers
import org.scalatest.FreeSpec

class EmployeeTestWithFreeSpec extends FreeSpec with MustMatchers {
  "An Employee" - {
    "is a curious thing" - {
      "they do what you tell them to, up to a certain point, and
           that point is 5PM" - {
         "but I digress, if you give it a firstName and lastName,
                the fullName should be
                the concactenation of both" in {
          val emp = new Employee("Lex", "Luthor")
          emp.firstName must equal("Lex")
          emp.lastName must equal("Luthor")
        }
      }
    }
  }
}
```

ScalaTest FreeSpec

```
package com.evolutionnext.scala.study

import org.scalatest.matchers.MustMatchers
import org.scalatest.FreeSpec

class EmployeeTestWithFreeSpec extends FreeSpec with MustMatchers {
  "An Employee" - {
    "is a curious thing" - {
      "they do what you tell them to, up to a certain point, and
           that point is 5PM" - {
         "but I digress, if you give it a firstName and lastName,
                the fullName should be
                the concactenation of both" in {
          val emp = new Employee("Lex", "Luthor")
          emp.firstName must equal("Lex")
          emp.lastName must equal("Luthor")
        }
      }
    }
  }
}
```

ScalaTest FreeSpec

```
package com.evolutionnext.scala.study

import org.scalatest.matchers.MustMatchers
import org.scalatest.FreeSpec

class EmployeeTestWithFreeSpec extends FreeSpec with MustMatchers {
  "An Employee" - {
    "is a curious thing" - {
      "they do what you tell them to, up to a certain point, and
           that point is 5PM" - {
         "but I digress, if you give it a firstName and lastName,
                the fullName should be
                the concactenation of both" in {
          val emp = new Employee("Lex", "Luthor")
          emp.firstName must equal("Lex")
          emp.lastName must equal("Luthor")
        }
      }
    }
  }
}
```

ScalaTest FreeSpec Results

```
[info] == com.evolutionnext.scala.study.EmployeeTestWithFreeSpec ==
[info] EmployeeTestWithFreeSpec:
[info] An Employee
[info]   is a curious thing
[info]     they do what you tell them to, up to a certain point, and
that point is 5PM
[info]     - but I digress, if you give it a firstName and lastName,
the fullName should be
[info]                   the concactenation of both
[info] == com.evolutionnext.scala.study.EmployeeTestWithFreeSpec ==
```

ScalaTest Feature Spec

```
class EmployeeTestFeatureSpec extends FeatureSpec with GivenWhenThen with ShouldMatchers{

  feature("A user should be able to create an Employee in different ways") {

    info("When I create an Employee")
    info("I'd like to just give it the firstName and lastName")
    info("with or without a social security number")

    scenario("""An Employee is created without a social security number
                but first and last name are added""") {

      given("A first name of 'Dan' and a last name of 'Hinojosa")
      val firstName = "Dan"
      val lastName = "Hinojosa"

      when("A new Employee is called with the first and last name")
      val employee = new Employee(firstName, lastName)

      then("employee.first name should return 'Dan'")
      employee.firstName should be("Dan")

      and("employee.lastName should be 'Hinojosa'")
      employee.lastName should be("Hinojosa")

      and("employee.ssn should be 000-00-000")
      employee.ssn should be("000-00-0000")
    }
  }
}
```

ScalaTest Feature Spec

```
class EmployeeTestFeatureSpec extends FeatureSpec with GivenWhenThen with ShouldMatchers{

  feature("A user should be able to create an Employee in different ways") {

    info("When I create an Employee")
    info("I'd like to just give it the firstName and lastName")
    info("with or without a social security number")

    scenario("""An Employee is created without a social security number
                but first and last name are added""") {

      given("A first name of 'Dan' and a last name of 'Hinojosa")
      val firstName = "Dan"
      val lastName = "Hinojosa"

      when("A new Employee is called with the first and last name")
      val employee = new Employee(firstName, lastName)

      then("employee.first name should return 'Dan'")
      employee.firstName should be("Dan")

      and("employee.lastName should be 'Hinojosa'")
      employee.lastName should be("Hinojosa")

      and("employee.ssn should be 000-00-000")
      employee.ssn should be("000-00-0000")
    }
  }
}
```

ScalaTest Feature Spec

```
class EmployeeTestFeatureSpec extends FeatureSpec with GivenWhenThen with ShouldMatchers{

  feature("A user should be able to create an Employee in different ways") {

    info("When I create an Employee")
    info("I'd like to just give it the firstName and lastName")
    info("with or without a social security number")

    scenario("""An Employee is created without a social security number
                but first and last name are added""") {

      given("A first name of 'Dan' and a last name of 'Hinojosa")
      val firstName = "Dan"
      val lastName = "Hinojosa"

      when("A new Employee is called with the first and last name")
      val employee = new Employee(firstName, lastName)

      then("employee.first name should return 'Dan'")
      employee.firstName should be("Dan")

      and("employee.lastName should be 'Hinojosa'")
      employee.lastName should be("Hinojosa")

      and("employee.ssn should be 000-00-000")
      employee.ssn should be("000-00-0000")
    }
  }
}
```

ScalaTest Feature Spec

```
class EmployeeTestFeatureSpec extends FeatureSpec with GivenWhenThen with ShouldMatchers{

  feature("A user should be able to create an Employee in different ways") {

    info("When I create an Employee")
    info("I'd like to just give it the firstName and lastName")
    info("with or without a social security number")

    scenario("""An Employee is created without a social security number
                but first and last name are added""") {

      given("A first name of 'Dan' and a last name of 'Hinojosa")
      val firstName = "Dan"
      val lastName = "Hinojosa"

      when("A new Employee is called with the first and last name")
      val employee = new Employee(firstName, lastName)

      then("employee.first name should return 'Dan'")
      employee.firstName should be("Dan")

      and("employee.lastName should be 'Hinojosa'")
      employee.lastName should be("Hinojosa")

      and("employee.ssn should be 000-00-000")
      employee.ssn should be("000-00-0000")
    }
  }
}
```

ScalaTest Feature Spec

