Table of Contents for Programming Scala, 2nd Edition

Chapter 21. Scala Tools and Libraries

This chapter fills in some details about the Scala tools we've already used, such as the compiler `scalac` and the REPL `scala`. We'll discuss build tool options and IDE and text editor integration, and look at testing libraries for Scala. Finally, we'll list some of the popular third-party Scala libraries you might find useful.

Command-Line Tools

Even if you do most of your work with IDEs or the SBT REPL, understanding how the command-line tools work gives you additional flexibility, as well as a fallback should the graphical tools fail you. Most of the time, you'll configure compiler flags through your `SBT` build files or IDE settings, and you'll invoke the REPL through your `SBT` session, using the `console` command.

Installing Scala described how to install the command-line tools. All of them are located in the *SCALA_HOME/bin* directory, where *SCALA_HOME* is the directory where you installed Scala.

You can read more information about the command-line tools at http://www.scala-lang.org/documentation/.

scalac Command-Line Tool

The `scalac` command compiles Scala source files and generates JVM class files.

The `scalac` command is just a shell-script wrapper around the `java` command, passing it the name of the Scala compiler's `Main` object. It adds Scala JAR files to the `CLASSPATH` and it defines several Scala-related system properties.

You invoke `scalac` like this:

```
scalac <options> <source
files>
```

Recall from A Taste of Scala that source file names don't have to match the public class name in the file. In fact, you can define multiple public classes in a file. Similarly, package declarations don't have to match the directory structure.

However, in order to conform to JVM requirements, a separate class file will be generated for each type with a name that corresponds to the type's name. Also, the class files will be written to directories corresponding to the package declarations.

Table 21-1 shows the list of the `scalac` options as reported by `scalac -help` for the 2.11.2 compiler (edited slightly).

Table 21-1. The scalac command options

Option	Description
`-Dproperty=value`	Pass `-Dproperty=value` directly to the runtime system.
`-Jflag`	Pass Java flag directly to the runtime system.
`-Pplugin:opt`	Pass an option to a compiler plug-in.
`-X`	Print a synopsis of advanced options (Table 21-3 discusses these advanced options).
`-bootclasspath path`	Override location of bootstrap class files.
`-classpath path`	Specify where to find user class files.
`-d directory or jar`	Destination for generated class files.
`-dependencyfile file`	Specify the file in which dependencies are tracked.
`-deprecation`	Output source locations where deprecated APIs are used.
`-encoding encoding`	Specify character encoding used by source files.
`-explaintypes`	Explain type errors in more detail.
`-extdirs dirs`	Override location of installed compiler extensions.
`-feature`	Emit warning and location for usages of features that should be imported explicitly.
`-g:level`	Specify `level` of generated debugging info: `none`, `source`, `line`, `vars` (default), `notailcalls`.
`-help`	Print a synopsis of standard options.
`-javabootclasspath path`	Override the Java boot classpath.
`-javaextdirs path`	Override the Java `extdirs` classpath.
`-language:feature`	Enable one or more language features: `dynamics`, `postfixOps`, `reflectiveCalls`, `implicitConversions`, `higherKinds`, `existentials`, and `experimental.macros` (comma-separated list, with no spaces).
`-no-specialization`	Ignore occurrences of the `@specialize` annotations.
`-nobootcp`	Do not use the boot classpath for the Scala JARs.
`-nowarn`	Generate no warnings.
`-optimise`	Generate faster byte code by applying optimizations to the program.

Option	Description
`-print`	Print program with all Scala-specific features removed.
`-sourcepath path`	Specify where to find input source files.
`-target:target`	Target platform for object files: `jvm-1.5` (deprecated), `jvm-1.6` (default), `jvm-1.7`.
`-toolcp path`	Add to the runner classpath.
`-unchecked`	Enable additional warnings where generated code depends on assumptions.
`-uniqid`	Uniquely tag all identifiers in debugging output.
`-usejavacp`	Utilize the `java.class.path` in classpath resolution.
`-usemanifestcp`	Utilize the manifest in classpath resolution.
`-verbose`	Output messages about what the compiler is doing.
`-version`	Print product version and exit.
`@file`	A text file containing compiler arguments (options and source files).

