Table of Contents for Programming Scala, 2nd Edition

Let’s start with a brief look at why you should give Scala a serious look. Then we’ll dive in and write some code.

Why Scala?

Scala is a language that addresses the needs of the modern software developer. It is a statically typed, mixed-paradigm, JVM language with a succinct, elegant, and flexible syntax, a sophisticated type system, and idioms that promote scalability from small, interpreted scripts to large, sophisticated applications. That’s a mouthful, so let’s look at each of those ideas in more detail:

A JVM and JavaScript language
: Scala exploits the performance and optimizations of the JVM, as well as the rich ecosystem of tools and libraries built around Java. But it’s not limited to the JVM! *Scala.js* is an experimental port to JavaScript.

Statically typed
: Scala embraces *static typing* as a tool for creating robust applications. It fixes many of the flaws of Java’s type system and it uses type inference to eliminate much of the typing boilerplate.

Mixed paradigm—object-oriented programming
: Scala fully supports *object-oriented programming* (OOP). Scala improves Java’s object model with the addition of *traits*, a clean way of implementing types using *mixin composition*. In Scala, everything *really* is an object, even numeric types.

Mixed paradigm—functional programming
: Scala fully supports *functional programming* (FP). FP has emerged as the best tool for thinking about problems of concurrency, *Big Data*, and general code correctness. Immutable values, first-class functions, functions without side effects, “higher-order” functions, and function collections all contribute to concise, powerful, correct code.

A sophisticated type system
: Scala extends the type system of Java with more flexible generics and other enhancements to improve code correctness. With type inference, Scala code is often as concise as code in dynamically typed languages.

A succinct, elegant, and flexible syntax
: Verbose expressions in Java become concise idioms in Scala. Scala provides several facilities for building *domain-specific languages* (DSLs), APIs that feel “native” to users.

Scalable—architectures
: You can write small, interpreted scripts to large, distributed applications in Scala. Four language mechanisms promote scalable composition of systems: 1) *mixin* composition using *traits*; 2) abstract type members and generics; 3) nested classes; and 4) explicit *self types*.

The name *Scala* is a contraction of the words *scalable language*. It is pronounced *scah-lah*, like the Italian word for “staircase.” Hence, the two “a”s are pronounced the same.

Scala was started by Martin Odersky in 2001. The first public release was January 20th, 2004 (see http://bit.ly/1toEmFE). Martin is a professor in the School of Computer and Communication Sciences at the Ecole Polytechnique Fédérale de Lausanne (EPFL). He spent his graduate years working in the group headed by Niklaus Wirth, of Pascal fame. Martin worked on Pizza, an early functional language on the JVM. He later worked on GJ, a prototype of what later became Generics in Java, along with Philip Wadler, one of the designers of Haskell. Martin was hired by Sun Microsystems to produce the reference implementation of `javac`, the descendant of which is the Java compiler that ships with the Java Developer Kit (JDK) today.

The Seductions of Scala

The rapid growth of Scala users since the first edition of this book confirms my view that Scala is a language for our time. You can leverage the maturity of the JVM, libraries, and production tools, while enjoying state-of-the-art language features with a concise, yet expressive syntax for addressing today's challenges, such as Big Data, scaling through concurrency, and providing highly available and robust services.

In any field of endeavor, the professionals need sophisticated, powerful tools and techniques. It may take a while to master them, but you make the effort because mastery is the key to your success.

I believe Scala is a language for *professional* developers. Not all users are professionals, of course, but Scala is the kind of language a professional in our field needs, rich in features, highly performant, expressive for a wide class of problems. It will take you a while to master Scala, but once you do, you won't feel constrained by your programming language.

What About Java 8?

Java 8 is the most significant update to Java since Java 5 introduced generics. Now we have real anonymous functions, called *lambdas*. You'll see why they are so useful in this book. Interfaces have been extended to allow "default" implementations of the methods they declare, making them more usable as *composable mixins*, like Scala's *traits*. These features are arguably the two most valuable improvements that Scala brought to the JVM compared to Java before version 8. So, is there any point in switching?

Scala adds many improvements that Java may never have, due to backward compatibility limitations, or Java may eventually have them but not until years from now. For example, Scala has richer type inference than Java can provide. Scala has powerful *pattern matching* and *for comprehensions* that dramatically reduce code size and coupling between types. You'll see why they are so valuable as we go.

Also, many organizations are understandably cautious about upgrading their JVM infrastructure. For them deploying the Java 8 JVM may not be an option for a while. At least those organizations can use Scala now with the Java 6 and 7 JVMs.

Still, if you can use Java 8 you might decide it's the best path forward for your team. Reading this book will still teach you many useful techniques that you can apply to Java 8 applications. However, I suspect you'll still find all the additional features of Scala worth the switch.

Okay, let's get started.

Installing Scala

To get up and running as quickly as possible, this section describes how to install some command-line tools that are all you need to work with the examples in the book.[] The examples used in this book were written and compiled using Scala version 2.11.2, the latest release at the time of this writing. Most also work unmodified with the previous release, Scala version 2.10.4, because many teams are still using that version.

Note

Scala 2.11 introduced some new features compared to 2.10, but the release mostly focused on general performance improvements and library refactoring. Scala 2.10 introduced a number of new features compared to 2.9. Because your organization may be using any of these versions, we'll discuss the most important differences as we go. (See an overview of 2.11 here and an an overview of 2.10 here.)