```
class EmployeeTestFeatureSpec extends FeatureSpec with GivenWhenThen with ShouldMatchers{

  feature("A user should be able to create an Employee in different ways") {

    info("When I create an Employee")
    info("I'd like to just give it the firstName and lastName")
    info("with or without a social security number")

    scenario("""An Employee is created without a social security number
                but first and last name are added""") {

      given("A first name of 'Dan' and a last name of 'Hinojosa")
      val firstName = "Dan"
      val lastName = "Hinojosa"

      when("A new Employee is called with the first and last name")
      val employee = new Employee(firstName, lastName)

      then("employee.first name should return 'Dan'")
      employee.firstName should be("Dan")

      and("employee.lastName should be 'Hinojosa'")
      employee.lastName should be("Hinojosa")

      and("employee.ssn should be 000-00-000")
      employee.ssn should be("000-00-0000")
    }
  }
}
```

ScalaTest Feature Spec

```
class EmployeeTestFeatureSpec extends FeatureSpec with GivenWhenThen with ShouldMatchers{

  feature("A user should be able to create an Employee in different ways") {

    info("When I create an Employee")
    info("I'd like to just give it the firstName and lastName")
    info("with or without a social security number")

    scenario("""An Employee is created without a social security number
                but first and last name are added""") {

      given("A first name of 'Dan' and a last name of 'Hinojosa")
      val firstName = "Dan"
      val lastName = "Hinojosa"

      when("A new Employee is called with the first and last name")
      val employee = new Employee(firstName, lastName)

      then("employee.first name should return 'Dan'")
      employee.firstName should be("Dan")

      and("employee.lastName should be 'Hinojosa'")
      employee.lastName should be("Hinojosa")

      and("employee.ssn should be 000-00-000")
      employee.ssn should be("000-00-0000")
    }
  }
}
```

ScalaTest Feature Spec

```
class EmployeeTestFeatureSpec extends FeatureSpec with GivenWhenThen with ShouldMatchers{

  feature("A user should be able to create an Employee in different ways") {

    info("When I create an Employee")
    info("I'd like to just give it the firstName and lastName")
    info("with or without a social security number")

    scenario("""An Employee is created without a social security number
                but first and last name are added""") {

      given("A first name of 'Dan' and a last name of 'Hinojosa")
      val firstName = "Dan"
      val lastName = "Hinojosa"

      when("A new Employee is called with the first and last name")
      val employee = new Employee(firstName, lastName)

      then("employee.first name should return 'Dan'")
      employee.firstName should be("Dan")

      and("employee.lastName should be 'Hinojosa'")
      employee.lastName should be("Hinojosa")

      and("employee.ssn should be 000-00-000")
      employee.ssn should be("000-00-0000")
    }
  }
}
```

ScalaTest Feature Spec

```
class EmployeeTestFeatureSpec extends FeatureSpec with GivenWhenThen with ShouldMatchers{

  feature("A user should be able to create an Employee in different ways") {

    info("When I create an Employee")
    info("I'd like to just give it the firstName and lastName")
    info("with or without a social security number")

    scenario("""An Employee is created without a social security number
                but first and last name are added""") {

      given("A first name of 'Dan' and a last name of 'Hinojosa")
      val firstName = "Dan"
      val lastName = "Hinojosa"

      when("A new Employee is called with the first and last name")
      val employee = new Employee(firstName, lastName)

      then("employee.first name should return 'Dan'")
      employee.firstName should be("Dan")

      and("employee.lastName should be 'Hinojosa'")
      employee.lastName should be("Hinojosa")

      and("employee.ssn should be 000-00-000")
      employee.ssn should be("000-00-0000")
    }
  }
}
```

ScalaTest Feature Spec

```
class EmployeeTestFeatureSpec extends FeatureSpec with GivenWhenThen with ShouldMatchers{

  feature("A user should be able to create an Employee in different ways") {

    info("When I create an Employee")
    info("I'd like to just give it the firstName and lastName")
    info("with or without a social security number")

    scenario("""An Employee is created without a social security number
                but first and last name are added""") {

      given("A first name of 'Dan' and a last name of 'Hinojosa")
      val firstName = "Dan"
      val lastName = "Hinojosa"

      when("A new Employee is called with the first and last name")
      val employee = new Employee(firstName, lastName)

      then("employee.first name should return 'Dan'")
      employee.firstName should be("Dan")

      and("employee.lastName should be 'Hinojosa'")
      employee.lastName should be("Hinojosa")

      and("employee.ssn should be 000-00-000")
      employee.ssn should be("000-00-0000")
    }
  }
}
```

ScalaTest Feature Spec

```
class EmployeeTestFeatureSpec extends FeatureSpec with GivenWhenThen with ShouldMatchers{

  feature("A user should be able to create an Employee in different ways") {

    info("When I create an Employee")
    info("I'd like to just give it the firstName and lastName")
    info("with or without a social security number")

    scenario("""An Employee is created without a social security number
                but first and last name are added""") {

      given("A first name of 'Dan' and a last name of 'Hinojosa")
      val firstName = "Dan"
      val lastName = "Hinojosa"

      when("A new Employee is called with the first and last name")
      val employee = new Employee(firstName, lastName)

      then("employee.first name should return 'Dan'")
      employee.firstName should be("Dan")

      and("employee.lastName should be 'Hinojosa'")
      employee.lastName should be("Hinojosa")

      and("employee.ssn should be 000-00-000")
      employee.ssn should be("000-00-0000")
    }
  }
}
```

Specs2

Specs2

- Created by Eric Torreborre
- Testing Framework
- http://etorreborre.github.com/specs2/
- Unit and Acceptance Testing
- Concurrent
- Fragments
- Multiple Reports
- SBT, Maven (with JUnit)

Specs2 Unit Specification

```
package com.xyzcorp.scala.study
import org.specs.mutable._
class EmployeeUnitSpecification extends Specification {
    "An Employee" should {
      "return the same first name and last given to its
       constructor" in {
         val emp = new Employee("Lex", "Luthor")
         emp.firstName must be("Lex")
         emp.lastName must be("Luthor") //Different matcher
                                        //methods for specs
       }
       "throw IllegalArgumentException if an empty
        string is given as a first name" in {
           new Employee("", "Luthor") must
              throwA[IllegalArgumentException]
       }
    }
}
```

Specs2 Unit Specification Results

```
[info] == com.evolutionnext.scala.study.EmployeeUnitSpecification ==
[info] An employee should
[info] + return the same first name and last given to it's
constructor
[info] + throw IllegalArgumentException if an empty string is given
as a first name
[info]
[info] Total for specification EmployeeUnitSpecification
[info] Finished in 276 ms
[info] 2 examples, 0 failure, 0 error
[info]
[info] == com.evolutionnext.scala.study.EmployeeUnitSpecification ==
```

Specs2 DataTables