Let's discuss a few of these options.

`-encoding`
Use `UTF8` if you use non-ASCII characters in names or the allowed symbols, such as ⇒ (Unicode \u21D2) instead of `=>`.

Use `-explaintypes` when you need a more complete explanation for a type error.

Starting with Scala 2.10, some more advanced language features have been made optional, so that teams can selectively enable the ones they want to use. This is part of an ongoing effort to address complexity concerns about Scala, while still allowing advanced constructs to be used when desired. Use `-feature` to emit a warning and the source location for any usage of these features when the corresponding import statement is missing in the source code and the corresponding `-language:feature` compiler flag wasn't used.

The list of optional language features is defined by values in the `scala.language` object. Its Scaladoc page also explains why these features are optional. Table 21-2 lists the features.

Table 21-2. The optional language features

Name	Description
`dynamics`	Enables the Dynamic trait (see Chapter 19).
`postfixOps`	Enables postfix operators (e.g., `100 toString`).
`reflectiveCalls`	Enables using structural types (see Reflecting on Types).
`implicitConversions`	Enables defining implicit methods and members (see Implicit Conversions).
`higherKinds`	Enables writing higher-kinded types (see Higher-Kinded Types).

Name	Description
`existentials`	Enables writing existential types (see Existential Types).
`experimental`	Contains newer features that have not yet been tested in production. Macros are the only experimental feature in Scala at this time (see Macros).

The advanced `-X` options (printed by `scalac -X` and `scala -X`) control verbose diagnostic output, fine-tune the compiler behavior, control use of experimental extensions and plug-ins, etc. Table 21-3 discusses a few of these options.

Table 21-3. Some of the `-X` advanced options

Option	Description
`-Xcheckinit`	Wrap field accessors to throw an exception on uninitialized access (see Overriding fields in traits).
`-Xdisable-assertions`	Generate no assertions or assumptions.
`-Xexperimental`	Enable experimental extensions.
`-Xfatal-warnings`	Fail the compilation if there are any warnings.
`-Xfuture`	Turn on “future” language features (if any for a particular release).
`-Xlint`	Enable recommended additional warnings.
`-Xlog-implicit-conversions`	Print a message whenever an implicit conversion is inserted.
`-Xlog-implicits`	Show more detail on why some implicits are not applicable.
`-Xmain-class path`	Specify the class to use for the `Main-Class` entry in the JAR file’s manifest. Only useful with `-d jar` option.
`-Xmigration:v`	Warn about constructs whose behavior may have changed since version *v* of Scala.
`-Xscript object`	Treat the source file as a script and wrap it in a main method.
`-Y`	Print a synopsis of *private* options, which are used by implementers of new language features.

To see an example of the extra warnings provided by `-Xlint`, consider the following, where we first start the REPL with the `-Xlint` option:

```
$ scala -Xlint
Welcome to Scala version 2.11.2 ...
...
scala>  def hello = println("hello!")
<console>:7: warning: side-effecting nullary methods are discouraged:
                      `def
  suggest defining as hello()`      instead
       def hello = println("hello!")
           ^
hello: Unit
```

Any function that returns `Unit` can only perform side effects. In this case, we write output. It's a common convention in Scala code to only use *nullary* methods, those with no argument list, for functions without side effects. Hence, this warning.

The `-Xscript` option is useful when you want to compile a script file as if it were a regular Scala source file, usually to eliminate the startup overhead of compiling the script repeatedly.

Tip

I recommend routine use of the `-deprecation`, `-unchecked`, `feature`, and `-Xlint` options. (The `-Xlint` option may throw too many warnings for some code bases.) They help prevent some bugs and encourage you to eliminate use of obsolete libraries. We use these flags, as well as a few others, in the *build.sbt* file in the code examples.