Here are the steps:

Install Java
: Until Scala 2.12 comes along, Java 6, 7, or 8 can be used and it must be installed on your computer (Scala 2.12, which is planned for early 2016, will support Java 8 only). If you need to install Java, go to the Oracle website and follow the instructions to install the full Java Development Kit (JDK).

Install SBT
: Install the de facto build tool for Scala, *SBT* by following the instructions at *scala-sbt.org*. When you are finished, you will have an `sbt` command that you can run from a Linux or OS X terminal or Windows command window. (Other build tools can be used, as we'll see in Other Build Tools.)

Get the book's code examples
: Download the code examples as described in Getting the Code Examples. Expand the archive somewhere convenient on your computer.

Start SBT
: Open a shell or command window and move to the directory where you expanded the code examples. Type the command `sbt test`, which will download all the dependencies you need, including the Scala compiler and third-party libraries. This will take a while and you'll need an Internet connection. Then `sbt` will compile the code and run the unit tests. You'll see lots of output, ending with a "success" message. If you run the command again, it should finish very quickly because it won't need to do anything again.

Congratulations! You're ready to get started. However, you might want to install a few more things that are useful.

Tip

For most of the book, we'll use these tools indirectly through SBT, which downloads the Scala compiler version we want, the standard library, and the required third-party dependencies automatically.

It's handy to download the Scala tools separately, for those times when you aren't working in SBT. We'll run a few of our examples using Scala outside SBT.

Follow the links on the official Scala website to install Scala and optionally, the *Scaladocs*, the analog of *Javadocs* for Scala (in Scala 2.11, the Scala library and Scaladocs have been split into several, smaller libraries). You can also read the Scaladocs online. For your convenience, most mentions of a type in the Scala library will be a link corresponding to a Scaladocs page.

A handy feature of the Scaladocs is a search field above the list of types on the lefthand side. It is very handy for finding a type quickly. Also, the entry for each type has a link to view the corresponding source code in Scala's GitHub repository, which is a good way to learn how the library was implemented. Look for the link on the line labeled "Source." It will be near the bottom of the overview discussion for the type.

Any text editor or IDE (integrated development environment) will suffice for working with the examples. You can find Scala support plug-ins for all the major editors and IDEs. For more details, see Integration with IDEs and Text Editors. In general, the community for your favorite editor is your best source of up-to-the-minute information on Scala support.

Using SBT

We'll learn how SBT works in SBT, the Standard Build Tool for Scala. For now, let's cover the basics we need to get started.

When you start the `sbt` command, if you don't specify a task to run, SBT starts an interactive REPL (*Read, Eval, Print, Loop*). Let's try that now and see a few of the available "tasks."

In the listing that follows, the `$` is the shell command prompt (e.g., `bash`), where you start the `sbt` command, the `>` is the default SBT interactive prompt, and the `#` starts an `sbt` comment. You can type most of these commands in any order:

```
$ sbt
> help       # Describe commands.
> tasks      # Show the most commonly-used, available tasks.
> tasks -V   # Show ALL the available tasks.
> compile    # Incrementally compile the code.
> test       # Incrementally compile the code and run the tests.
> clean      # Delete all build artifacts.
> ~test      # Run incr. compiles and tests whenever files are
saved.
             # This works for any command prefixed by "~".
> console    # Start the Scala REPL.
> run        # Run one of the "main" routines in the project.
> show x     # Show the definition of variable "x".
> eclipse    # Generate Eclipse project files.
> exit       # Quit the REPL (also control-d works).
```

I run `~test` all the time to keep compiling changes and running the corresponding tests. SBT uses an incremental compiler and test runner, so I don't have to wait for a full rebuild every time. When you want to run a different task or exit `sbt`, just hit Return.

The `eclipse` task is handy if you use Eclipse with its Scala plug-in. It generates the appropriate project files so you can import the code as an Eclipse project. If you'll use Eclipse to work with the example code, run the `eclipse` task now.

If you use a recent release of IntelliJ IDEA with its Scala plug-in, you can simply import the SBT project directly.

Scala has its own REPL. You can invoke it using the `console` command. Most of the time in this book when you try the examples yourself in the REPL, you'll do so by first running `console`:

```
$ sbt
> console
[info] Updating {file:/.../prog-scala-2nd-ed/}prog-scala-2nd-ed...
[info] ...
[info] Done updating.
[info] Compiling ...
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.11.2 (Java HotSpot(TM) 64-Bit Server VM, Java ...).
Type in expressions to have them evaluated.
Type :help for more information.

scala> 1 + 2
res0: Int = 3

scala> :quit
```

I've elided some of the output here. Like the SBT REPL, you can also exit with *Ctrl-D*.

When you run `console`, SBT builds your project first and makes the build products available on the `CLASSPATH`. Hence, you can experiment with your code using the REPL.

Tip

Using the Scala REPL is a very effective way to experiment with code idioms and to learn an API, even Java APIs. Invoking it from SBT using the `console` task conveniently adds project dependencies and the compiled project code to the classpath for the REPL.

Running the Scala Command-Line Tools

If you installed the Scala command-line tools separately, the Scala compiler is called `scalac`, analogous to the Java compiler `javac`. We won't use it directly, relying instead on SBT to invoke it for us, but the command syntax is straightforward if you've ever run `javac`.

