Building Scala

Building Scala, Using Scala with Java

Agenda

1. Scala Build Tools
2. SBT Intro
3. SBT Commands
4. Customizing SBT
5. Mixing Java and Scala
6. Java 8 Support (Lambdas, etc.)
7. Java Collections, Primitives and Nulls

Maven

- Uses Zinc for incremental compilation
- Strong polyglot languages support
- Fast to create a new Scala project:

```
mvn archtype:generate
mvn clean test
```

- Many archetypes, including Scala, Akka, Play, etc.

Gradle

- Scala plugin: `apply plugin: 'scala'`
- DSL in Groovy
- Support for Zinc:

```
tasks.withType(ScalaCompile) {
  scalaCompileOptions.useAnt = false
}
```

- Dependency management:

```
dependencies {
  testCompile "org.scala-lang:scala-library:2.12.1"
}
```

Other Options

- Pants: Twitter’s Open Source Build Tool
    - Python DSL
    - http://pantsbuild.github.io/
- Apache Buildr
    - Ruby DSL
    - http://buildr.apache.org/
- Ant
    - XML DSL :-)
    - http://tutorials.jenkov.com/scala/compiling-with-ant.html
- sbt

SBT

- Written in Scala, includes Scala like DSL
- Officially, name means nothing
- Fast Compile/Test, also Continuous
- https://scala-sbt.org

Using SBT

- Interactive mode
  - `help` & `tasks`
- Common commands:
  - `clean`
  - `compile`
  - `project` (for multiple project builds)
  - `test` & `test:compile`
  - `publish`, `publish-local` & `publish-signed`
  - `console` & `test:console`
- ~ commands
- Create a new project, e.g.: `sbt new scala/scala-seed.g8`

SBT Project Source Layout

- Same as Maven defaults

```
src/
  main/
    resources/
       <files to include in main jar here>
    scala/
       <main Scala sources>
    java/
       <main Java sources>
  test/
    resources
       <files to include in test jar here>
    scala/
       <test Scala sources>
    java/
       <test Java sources>
```

- Play projects do things a little differently

build.sbt

- Easiest way in to sbt
- Scala like DSL, simplified dialect
- Now only 3 main operators to learn:

```
:=            -   set a value
+=            -   add a value to existing
++=           -   add a sequence of values to existing
```

- Blank lines between expressions are now optional
- Can embed any standard Scala code in {}s

Example build.sbt

```
name := """scala-library-seed"""

organization := "com.example"

licenses += ("MIT", url("http://opensource.org/licenses/MIT"))

javacOptions ++= Seq("-source", "1.6", "-target", "1.6")

scalaVersion := "2.10.4"

crossScalaVersions := Seq("2.10.4", "2.11.2", "2.12.1")

libraryDependencies ++= Seq(
  "org.scalatest" %% "scalatest" % "2.2.1" % "test"
)

bintraySettings

com.typesafe.sbt.SbtGit.versionWithGit
```

Example plugins.sbt

```
resolvers += Resolver.url(
  "bintray-sbt-plugin-releases",
  url("http://dl.bintray.com/content/sbt/sbt-plugin-releases"))(
    Resolver.ivyStylePatterns)

addSbtPlugin("me.lessis" % "bintray-sbt" % "0.1.2")

resolvers += "jgit-repo" at "http://download.eclipse.org/jgit/maven"

addSbtPlugin("com.typesafe.sbt" % "sbt-git" % "0.6.4")
```

Making a Custom Setting

In `build.sbt`:

```scala
val isAwesome = settingKey[Boolean]("Some boolean setting")

isAwesome := true

val totally = settingKey[String]("rating of totalness of the statement")

totally := "100% totally"

val totallyAwesome = settingKey[String]("How awesome is this project")

totallyAwesome := totally.value + {
  println("Checking project awesomeness")
  if (isAwesome.value) " awesome." else " not awesome."
}
```

And a Custom Task

```
val checkAwesome = taskKey[Unit]("Check project awesomeness")

checkAwesome := {
  val _ = (compile in Test).value  // force the test:compile task
  println("The project is " + totallyAwesome.value)
}
```

- Settings are evaluated once per sbt run (like a val)
- Tasks are evaluated once per `build` operation (like a def)

Multiple Project Support

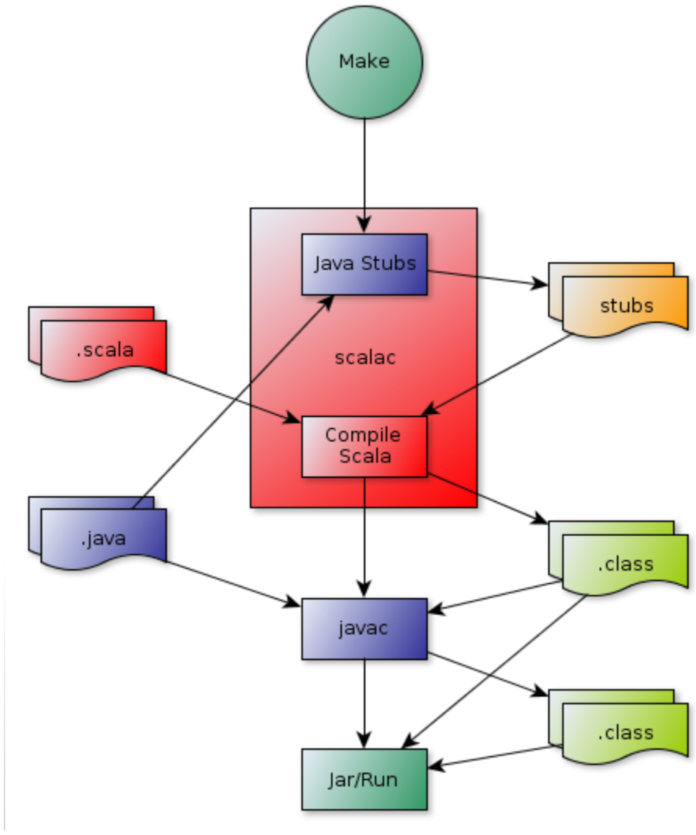
- Multiple projects can be defined in the same build.sbt file
- Assign `project` definitions to vals that represent the projects
- By default the name of the project val will define the folder used, this may be overridden

```
lazy val util = project

lazy val extras = project

lazy val prod = project.dependsOn(util, extras)

lazy val root = project.in(file("."))
  .aggregate(util, extras, prod)
  .settings(aggregate in update := false)
```