```
import org.specs2.matcher.DataTables
import org.specs2.mutable.Specification

class EmployeeSpecsDataTable extends Specification with DataTables {
    """Create a battery of first names and last names and validate
        the full names""" in {
       "First Name" | "Last Name"    | "Full Name"           |>
       "Jay"        !! "Zimmerman"   ! "Jay Zimmerman"       |
       "Venkat"     !! "Subramaniam" ! "Venkat Subramaniam" |
       "Ken"        !! "Sipe"        ! "Ken Sipe"            |
       "Tim     "   !! "Berglund"    ! "Tim Berglund"        |
       "Matthew"    !! "McCullough"  ! "Matthew McCullough" |
       "Brian"      !! "   Sletten"  ! "Brian Sletten"       |
       { (firstName, lastName, fullName) =>
            val emp = new Employee(firstName, lastName)
            emp.fullName must_== fullName
       }
   }
}
```

Specs2 DataTables

```
import org.specs2.matcher.DataTables
import org.specs2.mutable.Specification

class EmployeeSpecsDataTable extends Specification with DataTables {
    """Create a battery of first names and last names and validate
        the full names""" in {
        "First Name" | "Last Name"    | "Full Name"           |>
        "Jay"        !! "Zimmerman"   ! "Jay Zimmerman"       |
        "Venkat"     !! "Subramaniam" ! "Venkat Subramaniam"  |
        "Ken"        !! "Sipe"        ! "Ken Sipe"            |
        "Tim     "   !! "Berglund"    ! "Tim Berglund"        |
        "Matthew"    !! "McCullough"  ! "Matthew McCullough"  |
        "Brian"      !! "   Sletten"  ! "Brian Sletten"       |
        { (firstName, lastName, fullName) =>
            val emp = new Employee(firstName, lastName)
            emp.fullName must_== fullName
        }
    }
}
```

Specs2 DataTables Results

```
[info] == com.evolutionnext.scala.study.EmployeeSpecsDataTable ==
[error] x Create a battery of first names and last names and
validate the full names
[error]     |First Name|Last Name|Full Name|
[error] + |Jay|Zimmerman|Jay Zimmerman|
[error] + |Venkat|Subramaniam|Venkat Subramaniam|
[error] + |Ken|Sipe|Ken Sipe|
[error] x |Tim     |Berglund|Tim Berglund| 'Tim      Berglund' is
not equal to 'Tim Berglund'
[error] + |Matthew|McCullough|Matthew McCullough|
[error] x |Brian|   Sletten|Brian Sletten| 'Brian    Sletten' is not
equal to 'Brian Sletten' (EmployeeSpecsDataTable.scala:15)
[info]
[info] Total for specification EmployeeSpecsDataTable
[info] Finished in 31 ms
[info] 1 example, 1 failure, 0 error
[info]
[info] == com.evolutionnext.scala.study.EmployeeSpecsDataTable ==
```

Specs2 Acceptance Specification

```
import org.specs2._

class EmployeeAcceptanceSpecification extends Specification {  def is =
  "An employee should"                                                ^
                                                                     p^
  "return the same first name and last given to it's constructor"   !e1^
  "throw IllegalArgumentException if first name is empty"           !e2^
                                                                    end
  def e1 = {
    val emp = new Employee("Lex", "Luthor")
    emp.firstName must be("Lex")
    emp.lastName must be("Luthor")
  }

  def e2 = { new Employee("", "Luthor") must throwA[IllegalArgumentException] }
}
```

Specs2 Acceptance Specification

```
import org.specs2._

class EmployeeAcceptanceSpecification extends Specification {  def is =
  "An employee should"                                                ^
                                                                     p^
  "return the same first name and last given to it's constructor"   !e1^
  "throw IllegalArgumentException if first name is empty"           !e2^
                                                                    end

  def e1 = {
    val emp = new Employee("Lex", "Luthor")
    emp.firstName must be("Lex")
    emp.lastName must be("Luthor")
  }

  def e2 = { new Employee("", "Luthor") must throwA[IllegalArgumentException] }
}
```

Specs2 Acceptance Specification

```
import org.specs2._

class EmployeeAcceptanceSpecification extends Specification {  def is =
  "An employee should"                                                ^
                                                                     p^
  "return the same first name and last given to it's constructor"   !e1^
  "throw IllegalArgumentException if first name is empty"           !e2^
                                                                    end
  def e1 = {
    val emp = new Employee("Lex", "Luthor")
    emp.firstName must be("Lex")
    emp.lastName must be("Luthor")
  }

  def e2 = { new Employee("", "Luthor") must throwA[IllegalArgumentException] }
}
```

Specs2 Acceptance Specification

```
import org.specs2._

class EmployeeAcceptanceSpecification extends Specification {  def is =
  "An employee should"                                                ^
                                                                     p^
  "return the same first name and last given to it's constructor"   !e1^
  "throw IllegalArgumentException if first name is empty"           !e2^
                                                                    end
  def e1 = {
    val emp = new Employee("Lex", "Luthor")
    emp.firstName must be("Lex")
    emp.lastName must be("Luthor")
  }

  def e2 = { new Employee("", "Luthor") must throwA[IllegalArgumentException] }
}
```

Specs2 Acceptance Specification

```
import org.specs2._

class EmployeeAcceptanceSpecification extends Specification {  def is =
  "An employee should"                                                ^
                                                                     p^
  "return the same first name and last given to it's constructor"   !e1^
  "throw IllegalArgumentException if first name is empty"           !e2^
                                                                    end

  def e1 = {
    val emp = new Employee("Lex", "Luthor")
    emp.firstName must be("Lex")
    emp.lastName must be("Luthor")
  }

  def e2 = { new Employee("", "Luthor") must throwA[IllegalArgumentException] }
}
```

Specs2 Acceptance Specification

```
import org.specs2._

class EmployeeAcceptanceSpecification extends Specification {  def is =
  "An employee should"                                                ^
                                                                     p^
  "return the same first name and last given to it's constructor"   !e1^
  "throw IllegalArgumentException if first name is empty"           !e2^
                                                                    end
  def e1 = {
    val emp = new Employee("Lex", "Luthor")
    emp.firstName must be("Lex")
    emp.lastName must be("Luthor")
  }

  def e2 = { new Employee("", "Luthor") must throwA[IllegalArgumentException] }
}
```

Specs2 Acceptance Specification