In your command window, try these commands to see what version you are running and to see help on the command-line arguments. As before, you type the text after the `$` prompt. The rest of the text is the command output:

```
$ scalac -version
Scala compiler version 2.11.2 -- Copyright 2002-2013, LAMP/EPFL
$ scalac -help
Usage: scalac <options> <source files>
where possible standard options include:
  -Dproperty=value       Pass -Dproperty=value directly to the runtime system
.
  -J<flag>               Pass <flag> directly to the runtime system.
  -P:<plugin>:<opt>      Pass an option to a plugin
  ...
```

Similarly, the `scala` command, which is similar to `java`, is used to run programs:

```
$ scala -version
Scala code runner version 2.11.2 -- Copyright 2002-2013, LAMP/EPFL
$ scala -help
Usage: scala <options> [<script|class|object|jar> <arguments>]
   or  scala -help

All options to scalac (see scalac -help) are also allowed.
...
```

We will also occasionally run `scala` to invoke Scala "script" files, something that the `java` command doesn't support. Consider this example script from the code examples:

```
// src/main/scala/progscala2/introscala/upper1.sc

class Upper {
  def upper(strings: String*): Seq[String] = {
    strings.map((s:String) => s.toUpperCase())
  }
}

val up = new Upper
println(up.upper("Hello", "World!"))
```

Let’s run it with the `scala` command. Try this example, where the Linux and Mac OS X paths are shown. I’m assuming your current working directory is the root of the code examples. For Windows, use backslashes instead:

```
$ scala src/main/scala/progscala2/introscala/upper1.sc
ArrayBuffer(HELLO, WORLD!)
```

And thus we have have satisfied the requirement of the Programming Book Guild that our first program must print “Hello World!”

Finally, if you invoke `scala` without a compiled `main` routine to run or a script file, `scala` enters the REPL mode, like running `console` in `sbt`. (However, it won’t have the same classpath you get when running the `console` tasks in `sbt`.) Here is a REPL session illustrating some useful commands (if you didn’t install Scala separately, just start `console` in `sbt` to play with the Scala REPL). The REPL prompt is now `scala>` (some output elided):

```
$ scala
Welcome to Scala version 2.11.2 (Java HotSpot(TM)...).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :help
All commands can be abbreviated, e.g. :he instead of :help.
:cp <path>                add a jar or directory to the classpath
:edit <id>|<line>         edit history
:help [command]           print this summary or command-specific help
:history [num]            show the history (optional num is commands to
show)
...  Other messages

scala> val s = "Hello, World!"
s: String = Hello, World!

scala> println("Hello, World!")
Hello, World!

scala> 1 + 2
res3: Int = 3

scala> s.con<tab>
concat   contains   contentEquals

scala> s.contains("el")
res4: Boolean = true

scala> :quit
$    # back at the shell prompt.
```

We assigned a string, `"Hello, World!"`, to a variable named `s`, which we declared as an immutable value using the `val` keyword. The `println` function prints a string to the console, followed by a line feed.

This `println` is effectively the same thing as Java's System.out.println. Also, Scala uses Java Strings.

Next, note that when we added two numbers, we didn't assign the result to a variable, so the REPL made up a name for us, `res3`, which we could have used in subsequent expressions.

The REPL supports tab completion. The input command shown as `s.con<tab>` is used to indicate that a tab was typed after the `s.con`. The REPL responded with a list of methods on `String` that could be called. The expression was completed with a call to the `contains` method.

Finally, we used `:quit` to exit the REPL. Ctrl-D can also be used.

We'll see additional REPL commands as we go and we'll explore the REPL commands in depth in Command-Line Tools.

Running the Scala REPL in IDEs

Let's quickly discuss one other way to run the REPL that's handy if you use Eclipse, IntelliJ IDEA, or NetBeans. Eclipse and IDEA support a *worksheet* feature that let's you edit Scala code as you would normally edit code for compilation or scripting, but the code is interpreted immediately whenever you save the file. Hence, it's more convenient than using the REPL when you need to modify and rerun nontrivial code fragments. NetBeans has a similar *Interactive Console* feature.

If you use one of these IDEs, see Integration with IDEs and Text Editors for information on the Scala plug-ins and how to use the *worksheet* or *Interactive Console* feature.

A Taste of Scala

In the rest of this chapter and the two chapters that follow, we'll do a rapid tour of many of Scala's features. As we go, we'll discuss just enough of the details to understand what's going on, but many of the deeper background details will have to wait for later chapters. Think of it as a primer on Scala syntax and a taste of what programming in Scala is like day to day.

Tip

When we mention a type in the Scala library, you might find it useful to read more in the Scaladocs about it. The Scaladocs for the current release of Scala can be found here. Note that a search field is shown above the list of types on the lefthand side. It is very handy for finding a type quickly, because the Scaladocs segregate types by package, rather than list them all alphabetically, as in Javadocs.

We will use the Scala REPL most of the time in this book. Recall that you can run it one of three ways, either directly using the `scala` command with no script or "main" argument, using one of the SBT `console` commands, or using the worksheet feature in the popular IDEs.

If you don't use an IDE, I recommend using SBT most of the time, especially when you're working with a particular project. That's what we'll do here, but once you've started `scala` directly or created a worksheet in an IDE, the steps will be the same. Take your pick. Actually, even if you *prefer* IDEs, give SBT in a command window a try, just to see what it's like. I personally rarely use IDEs, but that's just my personal preference.