The scala Command-Line Tool

The `scala` command runs the program, if given. Otherwise, it starts the REPL. It is also a shell script, like `scalac`. You invoke `scala` like this:

```
scala <options> [<script|class|object|jar>
<arguments>]
```

The same options accepted by `scalac` apply here, plus the additional options shown in Table 21-4.

Table 21-4. The additional scala command options

Option	Description
`-howtorun`	What to run, a `script`, `object`, `jar`, or make a `guess` (default).
`-i file`	Preload the contents of `file` before starting the REPL.
`-e string`	Execute `string` as if it were entered in the REPL.
`-save`	Save the compiled script in a JAR file for future use, thereby avoiding the overhead of recompilation.
`-nc`	Don't run the compilation daemon, `fsc`, the offline compiler that is otherwise started automatically to avoid the overhead of restarting the compiler each time.

The first nonoption argument is interpreted as the program to run. If nothing is specified, the REPL is started. When specifying a program, any arguments after the program argument will be passed to it in the `args` array we've used previously in various examples.

Unless you specify the `-howtorun` option, `scala` will infer the nature of the program. If it is a file of Scala source code, it will be executed as a script. If it is class file with a `main` routine or a JAR file with a valid `Main-Class` attribute, it will be executed as a typical Java program.

Use the `-i file` option in the interactive mode when you want to preload a file before typing commands. Once in the REPL, you can also load a file using the command `:load filename`. Such a file is useful when you find yourself repeating the same commands when you start a REPL.

When using the REPL, you have several commands at your disposal. Enter `:help` to see a list of them with brief descriptions. Table 21-5 lists the available commands for Scala 2.11.2, edited slightly.

Table 21-5. Commands available within the Scala REPL

Command	Description
`:cp path`	Add a JAR or directory to the classpath.
`:edit id or line`	Edit the input history.
`:help [command]`	Print this summary or command-specific help.
`:history [num]`	Show the history (optional `num` is the number of commands to show).
`:h? string`	Search the history.
`:imports [name name …]`	Show the import history, identifying sources of names.
`:implicits [-v]`	Show the implicits in scope (optional `-v` for more verbose output).
`:javap path or class`	Disassemble a file or class name.
`:line id or line`	Place lines at the end of history.
`:load path`	Interpret lines in a file given by `path`.
`:paste [-raw] [path]`	Enter paste mode or paste a file.
`:power`	Enable power user mode (see the text that follows).
`:quit`	Exit the interpreter (or use Ctrl-D).
`:replay`	Reset execution and replay all previous commands.
`:reset`	Reset the REPL to its initial state, forgetting all session entries.
`:save path`	Save the session to a file for replaying later.

Command	Description
`:sh command line`	Run a shell command (result is implicitly ⇒ `List[String]`).
`:settings [+ or - ]options`	Enable (`+`)/disable(`-`) flags, set compiler options.
`:silent`	Disable/enable automatic printing of results.
`:type [-v] expr`	Display the type of an expression without evaluating it.
`:kind [-v] expr`	Display the kind of an expression's type.
`:warnings`	Show the suppressed warnings from the most recent line that had any warnings.

The "power user mode" enabled by `:power` adds additional commands for viewing in-memory data, such as the abstract syntax tree and interpreter properties, and for manipulating the compiler.

Invoking scripts with `scala` is tedious when you use these scripts frequently. On Windows and Unix-like systems, you can create standalone Scala scripts that don't require you to use the `scala script-file-name` invocation.