```
import org.specs2._

class EmployeeAcceptanceSpecification extends Specification {  def is =
  "An employee should"                                                ^
                                                                     p^
  "return the same first name and last given to it's constructor"   !e1^
  "throw IllegalArgumentException if first name is empty"           !e2^
                                                                    end

  def e1 = {
    val emp = new Employee("Lex", "Luthor")
    emp.firstName must be("Lex")
    emp.lastName must be("Luthor")
  }

  def e2 = { new Employee("", "Luthor") must throwA[IllegalArgumentException] }
}
```

Specs2 Acceptance Specification Results

```
[info] == com.evolutionnext.scala.study.EmployeeAcceptanceSpecification ==
[info] An employee should
[info]
[info] + return the same first name and last given to it's constructor
[info] + throw IllegalArgumentException if an empty string is given as a first
name
[info] Total for specification EmployeeAcceptanceSpecification
[info] Finished in 18 ms
[info] 2 examples, 0 failure, 0 error
[info]
[info] == com.evolutionnext.scala.study.EmployeeAcceptanceSpecification ==
```

Specs2 Given-When-Then Acceptance Specification

```
import org.specs2.specification.{Then, When, Given}
import org.specs2.execute.Result
import org.specs2.Specification

class EmployeeGWTAcceptanceSpecification extends Specification { def is =

  "An employee given-when-then example"                                  ^
                                                                         p^
  "Given a blank first name ${Lady}"                                     ^ firstName ^
  "And given a last name of ${Gaga}"                                     ^ lastName ^
  "Then I should get an Employee with name of ${Lady Gaga}"              ^ result ^
                                                                         end

   object firstName extends Given[String] {
     def extract(text: String): String = extract1(text)
   }

   object lastName extends When[String, Employee] {
     def extract(firstName:String, text:String) =
        new Employee(firstName, extract1(text))
   }

   object result extends Then[Employee] {
     def extract(employee: Employee, text: String): Result =
        employee.fullName must_== extract1(text)
   }
}
```

Specs2 Given-When-Then Acceptance Specification

```
import org.specs2.specification.{Then, When, Given}
import org.specs2.execute.Result
import org.specs2.Specification

class EmployeeGWTAcceptanceSpecification extends Specification { def is =

  "An employee given-when-then example"                                  ^
                                                                         p^
  "Given a blank first name ${Lady}"                                     ^ firstName ^
  "And given a last name of ${Gaga}"                                     ^ lastName ^
  "Then I should get an Employee with name of ${Lady Gaga}"              ^ result ^
                                                                         end

   object firstName extends Given[String] {
     def extract(text: String): String = extract1(text)
   }

   object lastName extends When[String, Employee] {
     def extract(firstName:String, text:String) =
        new Employee(firstName, extract1(text))
   }

   object result extends Then[Employee] {
     def extract(employee: Employee, text: String): Result =
        employee.fullName must_== extract1(text)
   }
}
```

Specs2 Given-When-Then Acceptance Specification

```
import org.specs2.specification.{Then, When, Given}
import org.specs2.execute.Result
import org.specs2.Specification

class EmployeeGWTAcceptanceSpecification extends Specification { def is =

  "An employee given-when-then example"                                  ^
                                                                         p^
  "Given a blank first name ${Lady}"                                     ^ firstName ^
  "And given a last name of ${Gaga}"                                     ^ lastName ^
  "Then I should get an Employee with name of ${Lady Gaga}"              ^ result ^
                                                                         end

   object firstName extends Given[String] {
     def extract(text: String): String = extract1(text)
   }

   object lastName extends When[String, Employee] {
     def extract(firstName:String, text:String) =
        new Employee(firstName, extract1(text))
   }

   object result extends Then[Employee] {
     def extract(employee: Employee, text: String): Result =
        employee.fullName must_== extract1(text)
   }
}
```

Specs2 Given-When-Then Acceptance Specification

```
import org.specs2.specification.{Then, When, Given}
import org.specs2.execute.Result
import org.specs2.Specification

class EmployeeGWTAcceptanceSpecification extends Specification { def is =

  "An employee given-when-then example"                                  ^
                                                                         p^
  "Given a blank first name ${Lady}"                                     ^ firstName ^
  "And given a last name of ${Gaga}"                                     ^ lastName ^
  "Then I should get an Employee with name of ${Lady Gaga}"              ^ result ^
                                                                         end

   object firstName extends Given[String] {
     def extract(text: String): String = extract1(text)
   }

   object lastName extends When[String, Employee] {
     def extract(firstName:String, text:String) =
        new Employee(firstName, extract1(text))
   }

   object result extends Then[Employee] {
     def extract(employee: Employee, text: String): Result =
        employee.fullName must_== extract1(text)
   }
}
```

Specs2 Given-When-Then Acceptance Specification

```
import org.specs2.specification.{Then, When, Given}
import org.specs2.execute.Result
import org.specs2.Specification

class EmployeeGWTAcceptanceSpecification extends Specification { def is =

  "An employee given-when-then example"                                  ^
                                                                         p^
  "Given a blank first name ${Lady}"                                     ^ firstName ^
  "And given a last name of ${Gaga}"                                     ^ lastName ^
  "Then I should get an Employee with name of ${Lady Gaga}"              ^ result ^
                                                                         end

   object firstName extends Given[String] {
     def extract(text: String): String = extract1(text)
   }

   object lastName extends When[String, Employee] {
     def extract(firstName:String, text:String) =
        new Employee(firstName, extract1(text))
   }

   object result extends Then[Employee] {
     def extract(employee: Employee, text: String): Result =
        employee.fullName must_== extract1(text)
   }
}
```

Specs2 Given-When-Then Acceptance Specification

```
import org.specs2.specification.{Then, When, Given}
import org.specs2.execute.Result
import org.specs2.Specification

class EmployeeGWTAcceptanceSpecification extends Specification { def is =

  "An employee given-when-then example"                                  ^
                                                                         p^
  "Given a blank first name ${Lady}"                                     ^ firstName ^
  "And given a last name of ${Gaga}"                                     ^ lastName ^
  "Then I should get an Employee with name of ${Lady Gaga}"              ^ result ^
                                                                         end

   object firstName extends Given[String] {
     def extract(text: String): String = extract1(text)
   }

   object lastName extends When[String, Employee] {
     def extract(firstName:String, text:String) =
        new Employee(firstName, extract1(text))
   }

   object result extends Then[Employee] {
     def extract(employee: Employee, text: String): Result =
        employee.fullName must_== extract1(text)
   }
}
```

Specs2 Given-When-Then Acceptance Specification