In a shell window, change to the root directory of the code examples and start `sbt`. At the `>` prompt, type `console`. From now on, we'll omit some of the "boilerplate" in the `sbt` and `scala` output.

Type in the following two lines of code at the `scala>` prompts:

```
                  "Programming
scala> val book = Scala"
book: java.lang.String = Programming Scala

scala> println(book)
Programming Scala
```

The first line uses the `val` keyword to declare an *immutable* variable named `book`. Using immutable values is recommended, because mutable data is a common source of bugs.

Note that the output returned from the interpreter shows you the type and value of `book`. Scala infers from the *literal* value `"Programming Scala"` that `book` is of type `java.lang.String`.

When type information is shown or explicit type information is added to declarations, these *type annotations* follow a colon after the item name. Why doesn't Scala follow Java conventions? Type information is often *inferred* in Scala. Hence, we don't always show type annotations explicitly in code. Compared to Java's `type item` convention, the `item: type` convention is easier for the compiler to analyze unambiguously when you omit the colon and the type annotation and just write `item`.

As a general rule, when Scala deviates from Java syntax, it is usually for a good reason, like supporting a new feature that would be difficult using Java syntax.

Tip

Showing the types in the REPL is very handy for learning the types that Scala infers for particular expressions. It's one example of exploration that the REPL enables.

Larger examples can be tedious to edit and resubmit using only the REPL. Hence, it's convenient to write Scala scripts in a text editor or IDE. You can then execute the script or copy and paste blocks of code.

Let's look again at the *upper1.sc* we ran earlier:

```
// src/main/scala/progscala2/introscala/upper1.sc

class Upper {
  def upper(strings: String*): Seq[String] = {
    strings.map((s:String) => s.toUpperCase())
  }
}

val up = new Upper
println(up.upper("Hello", "World!"))
```

Throughout the book, if a code example is part of the downloadable archive of examples, the first line will be a comment with the path to the file in the archive. Scala follows the same comment conventions as Java, C#, C, etc. A `// comment` goes to the end of a line, while a `/*comment*/` can cross line boundaries.

Also, recall from Getting the Code Examples in the Preface that the script files use the *.sc* extension as a naming convention, while compiled files use the normal *.scala* extension. *This is the book's convention only*. Normally, script files are also named with the *.scala* extension. However, SBT will attempt to compile these scripts when it builds the project and script files cannot be compiled (as we'll discuss in a moment).

First, let's run this script, then discuss the code in detail. Start `sbt` and then run `console` to start Scala. Then use the `:load` command to load (compile and run) the file:

```
scala> :load src/main/scala/progscala2/introscala/upper1.sc
Loading src/main/scala/progscala2/introscala/upper1.sc...
defined class Upper
up: Upper = Upper@4ef506bf    // Used Java's
Object.toString.
ArrayBuffer(HELLO, WORLD!)
```

The last line is the actual `println` output in the script. The other lines are feedback the REPL provides.

So, why can't scripts be compiled? Scripts are designed to be simple and one simplification is that you don't have to wrap declarations (variables and functions) in objects like you would for compiled Java *and* Scala code (a requirement for valid JVM byte code). The `scala` command uses a clever hack to reconcile these conflicting requirements; it wraps your script in an anonymous object that you don't see.

If you really want to compile your script into JVM byte code (a set of **.class* files), you can pass the `-Xscript <object>` arguments to `scalac`, where `<object>` is a name of your choosing. It will be the name of the "main" class that is the entry point for the generated Java application:

```
$ scalac -Xscript Upper1 src/main/scala/progscala2/introscala/upper1.sc
$ scala Upper1
ArrayBuffer(HELLO, WORLD!)
```

Look in your current directory and you will see several *.class* files with funny names. (Hint: some are *anonymous functions* turned into objects!) We'll come back to those names later, but *Upper1.class* has the `main` routine. Let's reverse engineer it with `javap` and the Scala analog, `scalap`:

```
$ javap -cp . Upper1
Compiled from "upper1.sc"
public final class Upper1 {
  public static void main(java.lang.String[]);
}
$ scalap -cp . Upper1
object Upper1 extends scala.AnyRef {
                 /* compiled code
  def this() = { */                  }
  def main(argv : scala.Array[scala.Predef.String]) : scala.Unit =
      /* compiled code
    { */                  }
}
```

OK, let's finally discuss the code itself. Here it is again:

```
// src/main/scala/progscala2/introscala/upper1.sc

class Upper {
  def upper(strings: String*): Seq[String] = {
    strings.map((s:String) => s.toUpperCase())
  }
}

val up = new Upper
println(up.upper("Hello", "World!"))
```

The `upper` method in the `Upper` class (no pun intended) converts the input strings to uppercase and returns them in a `Seq` (for "sequence"). The last two lines create an instance of `Upper` and use it to convert two strings, "Hello" and "World!" to uppercase and finally print the resulting `Seq`.

In Scala, a class begins with the `class` keyword and the entire class body is inside the outermost curly braces (`{…}`). In fact, the body is also the *primary constructor* for the class. If we needed to pass arguments to this constructor, we would put an argument list after the class name, `Upper`.