For Unix-like systems, the following example demonstrates how to make an executable script. Remember that you have to make the permissions executable, e.g., `chmod +x secho` :

```
#!/bin/sh
# src/main/scala/progscala2/toolslibs/secho
exec scala "$0" "$@"
!#
print("You entered: ")
args.toList foreach { s => printf("%s ", s)
}
println
```

Here is how you might use it:

```
$ secho Hello World
You entered: Hello
World
```

Similarly, here is an example Windows `.bat` command:

```
::#!
@echo off
call scala %0 %*
goto :eof
::!#
print("You entered: ")
args.toList foreach { s => printf("%s ", s)
}
println
```

Limitations of scala versus scalac

There are some limitations when running a source file with `scala` versus compiling it with `scalac`.

A script executed with `scala` is wrapped in an anonymous `object` that looks more or less like the following example:

```
object Script {
  def main(args: Array[String]): Unit = {
    new AnyRef {

// Your script code is inserted
here.
    }
  }
}
```

Scala `objects` cannot embed package declarations, which means you can't declare packages in scripts. This is why the examples in this book that declare packages must be compiled and executed separately.

Conversely, there are valid scripts that can't be compiled with `scalac`, unless the `-Xscript object` option is used, where `object` will be the name of the compiled object, replacing the word `Script` in the previous example. In other words, this compiler option creates the same wrapper that the REPL does implicitly.

An object wrapper is required for compilation because function definitions and function invocations outside of types are not allowed. The following example runs fine with `scala` as a script:

```
// src/main/scala/progscala2/toolslibs/example.sc

case class Message(name: String)

def printMessage(msg: Message) = println(msg)

                        "This works fine with the
printMessage(new Message(REPL"                          ))
```

However, if you try to compile the script with `scalac` without the `-Xscript` option, you get the following errors:

```
example.sc:3: error: expected class or object definition
def printMessage(msg: Message) = println(msg)
^
example.sc:5: error: expected class or object definition
printMessage(new Message("This works fine with the
REPL"))
^
two errors found
```

Instead, compile it and run it this way:

```
scalac -Xscript MessagePrinter
src/main/scala/progscala2/toolslibs/example.sc
scala  -classpath . MessagePrinter
```

Because the script effectively uses the default package, the generated class files will be in the current directory:

```
MessagePrinter$$anon$1$Message$.class
MessagePrinter$$anon$1$Message.class
MessagePrinter$$anon$1.class
MessagePrinter$.class
MessagePrinter.class
```

Try running `javap -private` (discussed in the next section) on each of these files to see the declarations they contain. The `-p` flag tells it to show all members, including private and protected members (use `javap -help` to see all the options). Omit the `.class`, e.g., `javap MessagePrinter$$anon$1$Message$`.

`MessagePrinter` and `MessagePrinter$` are wrappers generated by `scalac` to provide the entry point for the script as an "application." `MessagePrinter` has the `static main` method we need.

`MessagePrinter$$anon$1` is the generated Java class that wraps the whole script. The `printMessage` method in the script is a *private* method in this class.

`MessagePrinter$$anon$1$Message` is the `Message` class.

`MessagePrinter$$anon$1$Message$` is the `Message` companion object.

The scalap and javap Command-Line Tools

Decompilers are useful when you want to understand how Scala constructs are implemented in JVM byte code. They help you understand how Scala names are *mangled* into JVM-compatible names, when necessary.

The venerable `javap` comes with the JDK. It outputs declarations as they would appear in Java source code, even for class files that were compiled from Scala code by `scalac`. Therefore, running `javap` on these class files is a good way to see how Scala definitions are mapped to valid byte code.

The Scala distribution comes with `scalap`, which outputs declarations as they would appear in Scala source code.

However, the Scala 2.11.0 and 2.11.1 distributions omitted `scalap` by mistake. You can download the 2.11.1 JAR file here. Just copy it to the *lib* directory of your installation. Version 2.11.2 includes `scalap`.