Mixing Scala and Java

- And using Java libraries
- Scala can use any Java library
- But some make more sense than others
- This includes persistence layers, web frameworks, numerical and scientific libraries, etc.
- On the other hand, many of these have Scala wrappers already if they are popular

Scala/Java Compile Cycle



Calling Java from Scala

- Import any Java library
- Call Java methods just like Scala
- Can leave off ()s for empty params
- Can call using infix notation
- Can extend or "with" Java interfaces
- Can instantiate Java classes
- Scala handles conversion to/from primitives
- In Scala 2.12, SAMs can be satisfied with Scala functions

Scala 2.12 and Java 8

- Scala 2.12 requires Java 8
- Scala function literals compile to method handles
- Scala supports SAMs - Single Abstract Methods
- FunctionN are Java FunctionlInterfaces
- @interface to guarantee trait can be used from Java
- Java 8 Streams API also available to use

Java 8 / Scala 2.12 Function Compatibility

```scala
case class Person(name: String, age: Int)

import java.util.{Arrays, Comparator}

val javaArray = Array(Person("Harry", 25), Person("Sally", 22), Person("Tim", 33))

// the old SAM way
val comp1 = new Comparator[Person] {
  override def compare(o1: Person, o2: Person) = o1.age - o2.age
}

Arrays.sort(javaArray, comp1)

javaArray
// Array(Person(Sally,22), Person(Harry,25), Person(Tim,33))

// the new Java Lambda way
Arrays.sort(javaArray, (p1: Person, p2: Person) => p2.age - p1.age)

javaArray
// Array(Person(Tim,33), Person(Harry,25), Person(Sally,22))
```

Handling Nulls

- Nulls are discouraged in Scala
- But they are a fact of life in Java libraries

```
val a = javaObj.methodCanReturnNull(x)

a.toString                           // oops
// java.lang.NullPointerException

val b = Option(javaObj.methodCanReturnNull(x))

b.map(_.toString)                   // safe
None
```

- Wrapping an object from Java in `Option` will convert a reference to `Some` and a `null` to `None`

Options to Nulls

- Going back the other way is also easy, imagine a Java method that accepts nulls

```
val s1: String = "hello"
val s2: String = null // no no no no no

val os1 = Option(s1)
val os2 = Option(s2)

os1.orNull   // "hello"
os2.orNull   // null
```

- `orNull` on `Option` returns either the contents or a null reference (if None)

Java -> Scala Collections

```
val jl = new java.util.ArrayList[Int]
jl.add(1); jl.add(2); jl.add(3)

jl.map(_ * 2)
// error: value map is not a member of java.util.ArrayList[Int]
//        jl.map(_ * 2)

scala> import scala.collection.JavaConverters._

scala> jl.asScala.map(_ * 2)
res1: scala.collection.mutable.Buffer[Int] = ArrayBuffer(2, 4, 6)
```

- **Note:** `scala.collection.JavaConverters._` should be used, **not** `scala.collection.JavaConversions._`

Boxed Types Trouble

- Sometimes `JavaConverters` on their own is not enough:

```
// Java method signature:
public List<Integer> someJavaFunc(List<Integer> list) { ... }

val sl = List(1, 2, 3)

// Int is not an Integer!
val r1 = someJavaFunc(sl.asJava)
// Error:(28, 27) type mismatch;
// found    : java.util.List[Int]
// required: java.util.List[Integer]

val jl2 = sl.map( new java.lang.Integer(_) ) // explicitly box first
// jl2: List[Integer] = List(1, 2, 3)

val r2 = someJavaFunc(jl2.asJava)
// r2: java.util.List[Integer] = [1, 2, 3]
```

Scala Traits and Java Interfaces

- Java interfaces can be used as pure abstract Scala traits
- Conversely pure abstract Scala traits can be used as Java interfaces:

```
// Scala

trait DoSomethingToString {
  def doIt(s: String): String
}

// Java
class Shout implements DoSomethingToString {
  public String doIt(String s) {
    return s.toUpperCase();
  }
}
```

- Pure Scala traits may be the best solution for Java calling Scala

General Advice

- Java calling Scala
    - Provide empty trait based API around Scala implementation
    - Avoid function literals
    - Convert between nullable and Option
- Scala calling Java
    - Remember scala.collection.JavaConverters
    - Use implicit conversions (respectfully)
    - Remember the REPL

Exercises for Module 15

- Find the `Module15` class and follow the instructions to make the tests pass
- Some of the examples for this exercise are under src/test/java and are Java code