This bit of code starts the definition of a method:

```
def upper(strings: String*): Seq[String] = ...
```

Method definitions begin with the `def` keyword, followed by the method name and an optional argument list. Next comes an optional return type (it can be inferred in many cases), indicated by a colon and a type. Finally, an equals sign (`=`) separates the method signature from the method body.

The argument list in parentheses is actually a *variable-length argument list* of `String`s, indicated by the `*` after the `String` type for the `strings` argument. That is, you can pass in as many comma-separated strings as you want (including an empty list). Inside the method, the type of the `strings` parameter is actually `WrappedArray`, which wraps Java arrays.

The method return type appears after the argument list. In this case, the return type is `Seq[String]`, where `Seq` ("sequence") is an abstraction for collections that you can iterate through in a fixed order (as opposed to random or undefined order, like `Sets` and `Maps`, where no order is guaranteed). The actual type returned by this method will be `scala.collection.mutable.ArrayBuffer`, but callers don't really need to know that most of the time.

Note that `Seq` is a *parameterized type*, like a *generic* type in Java. It's a "sequence of something," in this case a sequence of strings. Note that Scala uses square brackets (`[…]`) for parameterized types, whereas Java uses angle brackets (`<…>`).

Note

Scala allows angle brackets to be used in *identifiers*, like method and variable names. For example, defining a "less than" method and naming it `<` is common and allowed by Scala, whereas Java doesn't allow characters like that in *identifiers*. So, to avoid ambiguity, Scala uses square brackets instead for parameterized types and disallows them in identifiers.

The body of the `upper` method comes after the equals sign (`=`). Why an equals sign? Why not just use curly braces (`{…}`) to indicate the method body, as in Java?

One reason is to reduce ambiguity. Scala infers semicolons when you omit them. It infers the method's return type in most cases. It lets you omit the argument list in the definition if the method takes no arguments.

The equals sign also emphasizes the principle in *functional programming* that values and functions are more closely aligned concepts. As we'll see, functions are passed as arguments to other functions, returned from functions, and assigned to variables, just like with objects.

Finally, Scala lets you omit the curly braces if the method body has a single expression. So, using an equals sign prevents several possible parsing ambiguities.

The method body calls the `map` method on the `strings` collection, which takes a *function literal* as an argument. Function literals are "anonymous" functions. In other languages, they are variously called *lambdas*, *closures*, *blocks*, or *procs*. Java 8 finally added true anonymous functions, called *lambdas*. Before Java 8, you would implement an interface, usually with an anonymous inner class, that declares a method to do the work. So, even before Java 8 you could sort of do the same thing, "parameterize" some outer behavior by passing in some nested behavior, but the bloated syntax really undermined and obscured the power of this technique.

In this case, we passed in the following function literal:

```
(s:String) => s.toUpperCase()
```

It takes an argument list with a single String argument named s. The body of the function literal is after the "arrow," =>. (UTF8 ⇒ is also allowed.) The body calls toUpperCase() on s. The result of this call is automatically returned by the function literal. In Scala, the last *expression* in a function or method is the return value. The return keyword exists in Scala, but it can only be used in methods, not in anonymous functions like this one. In fact, it is rarely used in methods.

So, the map method we call on the sequence strings passes each String to the function literal and builds up a new collection with the results returned by the function literal. If there were five elements in the original list, for example, the new list will also have five elements.

Continuing with the example, to exercise the code, we create a new Upper instance and assign it to a variable named up. As in Java, C#, and similar languages, the syntax new Upper creates a new instance. No argument list is required because the primary constructor doesn't take any arguments. The up variable is declared as a read-only "value" using the val keyword. It behaves like a final variable in Java.

Finally, we call the upper method on a list of strings, and print out the result with println(…).

We can actually simplify our script even further. Consider this simplified version:

```
// src/main/scala/progscala2/introscala/upper2.sc

object Upper {
  def upper(strings: String*) = strings.map(_.toUpperCase())
}

println(Upper.upper("Hello", "World!"))
```

This code does exactly the same thing, but with a third fewer characters.

On the first line, Upper is now declared as an object, which is a *singleton*. Scala makes the *Singleton Design Pattern* a first-class member of the language. We are declaring a class, but the Scala runtime will only create one instance of Upper. You can't write new Upper, for example. Scala uses objects for situations where other languages would use "class-level" members, like statics in Java. We don't really need more than one *instance* here, because Upper carries no state information, so a singleton is fine.

The Singleton Pattern is often criticized for its drawbacks. For example, it's hard to replace a singleton instance with a *test double* in unit tests, and forcing all computation through a single instance raises concerns about thread safety and performance. However, just as there are times when a static method or value is appropriate in languages like Java, there are times when singletons make sense, as in this example where no state is maintained and the object doesn't interact with the outside world. Hence, we have no reason to ever need a test double nor worry about thread safety when using Upper.

Note

Why doesn't Scala support statics? Compared to languages that allow static members (or equivalent constructs), Scala is more true to the vision that *everything* should be an object. The object construct keeps this policy more consistent than in languages that mix static and *instance* class members. Recall Java's static methods and fields are not tied to an actual instance of some type, whereas Scala objects are single instances of a type.

The implementation of upper on the second line is also simpler. Scala can usually infer the return type of the method, but not the types of the method arguments, so we drop the explicit declaration. Also, because there is only one expression in the method body, we can drop the braces and put the entire method definition on one line. The equals sign before the method body tells the compiler, as well as the human reader, where the method body begins.