Using *MessagePrinter.class* from the previous section, run `scalap -cp . MessagePrinter`. You should get the following output (reformatted to fit the page):

```
object MessagePrinter extends scala.AnyRef {
                  /* compiled code
  def this() = { */                   }
  def main(args : scala.Array[scala.Predef.String]) : scala.Unit = {
    /* compiled code
    */
  }
}
```

Compare this output with the output from `javap -cp . MessagePrinter`:

```
Compiled from "example.sc"
public final class MessagePrinter {
  public static void main(java.lang.String[]);
}
```

Now we see the declaration of `main` as we would typically see it in a Java source file.

Both these tools have a `-help` option that describes the invocation options they support.

As an exercise, try decompiling the class files generated from *progscala2/toolslibs/Complex.scala*, which implements `Complex` numbers. It has already been compiled by SBT. Try running `scalap` and `javap` on the resulting class files. Note how the declared package name `toolslibs` is specified. The class file built with Scala 2.11 can be decompiled as follows with `javap`:

```
javap -cp target/scala-2.11/classes toolslibs.Complex
```

How are the `+` and `-` method names encoded? What are the names of the "getter" methods for the `real` and `imaginary` fields? What Java types are used for these fields? What is the output produced by `scalap` and `javap`?

The scaladoc Command-Line Tool

The `scaladoc` command is analogous to `javadoc`. It is used to generate documentation from Scala source files, called Scaladocs. The `scaladoc` parser supports the same `@` annotations that `javadoc` supports, such as `@author`, `@param`, etc.

The easiest way to use `scaladoc` for your project is to run the `doc` task in SBT.

The fsc Command-Line Tool

The *fast scala compiler* runs as a daemon process to enable faster invocations of the compiler, mostly by

eliminating the startup overhead. It is particularly useful when running scripts repeatedly (for example, when rerunning a test suite until a bug can be reproduced). In fact, `fsc` is invoked automatically by the `scala` command. You can also invoke it directly.

Build Tools

Most new projects use SBT as their build tool, so we'll focus our discussion on it. However, Scala plug-ins have been implemented for several other build tools, including Ant, Maven (mvn), and Gradle.

SBT, the Standard Build Tool for Scala

is a sophisticated tool for building Scala and Java projects. It has lots of configuration options and plug-in capabilities. We've been using it all along for the code examples. Let's look at its features in a bit more detail, including the structure of the actual build file for the code. However, we'll just scratch the surface (see the SBT documentation and *SBT in Action*, by Joshua Suereth and Matthew Farwell (Manning) for more details).

By now you've installed SBT. If you do JVM-based web development, see also the new `sbt-web` project, which adds plug-ins for building and managing web assets such as HTML pages and CSS files from template languages.

Tip

The *fastest* way to get started with SBT is to copy and edit an existing build. For example, start with the build files in one of the Activator Templates.

SBT is similar to Maven in the sense that many of the tasks you'll need to do, like compiling and automated testing, are already built in, along with suitable dependencies between tasks, like compiling before testing. The build files in SBT are used to define project metadata, such as the name and version for releases, define dependencies using the Maven conventions and repositories (but using Ivy as the dependency resolution tool), and perform other customizations. A Scala-based DSL is used as the language.

SBT builds are defined by one or more build files, depending on the sophistication and customization required for your project. For the book's code examples, the *project* is relatively simple, although supporting two versions of Scala adds a little complexity.

The main build file is in the root directory of the code examples. It is named *build.sbt*. There are two additional files in the *project* subdirectory: *build.properties*, which defines the version of SBT we want to use, and *plugins.sbt*, which adds SBT plug-ins for generating Eclipse project files. (IntelliJ IDEA can import SBT projects directly.) It's also common to put *build.sbt* in the *project* directory.

Let's look at a simplified version of our *build.sbt*, starting with some definitions:

```
        "Programming Scala, Second Edition: Code
name := examples"

version := "2.0"

organization := "org.programming-scala"

scalaVersion := "2.11.2"

          name := "Programming Scala,
```

Definitions like … define variables. The DSL currently requires a blank line

between each definition, to make it easier to infer the end of the definition. If you forget a blank line, you'll get an error message that suggests the mistake.