```
import org.specs2.specification.{Then, When, Given}
import org.specs2.execute.Result
import org.specs2.Specification

class EmployeeGWTAcceptanceSpecification extends Specification { def is =

  "An employee given-when-then example"                                  ^
                                                                         p^
  "Given a blank first name ${Lady}"                                     ^ firstName ^
  "And given a last name of ${Gaga}"                                     ^ lastName ^
  "Then I should get an Employee with name of ${Lady Gaga}"              ^ result ^
                                                                         end

   object firstName extends Given[String] {
     def extract(text: String): String = extract1(text)
   }

   object lastName extends When[String, Employee] {
     def extract(firstName:String, text:String) =
        new Employee(firstName, extract1(text))
   }

   object result extends Then[Employee] {
     def extract(employee: Employee, text: String): Result =
        employee.fullName must_== extract1(text)
   }
}
```

Specs2 Given-When-Then Acceptance Specification Results

```
[info] == com.evolutionnext.scala.study.EmployeeGWTAcceptanceSpecification ==
[info] An employee given-when-then example
[info]
[info] Given a blank first name Lady
[info] And given a last name of Gaga
[info] + Then I should get an Employee with name of Lady Gaga
[info]
[info] Total for specification EmployeeGWTAcceptanceSpecification
[info] Finished in 24 ms
[info] 1 example, 0 failure, 0 error
[info]
[info] == com.evolutionnext.scala.study.EmployeeGWTAcceptanceSpecification ==
```

Scala Check

ScalaCheck

- Created by Rickard Nilsson
- Inspired by QuickCheck
- Test Data Generation
- http://code.google.com/p/scalacheck/
- Run by itself or with ScalaTest and Specs2

ScalaCheck (Pure)

```
package com.xyzcorp.scala.study
import org.scalacheck.Prop._
import org.scalacheck.Properties
object EmployeeScalaCheck extends Properties("Employee") {
    property("fullNameWithNoEmptyString") =
       forAll {(a: String, b: String) =>
         ((!a.isEmpty()) && (!b.isEmpty())) ==> {
           new Employee(a, b).fullName == a.trim() + " " + b.trim()
         }
       }
}
```

ScalaCheck (Pure)

```
package com.xyzcorp.scala.study
import org.scalacheck.Prop._
import org.scalacheck.Properties
object EmployeeScalaCheck extends Properties("Employee") {
    property("fullNameWithNoEmptyString") =
       forAll {(a: String, b: String) =>
         ((!a.isEmpty()) && (!b.isEmpty())) ==> {
           new Employee(a, b).fullName == a.trim() + " " + b.trim()
         }
       }
}
```

ScalaCheck (Pure)

```
package com.xyzcorp.scala.study
import org.scalacheck.Prop._
import org.scalacheck.Properties
object EmployeeScalaCheck extends Properties("Employee") {
    property("fullNameWithNoEmptyString") =
       forAll {(a: String, b: String) =>
         ((!a.isEmpty()) && (!b.isEmpty())) ==> {
           new Employee(a, b).fullName == a.trim() + " " + b.trim()
         }
       }
}
```

ScalaCheck (Pure)

```
package com.xyzcorp.scala.study
import org.scalacheck.Prop._
import org.scalacheck.Properties
object EmployeeScalaCheck extends Properties("Employee") {
    property("fullNameWithNoEmptyString") =
       forAll {(a: String, b: String) =>
         ((!a.isEmpty()) && (!b.isEmpty())) ==> {
           new Employee(a, b).fullName == a.trim() + " " + b.trim()
         }
       }
}
```

ScalaCheck (Pure)

```
package com.xyzcorp.scala.study
import org.scalacheck.Prop._
import org.scalacheck.Properties
object EmployeeScalaCheck extends Properties("Employee") {
    property("fullNameWithNoEmptyString") =
       forAll {(a: String, b: String) =>
         ((!a.isEmpty()) && (!b.isEmpty())) ==> {
           new Employee(a, b).fullName == a.trim() + " " + b.trim()
         }
       }
}
```

ScalaCheck (Pure)

```
package com.xyzcorp.scala.study
import org.scalacheck.Prop._
import org.scalacheck.Properties
object EmployeeScalaCheck extends Properties("Employee") {
    property("fullNameWithNoEmptyString") =
       forAll {(a: String, b: String) =>
         ((!a.isEmpty()) && (!b.isEmpty())) ==> {
           new Employee(a, b).fullName == a.trim() + " " + b.trim()
         }
       }
}
```

ScalaCheck (Pure)

```
package com.xyzcorp.scala.study
import org.scalacheck.Prop._
import org.scalacheck.Properties
object EmployeeScalaCheck extends Properties("Employee") {
    property("fullNameWithNoEmptyString") =
       forAll {(a: String, b: String) =>
         ((!a.isEmpty()) && (!b.isEmpty())) ==> {
           new Employee(a, b).fullName == a.trim() + " " + b.trim()
         }
       }
}
```

ScalaCheck Results

```
[info] == com.xyzcorp.scala.study.EmployeeScalaCheck ==
[info] + Employee.fullNameWithNoEmptyString: OK, passed 100 tests.
[info] == com.xyzcorp.scala.study.EmployeeScalaCheck ==
```

ScalaCheck Generators

```
import org.scalacheck.Prop._
import org.scalacheck.{Gen, Properties}

object EmployeeWithGeneratorsScalaCheck extends
       Properties("Employee") {
     val gen1 = Gen.oneOf("Abigail", "Amber", "Bertha",
                          "Cally", "Diana", "Esther", "Frannie",
                          "Texarkana", "Justine")
     val gen2 = Gen.oneOf("Adams", "Valles", "Simons",
                          "Gomez", "Patel", "Mehra",
                          "Groenfeld", "Thatcher",
                          "Greenfield")
     property("fullNameWithNoEmptyString") =
         forAll(gen1, gen2) {(a: String, b: String) => new
             Employee(a, b).fullName == a.trim() + " " + b.trim()
         }
}
```

ScalaCheck Generators

```
import org.scalacheck.Prop._
import org.scalacheck.{Gen, Properties}

object EmployeeWithGeneratorsScalaCheck extends
       Properties("Employee") {
     val gen1 = Gen.oneOf("Abigail", "Amber", "Bertha",
                          "Cally", "Diana", "Esther", "Frannie",
                          "Texarkana", "Justine")
     val gen2 = Gen.oneOf("Adams", "Valles", "Simons",
                          "Gomez", "Patel", "Mehra",
                          "Groenfeld", "Thatcher",
                          "Greenfield")
     property("fullNameWithNoEmptyString") =
         forAll(gen1, gen2) {(a: String, b: String) => new
             Employee(a, b).fullName == a.trim() + " " + b.trim()
         }
}
```

ScalaCheck Generators