Why can't Scala infer the method argument types? Technically, the type inference algorithm does *local type inference*, which means it doesn't work globally over your whole program, but locally within certain scopes. Hence, it can't tell what types the arguments must have, but it is able to infer the type of the method's returned value in most cases, because it sees the whole function body. Recursive functions are one exception where the execution scope extends beyond the scope of the body, so the return type must be declared.

In any event, the types in the argument lists do provide useful documentation for the reader. Just because Scala can infer the return type of a function, should you let it? For simple functions, where the return type is obvious to the reader, perhaps it's not that important to show it explicitly. However, sometimes the inferred type won't be what's expected, perhaps due to a bug or some subtle behavior triggered by certain input argument values or expressions in the function body. Explicit return types express what you *think* should be returned. They also provide useful documentation for the reader. Hence, I recommend erring on the side of adding return types rather than omitting them, especially in public APIs.

We have also exploited a shorthand for the function literal. Previously we wrote it as follows:

```
(s:String) => s.toUpperCase()
```

We have now shortened it to the following expression:

```
_.toUpperCase()
```

The `map` method takes a single function argument, where the function itself takes a single argument. In this case, the function body only uses the argument once, so we can use the "placeholder" indicator _ instead of a named parameter. That is, the _ acts like an anonymous variable, to which the string will be assigned before `toUpperCase` is called. The `String` type is inferred for us, too.

On the last line, using an `object` rather than a `class` simplifies the invocation. Instead of creating an instance with `new Upper`, we can just call the `upper` method on the `Upper` object directly. This is the same syntax you would use when calling static methods in a Java class.

Finally, Scala automatically imports many methods for I/O, like `println`, which is actually a method on the `Console` object in the `scala` package. Packages provide a "namespace" for scoping like they do in Java.

So, we don't need to call `scala.Console.println`. We can just use `println` by itself. This method is one of many methods and types imported automatically that are defined in a library object called `Predef`.

Let's do one more *refactoring*; convert the script into a compiled, command-line tool. That is, let's create a more traditional JVM application with a `main` method:

```
//
src/main/scala/progscala2/introscala/upper1.scala
package progscala2.introscala

object Upper {
  def main(args: Array[String]) = {
    args.map(_.toUpperCase()).foreach(printf("%s ",_
))
    println("")
  }
}
```

Recall that we use the *.scala* extension for code that we compile with `scalac`. Now the `upper` method has been renamed `main`. Because `Upper` is an `object`, this `main` method works exactly like a `main` `static` method in a Java class. It is the entry point to the `Upper` *application*.

Note

In Scala, `main` must be a method in an `object`. (In Java, `main` must be a `static` method in a `class`.) The command-line arguments for the application are passed to `main` in an array of strings, e.g., `args: Array[String]`.

The first code line in the file defines the `package` for the type, named `intro`. Inside the `Upper.main` method, the expression uses the same shorthand notation for `map` that we just examined:

```
args.map(_.toUpperCase())...
```

The call to `map` returns a new collection. We iterate through it with `foreach`. We use a `_` placeholder shortcut again in another *function literal* that we pass to `foreach`. In this case, each string in the collection is passed as an argument to `scala.Console.printf`, another function imported from `Predef`, which takes a format string followed by arguments to stuff into the string:

```
args.map(_.toUpperCase()).foreach(printf("%s ",_))
```

To be clear, these two uses of `_` are completely independent of each other, because they are in different scopes.

Function chaining and function-literal shorthands like these can take some getting used to, but once you are comfortable with them, they yield very readable, yet concise and powerful code that minimizes use of temporary variables and other boilerplate. If you are a Java programmer, imagine writing the same logic in pre-Java 8 code using anonymous inner classes.

The last line in `main` adds a final line feed to the output.

This time, you must first compile the code to a JVM *.class* file using `scalac` (the `$` is the command prompt):

```
$ scalac
src/main/scala/progscala2/introscala/upper1.scala
```

You should now have a new directory named *progscala2/introscala* that contains several *.class* files, including a file named *Upper.class*. Scala must generate valid JVM byte code. One requirement is that the directory structure must match the package structure. Java enforces this at the source code level, too, but Scala is more flexible. Note that the source file in our downloaded code examples is actually in a directory called *IntroScala*, but we use a different name for the package. Java also requires a separate file for each top-level class, but in Scala you can have as many types in each file as you want. While you don't have to follow Java conventions for organizing your source code into directories that match the package structure and one file per top-level class, many teams follow this convention anyway because it's familiar and helps keep track of where code is defined.

Now, you can execute this command for any list of strings. Here is an example:

```
$ scala -cp . progscala2.introscala.Upper Hello
World!
HELLO WORLD!
```

The `-cp .` option adds the current directory to the search classpath, although this isn't actually required in this case.

Try running the program with other arguments. Also, see what other class files are in the *progscala2/introscala* directory and use `javap` or `scalap`, as before, to see what definitions they contain.