Here are the dependencies defined in the file (many are elided):

```
libraryDependencies ++= Seq(
  "com.typesafe.akka"      %% "akka-actor"      % "2.3.4",
  "org.scalatest"          %% "scalatest"       % "2.2.1"  %
"test",
  "org.scalacheck"         %% "scalacheck"      % "1.11.5" %
"test",
  ...
)
```

Sometimes a sequence, `Seq`, is required for a definition, such as the list of dependencies we need, the *Akka actor* library, version 2.3.4, *ScalaTest* and *ScalaCheck* (see Test-Driven Development in Scala), as well as many others we've elided.

Not shown in this file are definitions of Maven-compatible repositories on the Internet to find these dependencies. SBT already knows a list of standard locations, but you can define custom ones, too. There are examples of repository specifications in the *project/plugins.sbt* file. See the definition of `resolvers`.

The rest of this definition of `libraryDependencies`, of which a few of the actual definitions, are shown here.

Finally, compiler flags are defined for `scalac` and `javac`:

```
scalacOptions = Seq(
  "-encoding", "UTF-8", "-optimise",
  "-deprecation", "-unchecked", "-feature", "-Xlint", "-Ywarn-infer-any")

javacOptions  ++= Seq("-Xlint:unchecked", "-Xlint:deprecation")
```

Not used in our file, but useful to know, is the ability to define multiple statements that are executed automatically when the REPL starts `console`, for example:

```
initialCommands in console := """
  |import
foo.bar._
  |import
foo.bar.baz._

|"""  .stripMargin
```

These are analogous to the `-i file` option for `scala` discussed earlier. There are two other variants of `console`. The first is `consoleQuick` (you can also type `console-quick`), which does not compile your code first. This is useful when you want to try out something but the code isn't currently building (or it will take a long time).

The other variant is `consoleProject` (or `console-project`), which ignores your code, but loads with the SBT and build definition on the `CLASSPATH`, along with some useful imports.

The `initialCommands in console` also apply to `consoleQuick`, but you can also define a custom value. In contrast, the `initialCommands in console` are not used for `consoleProject`, but you can define a custom value:

```
initialCommands in console := """println("Hello from console")"""
initialCommands in consoleQuick := """println("Hello from consoleQuick")"""
initialCommands in consoleProject := """println("Hello from consoleProject")"""
```

There are corresponding `cleanupCommands`, too, which are useful for automated cleanup of resources you might always use, e.g., database sessions.

Other Build Tools

The plug-ins for other build tools all exploit the same incremental compilation available in SBT, so build times should be roughly the same, independent of the build tool.

The Scala distribution's *lib/scala-compiler.jar* file includes Ant tasks for `scalac`, `fsc`, and `scaladoc`. They are used very much like the corresponding Java Ant tasks. The *build.xml* configuration required is described at the Scala Ant Tasks page, which is old but still valid.

A Scala Maven plug-in is available on the GitHub. It does not require Scala to be installed, because it will download Scala for you.

You can also use the integrated Maven support in Eclipse or IntelliJ.

If you prefer Gradle, details on the Gradle plug-in can be found on Gradle.

Integration with IDEs and Text Editors

If you come from a Java background, you are probably a little bit spoiled by the rich features of today's Java IDEs. Scala IDE plug-ins have come a long way since the first edition of this book and many professional teams now work exclusively with these tools. Scala support is still not as mature as comparable Java support, but all the essential pieces are there. Most of the IDE Scala plug-ins integrate with SBT or Maven for builds, and provide syntax highlighting, some automated refactorings, and a new *worksheet* feature that is a very nice alternative to the command-line REPL.

If you use Eclipse, see the *Scala IDE* project for details on installing and using the Scala plug-in into Eclipse. You can also download a complete Eclipse package with the plug-in already configured.