```
import org.scalacheck.Prop._
import org.scalacheck.{Gen, Properties}

object EmployeeWithGeneratorsScalaCheck extends
       Properties("Employee") {
     val gen1 = Gen.oneOf("Abigail", "Amber", "Bertha",
                          "Cally", "Diana", "Esther", "Frannie",
                          "Texarkana", "Justine")
     val gen2 = Gen.oneOf("Adams", "Valles", "Simons",
                          "Gomez", "Patel", "Mehra",
                          "Groenfeld", "Thatcher",
                          "Greenfield")
     property("fullNameWithNoEmptyString") =
         forAll(gen1, gen2) {(a: String, b: String) => new
             Employee(a, b).fullName == a.trim() + " " + b.trim()
         }
}
```

ScalaCheck Arbitrary

```
val someString = Arbitrary.arbitrary[String]
someString.sample: Option[String] = Some(髡섯 ş 邈仮□ □昔 □}□□▪□陮□□ □
已□ 曠峫 ɣ鴿裨㖏□珺鴉㹤防□圔□□ 죕膾玭□擺 ·ȯ琬 □□ꟷ捋)

val someInt = Arbitrary.arbitrary[Int]
someInt.sample: Option[Int] = Some(-1)
```

ScalaCheck with ScalaTest

ScalaCheck with ScalaTest

```
import org.scalatest.prop.GeneratorDrivenPropertyChecks
import org.scalatest.matchers.MustMatchers
import org.scalatest.WordSpec

class ScalaTestWithChecks extends WordSpec with
   GeneratorDrivenPropertyChecks with MustMatchers {
    "fullName" should {
       "return the concatenation of both first name
           and last name given a battery of Strings" in {
        forAll {
           (a: String, b: String) =>
               whenever((!a.isEmpty) && (!b.isEmpty)) {
                  new Employee(a, b).fullName ==
                    a.trim() + " " + b.trim()
        }
      }
    }
  }
}
```

ScalaCheck with ScalaTest

```
import org.scalatest.prop.GeneratorDrivenPropertyChecks
import org.scalatest.matchers.MustMatchers
import org.scalatest.WordSpec

class ScalaTestWithChecks extends WordSpec with
   GeneratorDrivenPropertyChecks with MustMatchers {
    "fullName" should {
       "return the concatenation of both first name
           and last name given a battery of Strings" in {
        forAll {
           (a: String, b: String) =>
               whenever((!a.isEmpty) && (!b.isEmpty)) {
                  new Employee(a, b).fullName ==
                    a.trim() + " " + b.trim()
        }
      }
    }
  }
}
```

ScalaCheck with ScalaTest

```
import org.scalatest.prop.GeneratorDrivenPropertyChecks
import org.scalatest.matchers.MustMatchers
import org.scalatest.WordSpec

class ScalaTestWithChecks extends WordSpec with
   GeneratorDrivenPropertyChecks with MustMatchers {
    "fullName" should {
       "return the concatenation of both first name
           and last name given a battery of Strings" in {
        forAll {
           (a: String, b: String) =>
               whenever((!a.isEmpty) && (!b.isEmpty)) {
                  new Employee(a, b).fullName ==
                    a.trim() + " " + b.trim()
        }
      }
    }
  }
}
```

ScalaCheck with ScalaTest

```
import org.scalatest.prop.GeneratorDrivenPropertyChecks
import org.scalatest.matchers.MustMatchers
import org.scalatest.WordSpec

class ScalaTestWithChecks extends WordSpec with
   GeneratorDrivenPropertyChecks with MustMatchers {
    "fullName" should {
       "return the concatenation of both first name
           and last name given a battery of Strings" in {
        forAll {
           (a: String, b: String) =>
               whenever((!a.isEmpty) && (!b.isEmpty)) {
                  new Employee(a, b).fullName ==
                    a.trim() + " " + b.trim()
        }
      }
    }
  }
}
```

ScalaCheck with ScalaTest Results

```
[info] == com.evolutionnext.scala.study.ScalaTestWithChecks ==
[info] ScalaTestWithChecks:
[info] fullName
[info] - should return the concatenation of both first name and last
name given a battery of Strings
[info] == com.evolutionnext.scala.study.ScalaTestWithChecks ==
```

ScalaCheck with Specs2

ScalaCheck with Specs2

```
import org.specs2.{Specification, ScalaCheck}
import org.specs2.specification.gen.{Then, When, Given}
import org.scalacheck.Arbitrary

class EmployeeGWTAcceptanceWithScalaCheck extends Specification with
ScalaCheck { def is =
  "An employee given-when-then example"                          ^
                                                                 p^
  "Given a first name 'a' And given a last name of 'b'"          ^ firstName ^
  "When I create an Employee using a and b"                      ^ create ^
  "Then I should get an Employee with name concatenating a b" ^ result ^
                                                                 end
   object firstName extends Given[(String, String)] {...}
   object create extends When[(String, String), Employee] {...}
   object result extends Then[Employee] {...}

}
```

ScalaCheck with Specs2

```
import org.specs2.{Specification, ScalaCheck}
import org.specs2.specification.gen.{Then, When, Given}
import org.scalacheck.Arbitrary

class EmployeeGWTAcceptanceWithScalaCheck extends Specification with
ScalaCheck { def is =
  "An employee given-when-then example"                              ^
                                                                     p^
  "Given a first name 'a' And given a last name of 'b'"              ^ firstName ^
  "When I create an Employee using a and b"                          ^ create ^
  "Then I should get an Employee with name concatenating a b"        ^ result ^
                                                                     end
  object firstName extends Given[(String, String)] {
    def extract(text: String) = for {
      x <- Arbitrary.arbitrary[String]
      if x.trim.nonEmpty
      y <- Arbitrary.arbitrary[String]
    } yield (x,y)
  }
  object create extends When[(String, String), Employee] {...}
  object result extends Then[Employee] {...}
}
```

ScalaCheck with Specs2

```
import org.specs2.{Specification, ScalaCheck}
import org.specs2.specification.gen.{Then, When, Given}
import org.scalacheck.Arbitrary

class EmployeeGWTAcceptanceWithScalaCheck extends Specification with
ScalaCheck { def is =
  "An employee given-when-then example"                              ^
                                                                     p^
  "Given a first name 'a' And given a last name of 'b'"              ^ firstName ^
  "When I create an Employee using a and b"                          ^ create ^
  "Then I should get an Employee with name concatenating a b"        ^ result ^
                                                                     end

  object firstName extends Given[(String, String)] {...}
  object create extends When[(String, String), Employee] {
    def extract(names:(String, String), text: String) = new
        Employee(names._1, names._2)
  }
  object result extends Then[Employee] {...}
}
```

ScalaCheck with Specs2