Finally, we didn't really need to compile the file, because the SBT build does that for us. We could run it at the SBT prompt using this command:

```
> run-main progscala2.introscala.Upper Hello
World!
```

Using the `scala` command, we need to point to the correct directory where SBT writes class files:

```
$ scala -cp target/scala-2.11/classes progscala2.introscala.Upper Hello
World!
HELLO WORLD!
```

Let's do one last refactoring of this code:

```
//
src/main/scala/progscala2/introscala/upper2.scala
package progscala2.introscala

object Upper2 {
  def main(args: Array[String]) = {
                                                       "
    val output = args.map(_.toUpperCase()).mkString("
)
    println(output)
  }
}
```

After mapping over the input arguments, instead of iterating through them with `foreach` to print each word, we call a convenience method on the iterable collections to make a string from them. The `mkString` method that takes a single argument lets us specify the delimiter between the collection elements. There is another `mkString` method that takes three arguments, a leftmost prefix string, the delimiter, and a rightmost suffix string. Try changing the code to use `mkString("[", ", ", "]")`. What's the output look like?

We saved the output of `mkString` to a variable and then used `println` to print it. We could have simply wrapped `println` around the whole `map` followed by `mkString`, but using a variable here makes the code easier to read.

A Taste of Concurrency

There are many reasons to be seduced by Scala. One reason is the Akka API for building robust concurrent applications using an intuitive model called *actors* (see http://akka.io).

This next example is a bit ambitious to tackle so soon, but it gives us a taste for how the power and elegance of Scala, combined with an intuitive concurrency API, can yield elegant and concise implementations of concurrent software. One of the reasons you might be investigating Scala is because you're looking for better ways to scale your applications by exploiting concurrency across the cores in your CPUs and the servers in your clusters. Here's one way to do it.

In the *Actor Model of Concurrency*, independent software entities called *actors* share no mutable state information with each other. Instead, they communicate by exchanging messages. By eliminating the need to synchronize access to shared, mutable state, it is far easier to write robust, concurrent applications. Each actor might mutate state as needed, but if it has exclusive access to that state and the actor framework guarantees that invocations of actor code are thread-safe, then the programmer doesn't have to use tedious and error-prone synchronization primitives.

In this simple example, instances in a geometric `Shape` hierarchy are sent to an actor for drawing on a display. Imagine a scenario where a rendering farm generates scenes for an animation. As the rendering of a scene is completed, the geometric shapes that are part of the scene are sent to an actor for a display subsystem.

To begin, we define a `Shape` class hierarchy:

```
// src/main/scala/progscala2/introscala/shapes/Shapes.scala
package progscala2.introscala.shapes

case class Point(x: Double = 0.0, y: Double = 0.0)                     //
❶

abstract class Shape() {
// ❷
  /**
   * Draw takes a function argument. Each shape will pass a
stringized
   * version of itself to this function, which does the
"drawing".

*/
                                         "draw:
  def draw(f: String => Unit): Unit = f(s${this.toString}"       )     //
❸
}

case class Circle(center: Point, radius: Double) extends Shape         //
❹

case class Rectangle(lowerLeft: Point, height: Double, width: Double) // ❺
      extends Shape

case class Triangle(point1: Point, point2: Point, point3: Point)       //
❻

      extends Shape
```

❶

Declare a class for two-dimensional points.

❷

Declare an abstract class for geometric shapes.

❸

Implement a `draw` method for “rendering” the shapes. We just write a formatted string.

❹

A circle with a center and radius.

❺

A rectangle with a lower-left point, height, and width. We assume for simplicity that the sides are are parallel to the horizontal and vertical axes.

❻

A triangle defined by three points.

The argument list after the `Point` class name is the list of constructor parameters. In Scala, the *whole* class body is the constructor, so you list the arguments for the *primary* constructor after the class name and before the class body.

In this case, there is no class body. Because we put the `case` keyword before the class declaration, each constructor parameter is automatically converted to a read-only (immutable) field of `Point` instances. That is, if you instantiate a `Point` instance named `point`, you can read the fields using `point.x` and `point.y`, but you *can't change their values*. Attempting to use `point.y = 3.0` triggers a compilation error.

You can also provide default values for the arguments. The `= 0.0` after each argument definition specifies `0.0` as the default. Hence, the user doesn't have to provide them explicitly, but they are inferred left to right. Let's use our SBT project to explore:

```
$ sbt
...
> compile
Compiling ...
[success] Total time: 15 s, completed ...
> console
[info] Starting scala interpreter...

scala> import progscala2.intro.shapes._
import progscala2.intro.shapes._

scala> val p00 = new Point
p00: intro.shapes.Point = Point(0.0,0.0)

scala> val p20 = new Point(2.0)
p20: intro.shapes.Point = Point(2.0,0.0)

scala> val p20b = new Point(2.0)
p20b: intro.shapes.Point = Point(2.0,0.0)

scala> val p02 = new Point(y = 2.0)
p02: intro.shapes.Point = Point(0.0,2.0)

scala> p00 == p20
res0: Boolean = false

scala> p20 == p20b
res1: Boolean = true
```

So, when we specified no arguments, Scala used `0.0` for both. When we specified one argument, Scala applied it to the leftmost argument, `x`, and used the default value for the remaining argument. We can even reference the arguments by name. For `p02`, we wanted to use the default value for `x`, but specify the value for `y`, so we used `Point(y = 2.0)`.

While there is no class body for `Point`, another feature of the `case` keyword is the compiler automatically generates several methods for us, including the familiar `toString`, `equals`, and `hashCode` methods in Java. The output shown for each point, e.g., `Point(2.0,0.0)`, is the `toString` output. The `equals` and `hashCode` methods are difficult for most developers to implement correctly, so autogeneration is a real benefit.