If you prefer Maven over SBT, see this link for details on using Maven for Scala builds within Eclipse.

Working with the Scala plug-in is very similar to working with the Java tooling in Eclipse. You can create Scala projects and files, run SBT builds and tests, and navigate and refactor code, all within the IDE.

The Eclipse plug-in pioneered a feature not found in the venerable Java plug-in, a *worksheet* that combines the interactivity of the REPL with the convenience of a text editor.

If you have a Scala project open, right-click the top-level project folder to invoke the pop-up menu, then navigate to New → Other. In the "Select a wizard" dialog, open the Scala Wizards folder and select Scala Worksheet. Give it a name and location. The corresponding file will be given the extension *.sc*. The worksheet opens populated with a

default `object` definition containing a single `println` statement. You can rename the object and delete the `println` statement if you want.

Now enter statements as you see fit. Every time you save the file, the contents will be evaluated and the results will be shown on the righthand side of the panel. Make some changes and save again. The worksheet behaves like a "window-oriented" REPL. It's particularly nice for experimenting with code snippets and APIs, including Java APIs!

If you use IntelliJ IDEA, open the Plugins preferences and search for the Scala plug-in to install. It offers comparable features to the Eclipse plug-in, including its own version of the worksheet feature.

Finally, NetBeans has a Scala plug-in with an *Interactive Console* feature that is similar to the worksheet feature in Eclipse and IntelliJ IDEA. See SourceForge for information about the NetBeans plug-in.

Text Editors

While IDEs are popular with Scala developers, you'll also find that many of them prefer using a text editor, like Emacs, Vim, and SublimeText.

Consult the official documentation and community forums for your favorite editor to find the available plug-ins and configuration options for Scala development. Several editor plug-ins can use the ENSIME plug-in, originally designed for Emacs, which provides some "IDE-like" capabilities, such as navigation and some refactorings.

Test-Driven Development in Scala

Test-driven development (TDD) is an established practice in software development with the goal of driving the design of code through tests. A test for a bit of functionality is written *first*, then the code that makes the test pass is written *afterwards*.

However, TDD is more popular among object-oriented programmers. Functional programmers tend to use the REPL to work out types and algorithms, *then* write the code. This has the disadvantage of not creating a permanent, automated "verifier" suite, like TDD produces, but it's true that functional code is less likely to break over time if it is *pure*. As compensation, many functional programmers will write some tests *after* the code is written, to provide a regression-testing suite.

However you write tests, ScalaTest and Specs2 provide DSLs for various testing styles. ScalaTest in particular lets you pick from a variety of styles by mixing in different traits.

In functional languages with rich type systems, like Scala, specifying the types is also seen as a regression-testing capability, one that's exercised every time the compiler is invoked. The goal is to define types that eliminate the possibility of invalid states, when possible.

These types should have well-defined properties. *Property-based testing* or *type-based property-based testing* is another angle on testing popularized by Haskell's QuickCheck and now ported to many languages. Conditions for a type are specified that should be true for all instances of the type. Recall our discussion in Algebraic Data Types. A property-based testing tool tries the conditions using a representative sample of instances that are automatically generated. It verifies that the conditions are satisfied for all the instances (in some cases, combinations of them). In contrast, with a conventional TDD tool, it would be up to the test writer to generate a representative set of example instances and try all possibilities.

ScalaCheck is a Scala port of QuickCheck. Both ScalaTest and Specs2 can drive ScalaCheck property tests. Also, both can be used with JUnit and TestNG, making it easy to mix Java and Scala testing.

Tip

If you work in a Java-only shop and you are interested in trying Scala with minimal risk, consider introducing one or more of these Scala testing tools to test-drive your Java code. It's a low-risk, if limited way to try Scala.

Similarly, all three tools are now supported by the SBT and the plug-ins for Ant, Maven, and Gradle.

The tests provided with the code examples are written in ScalaTest and ScalaCheck, with some JUnit tests to demonstrate Java-Scala interoperability.