```
import org.specs2.{Specification, ScalaCheck}
import org.specs2.specification.gen.{Then, When, Given}
import org.scalacheck.Arbitrary

class EmployeeGWTAcceptanceWithScalaCheck extends Specification with
ScalaCheck { def is =
  "An employee given-when-then example"                              ^
                                                                     p^
  "Given a first name 'a' And given a last name of 'b'"              ^ firstName ^
  "When I create an Employee using a and b"                          ^ create ^
  "Then I should get an Employee with name concatenating a b"        ^ result ^
                                                                     end
  object firstName extends Given[(String, String)] {...}
  object create extends When[(String, String), Employee] {...}
  object result extends Then[Employee] {
    def extract(text: String)(implicit emp:Arbitrary[Employee]) = {
      check{(emp:Employee) => emp.fullName must_==
          (emp.firstName + " " + emp.lastName)}
    }
  }
}
```

ScalaCheck with Specs2 Results

```
[info] == com.evolutionnext.scala.study.EmployeeGWTAcceptanceWithScalaCheck ==
[info] An employee given-when-then example
[info]
[info] Given a first name 'a' And given a last name of 'b'
[info] When I create an Employee using a and b
[info] + Then I should get an Employee with name concatenating a b
[info]
[info] Total for specification EmployeeGWTAcceptanceWithScalaCheck
[info] Finished in 84 ms
[info] 1 example, 100 expectations, 0 failure, 0 error
[info]
[info] == com.evolutionnext.scala.study.EmployeeGWTAcceptanceWithScalaCheck ==
```

Borachio

Borachio

- Created by Paul Butcher
- Mocking Framework
- Traits and Functions
- http://borachio.com/
- Mixin Trait **`com.borachio.scalatest.MockFactory`** or **`com.borachio.specs2.MockFactory`**
- Future Borachio 2.0-SNAPSHOT
  - Support for type-parameterized methods
  - Support for classes with non-trivial constructors
  - Support for final classes or classes with final methods or private constructors
  - Support for companion objects

Borachio

```
class EmployeeScalaTestWithBorachio extends WordSpec
      with MustMatchers with MockFactory {

  "MyTickerTape" should {
    "return a map of my favorite symbols as keys, and their" +
     "price as values" in {
       val sqp = mock[StockQuoteProvider]
       sqp expects 'getPrice withArgs ("ATT") returning
                   (BigDecimal("33.00"))
       sqp expects 'getPrice withArgs ("ADVS") returning
                   (BigDecimal("12.34"))
       sqp expects 'getPrice withArgs ("MTN") returning
                   (BigDecimal(".22"))

       val tickerTape = new MyTickerTape(sqp, List("ATT", "ADVS", "MTN"))
       tickerTape.getPrices("ATT") must be(BigDecimal(33.00))
       tickerTape.getPrices("ADVS") must be(BigDecimal(12.34))
       tickerTape.getPrices("MTN") must be(BigDecimal(.22))
    }
  }
}
```

Borachio

```
class EmployeeScalaTestWithBorachio extends WordSpec
      with MustMatchers with MockFactory {

  "MyTickerTape" should {
    "return a map of my favorite symbols as keys, and their" +
     "price as values" in {
       val sqp = mock[StockQuoteProvider]
       sqp expects 'getPrice withArgs ("ATT") returning
                   (BigDecimal("33.00"))
       sqp expects 'getPrice withArgs ("ADVS") returning
                   (BigDecimal("12.34"))
       sqp expects 'getPrice withArgs ("MTN") returning
                   (BigDecimal(".22"))

       val tickerTape = new MyTickerTape(sqp, List("ATT", "ADVS", "MTN"))
       tickerTape.getPrices("ATT") must be(BigDecimal(33.00))
       tickerTape.getPrices("ADVS") must be(BigDecimal(12.34))
       tickerTape.getPrices("MTN") must be(BigDecimal(.22))
    }
  }
}
```

Borachio

```
class EmployeeScalaTestWithBorachio extends WordSpec
      with MustMatchers with MockFactory {

  "MyTickerTape" should {
    "return a map of my favorite symbols as keys, and their" +
     "price as values" in {
       val sqp = mock[StockQuoteProvider]
       sqp expects 'getPrice withArgs ("ATT") returning
                   (BigDecimal("33.00"))
       sqp expects 'getPrice withArgs ("ADVS") returning
                   (BigDecimal("12.34"))
       sqp expects 'getPrice withArgs ("MTN") returning
                   (BigDecimal(".22"))

       val tickerTape = new MyTickerTape(sqp, List("ATT", "ADVS", "MTN"))
       tickerTape.getPrices("ATT") must be(BigDecimal(33.00))
       tickerTape.getPrices("ADVS") must be(BigDecimal(12.34))
       tickerTape.getPrices("MTN") must be(BigDecimal(.22))
    }
  }
}
```

Borachio

```
class EmployeeScalaTestWithBorachio extends WordSpec
      with MustMatchers with MockFactory {

  "MyTickerTape" should {
    "return a map of my favorite symbols as keys, and their" +
     "price as values" in {
       val sqp = mock[StockQuoteProvider]
       sqp expects 'getPrice withArgs ("ATT") returning
                   (BigDecimal("33.00"))
       sqp expects 'getPrice withArgs ("ADVS") returning
                   (BigDecimal("12.34"))
       sqp expects 'getPrice withArgs ("MTN") returning
                   (BigDecimal(".22"))

       val tickerTape = new MyTickerTape(sqp, List("ATT", "ADVS", "MTN"))
       tickerTape.getPrices("ATT") must be(BigDecimal(33.00))
       tickerTape.getPrices("ADVS") must be(BigDecimal(12.34))
       tickerTape.getPrices("MTN") must be(BigDecimal(.22))
    }
  }
}
```

Borachio

```
class EmployeeScalaTestWithBorachio extends WordSpec
      with MustMatchers with MockFactory {