Tip

All three tools provide excellent examples of Scala *internal* DSLs. Study them for ideas when you want to write your own DSLs. Study the implementations to learn tricks of the trade.

Third-Party Libraries

Since the first edition of this book, the number of third-party libraries written in Scala has grown enormously. Some widely used libraries today didn't exist at that time and some of the libraries that were popular then have waned. This trend will certainly continue, so consider this section to be a snapshot in time. It is also not intended to be comprehensive. Use it as a starting point, then search the Web to see what options exist for your needs.

A good place to start is http://typelevel.org, which aggregates a variety of powerful libraries for different purposes.

Of course you can use any JVM library written in another language, too. I won't cover those options.

Let's start with "full stack" libraries and frameworks for building web-based applications. By "full stack," I mean everything from backend services to template engines for HTML, JavaScript, and CSS. Others are more limited in their scope, focusing on specific tasks. Table 21-6 summarizes the most popular, currently available options.

Table 21-6. Libraries for web-based applications

Name	URL	Description
Play	http://www.playframework.com/	Typesafe-supported, full-stack framework with Scala and Java APIs. It is also integrated with Akka.
Lift	http://liftweb.net/	The first, and still-popular, full-stack framework.

Table 21-7 describes libraries that are oriented toward backend services.

Table 21-7. Libraries for services

Name	URL	Description
Akka	http://akka.io	Comprehensive, actor-based, distributed computing system. Discussed in Robust, Scalable Concurrency with Actors.
Finagle	http://bit.ly/1vowyQO	An extensible system for building JVM services based on functional abstractions. Developed at Twitter, it is used to construct many of their services.[]
Unfiltered	http://bit.ly/1s0F5s3	A toolkit for servicing HTTP requests that provides a consistent API in front of various backend services.
Dispatch	http://bit.ly/1087FQQ	API for asynchronous HTTP.

Table 21-8 describes various advanced libraries that explore features of the type system, implement functional programming constructs, etc.

Table 21-8. Advanced libraries

These two libraries are listed at typelevel. Other libraries described there also explore advanced language features.

The *scala.io* package provides limited capabilities for I/O and the Java APIs are hard to use. Two third-party projects listed in Table 21-9 seek to fill the gap.

Table 21-10 describes miscellaneous libraries for particular design problems.

Table 21-10. Miscellaneous libraries

Name	URL	Description
scopt	https://github.com/scopt/scopt	A library for command-line parsing.
Typesafe Config	https://github.com/typesafehub/config	A configuration library (Java API).
ScalaARM	http://bit.ly/13oZkdG	Joshua Suereth's library for automatic resource management.
Typesafe Activator	https://github.com/typesafehub/activator	A tool for managing sample Scala projects. Hosted at http://typesafe.com/activator.

See Table 18-1 in Chapter 18 for a list of libraries for *Big Data* and mathematics.

Finally, recall from the Preface that we said the Scala 2.11 release has modularized the library to decompose it into smaller JAR files, so library components that are less frequently used are made optional. Table 21-11 describes them. You can find them at the Maven repository.

Table 21-11. Scala 2.11 optional modules

Name	Artifact Name	Description
XML	scala-xml	XML parsing and construction.
Parser Combinators	scala-parser-combinators	Library of combinators for building parsers.
Swing	scala-swing	A Swing library.
Async	scala-async	An asynchronous programming facility for Scala that offers a direct API for working with Futures.
Partest	scala-partest	Testing framework for the Scala compiler and library.
Partest Interface	scala-partest-interface	Testing framework for the Scala compiler and library.

For a comprehensive list of current, third-party libraries, see the Awesome Scala list on GitHub. Also, http://ls.implicit.ly/ aggregates many Scala libraries.

Recap and What's Next

This chapter filled in details about the Scala tools you'll use daily. Next, we'll look at how Java and Scala code interoperate with each other.