  "MyTickerTape" should {
    "return a map of my favorite symbols as keys, and their" +
     "price as values" in {
       val sqp = mock[StockQuoteProvider]
       sqp expects 'getPrice withArgs ("ATT") returning
                   (BigDecimal("33.00"))
       sqp expects 'getPrice withArgs ("ADVS") returning
                   (BigDecimal("12.34"))
       sqp expects 'getPrice withArgs ("MTN") returning
                   (BigDecimal(".22"))

       val tickerTape = new MyTickerTape(sqp, List("ATT", "ADVS", "MTN"))
       tickerTape.getPrices("ATT") must be(BigDecimal(33.00))
       tickerTape.getPrices("ADVS") must be(BigDecimal(12.34))
       tickerTape.getPrices("MTN") must be(BigDecimal(.22))
    }
  }
}
```

Borachio (with Sequence)

```
class EmployeeScalaTestWithBorachio extends WordSpec
      with MustMatchers with MockFactory {

  "MyTickerTape" should {
    "MyTickerTape" should {
       "return a map of my favorite symbols as keys, and their price as
           values, in the same order as it was given " in {

         val sqp = mock[StockQuoteProvider]
         inSequence {
           sqp expects 'getPrice withArgs ("ATT") returning
                (BigDecimal("33.00"))
           sqp expects 'getPrice withArgs ("ADVS") returning
                (BigDecimal("12.34"))
           sqp expects 'getPrice withArgs ("MTN") returning
                (BigDecimal(".22"))
         }

         val tickerTape = new MyTickerTape(sqp, List("ATT", "ADVS", "MTN"))
         val prices = tickerTape.getPrices
         prices("ATT") must be(BigDecimal(33.00))
         prices("ADVS") must be(BigDecimal(12.34))
         prices("MTN") must be(BigDecimal(.22))
    }
  }
}
```

Borachio (with Sequence)

```
class EmployeeScalaTestWithBorachio extends WordSpec
      with MustMatchers with MockFactory {

  "MyTickerTape" should {
    "MyTickerTape" should {
       "return a map of my favorite symbols as keys, and their price as
           values, in the same order as it was given " in {

         val sqp = mock[StockQuoteProvider]
         inSequence {
           sqp expects 'getPrice withArgs ("ATT") returning
                (BigDecimal("33.00"))
           sqp expects 'getPrice withArgs ("ADVS") returning
                (BigDecimal("12.34"))
           sqp expects 'getPrice withArgs ("MTN") returning
                (BigDecimal(".22"))
         }

         val tickerTape = new MyTickerTape(sqp, List("ATT", "ADVS", "MTN"))
         val prices = tickerTape.getPrices
         prices("ATT") must be(BigDecimal(33.00))
         prices("ADVS") must be(BigDecimal(12.34))
         prices("MTN") must be(BigDecimal(.22))
    }
  }
}
```

Borachio (Functional Testing)

```
"Borachio" should {
    "mock functions by their signature" in {
      val tickerTapeFunction =
           mockFunction[String, BigDecimal]

      tickerTapeFunction expects ("ATT") returning
             BigDecimal("33.00")
      tickerTapeFunction expects ("ADVS") returning
             BigDecimal("12.34")
      tickerTapeFunction expects ("MTN") returning
             BigDecimal(".22")

      expect(List
          (BigDecimal("33.00"), BigDecimal("12.34"),
               BigDecimal(".22"))) {
        List("ATT", "ADVS", "MTN").map(tickerTapeFunction)
      }
    }
  }
```

Borachio (Functional Testing)

```
"Borachio" should {
    "mock functions by their signature" in {
      val tickerTapeFunction =
           mockFunction[String, BigDecimal]

      tickerTapeFunction expects ("ATT") returning
             BigDecimal("33.00")
      tickerTapeFunction expects ("ADVS") returning
             BigDecimal("12.34")
      tickerTapeFunction expects ("MTN") returning
             BigDecimal(".22")

      expect(List
          (BigDecimal("33.00"), BigDecimal("12.34"),
               BigDecimal(".22"))) {
        List("ATT", "ADVS", "MTN").map(tickerTapeFunction)
      }
    }
  }
```

Borachio (Functional Testing)

```
"Borachio" should {
    "mock functions by their signature" in {
      val tickerTapeFunction =
           mockFunction[String, BigDecimal]

      tickerTapeFunction expects ("ATT") returning
             BigDecimal("33.00")
      tickerTapeFunction expects ("ADVS") returning
             BigDecimal("12.34")
      tickerTapeFunction expects ("MTN") returning
             BigDecimal(".22")

      expect(List
          (BigDecimal("33.00"), BigDecimal("12.34"),
               BigDecimal(".22"))) {
        List("ATT", "ADVS", "MTN").map(tickerTapeFunction)
      }
    }
  }
```

Borachio (Functional Testing)

```
"Borachio" should {
    "mock functions by their signature" in {
      val tickerTapeFunction =
           mockFunction[String, BigDecimal]

      tickerTapeFunction expects ("ATT") returning
             BigDecimal("33.00")
      tickerTapeFunction expects ("ADVS") returning
             BigDecimal("12.34")
      tickerTapeFunction expects ("MTN") returning
             BigDecimal(".22")

      expect(List
          (BigDecimal("33.00"), BigDecimal("12.34"),
               BigDecimal(".22"))) {
        List("ATT", "ADVS", "MTN").map(tickerTapeFunction)
      }
    }
  }
```

Borachio (Functional Testing)

```
"Borachio" should {
    "mock functions by their signature" in {
      val tickerTapeFunction =
           mockFunction[String, BigDecimal]

      tickerTapeFunction expects ("ATT") returning
             BigDecimal("33.00")
      tickerTapeFunction expects ("ADVS") returning
             BigDecimal("12.34")
      tickerTapeFunction expects ("MTN") returning
             BigDecimal(".22")

      expect(List
          (BigDecimal("33.00"), BigDecimal("12.34"),
               BigDecimal(".22"))) {
        List("ATT", "ADVS", "MTN").map(tickerTapeFunction)
      }
    }
  }
```

Borachio (Functional Testing)

```
"Borachio" should {
    "mock functions by their signature" in {
      val tickerTapeFunction =
           mockFunction[String, BigDecimal]

      tickerTapeFunction expects ("ATT") returning
             BigDecimal("33.00")
      tickerTapeFunction expects ("ADVS") returning
             BigDecimal("12.34")
      tickerTapeFunction expects ("MTN") returning
             BigDecimal(".22")

      expect(List
          (BigDecimal("33.00"), BigDecimal("12.34"),
               BigDecimal(".22"))) {
        List("ATT", "ADVS", "MTN").map(tickerTapeFunction)
      }
    }
  }
```

Questions?