When we asked if `p00 == p20` and `p20 == p20b`, Scala invoked the generated `equals` method. This is in contrast with Java, where `==` just compares *references*. You have to call `equals` explicitly to do a *logical* comparison.

A final feature of case classes we'll mention now is that the compiler also generates a *companion object*, a singleton of the same name, for each case class (in this case, `object Point`).

Note

You can define companions yourself. Any time an `object` and a `class` have the same name *and* they are defined in the same file, they are *companions*.

You can add methods to the companion object; we'll see how later on. However, it will have several methods added automatically, one of which is named `apply`. It takes the same argument list as the constructor.

Any time you write an object followed by an argument list, Scala looks for an `apply` method to call. That is to say, the following two lines are equivalent:

```
val p1 = Point.apply(1.0, 2.0)
val p2 = Point(1.0, 2.0)
```

It's a compilation error if no `apply` method exists for the object. The arguments must conform to the expected argument list, as well.

The `Point.apply` method is effectively a *factory* for constructing `Points`. The behavior is simple here; it's just like calling the constructor without the `new` keyword. The companion object generated is equivalent to this:

```
object Point {
  def apply(x: Double = 0.0, y: Double = 0.0) = new Point(x, y)
  ...
}
```

However, a more sophisticated use of the companion object `apply` methods is possible for class hierarchies. A parent class object could choose to instantiate a particular subtype that works best for the argument list. For example, a data structure must have an implementation that is optimal for a small number of elements and a different implementation that is optimal for a large number of elements. A factory can hide this logic, giving the user a single uniform interface.

Note

When an argument list is put after an object, Scala looks for an `apply` method to call that matches the argument list. Put another way, `apply` is *inferred*. Syntacticly, any object with an `apply` method behave like a function.

Putting an `apply` method on a companion object is the conventional idiom for defining a factory method for the class. An `apply` method on a class that isn't an `object` has whatever meaning is appropriate for instances of that class. For example, `Seq.apply(index: Int)` retrieves the element at position `index` for sequences (counting from zero).

`Shape` is an abstract class. As in Java, we can't instantiate an abstract class, even if none of the members is abstract. `Shape.draw` is defined, but we only want to instantiate concrete shapes: `Circle`, `Rectangle`, and `Triangle`.

Note the argument passed to `draw`. It is a *function* of type `String => Unit`. That is, `f` takes a `String` argument and returns `Unit`, which is a real type, but it behaves roughly like `void` in Java. The name is a common convention in functional programming.

The idea is that callers of `draw` will pass a function that does the actual drawing when given a string representation of the shape.

Tip

When a function returns `Unit` it is *totally side-effecting*. There's nothing useful that can be done with `Unit`, so the function can only perform side effects on some state, either globally, like performing input or output (I/O), or locally in some object.

Normally in functional programming, we prefer *pure* functions that have no side effects and return all their work as their return value. These functions are far easier to reason about, test, and reuse. Side effects are a common source of bugs. However, real-world programs require I/O, at least.

`Shape.draw` demonstrates the idea that functions are *first-class values*, just like `Strings`, `Ints`, `Points`, and other objects. Like other values, we can assign functions to variables, pass them to other functions as arguments, as in `draw`, and return them from functions. We'll use this feature as a powerful tool for building composable, yet flexible software.

When a function accepts other functions as arguments or returns functions as values, it is called a *higher-order function* (HOF).

You could say that `draw` defines a *protocol* that all shapes have to support, but users can customize. It's up to each shape to serialize its state to a string representation through its `toString` method. The `f` method is called by `draw` and it constructs the final string using an *interpolated string*, a feature introduced in Scala 2.10.

Warning

If you forget the `s` before the interpolated string, you'll get the literal output `draw: ${this.toString}`, i.e., with no interpolation.

`Circle`, `Rectangle`, and `Triangle` are concrete subclasses of `Shape`. They have no class bodies, because the `case` keyword defines all the methods we need, such as the `toString` methods required by `Shape.draw`.

For simplicity, we assume that `Rectangles` are not rotated relative to the *x* and *y* axes. Hence, all we need is one point, the lower lefthand point will do, and the height and width of the rectangle. `Triangle` takes three `Points` as its constructor arguments.

In our simple program, the `f` we will pass to `draw` will just write the string to the console, but you could build a real graphics application that uses an `f` to render the shape to a display.

Now that we have defined our shapes types, let's return to actors. We'll use the Akka library distributed by Typesafe. We have already defined it as a dependency in our project *build.sbt* file.

Here is the code for our `ShapesDrawingActor`:

```
// src/main/scala/progscala2/introscala/shapes/ShapesDrawingActor.scala
package progscala2.introscala.shapes

object Messages {
// ❶
  object Exit
// ❷
  object Finished
  case class Response(message: String)
// ❸
}

import akka.actor.Actor
// ❹

class ShapesDrawingActor extends Actor {
// ❺
  import Messages._
// ❻

  def receive = {
// ❼
    case s: Shape =>
      s.draw(str => println(s"ShapesDrawingActor: $str"))
                         "ShapesDrawingActor: $s
      sender ! Response(sdrawn"                        )
    case Exit =>
               "ShapesDrawingActor:
      println(sexiting..."                     )
      sender ! Finished
                        // default. Equivalent to "unexpected:
    case unexpected =>  Any"
                               "ERROR: Unknown message:
      val response = Response(s$unexpected"                         )
      println(s"ShapesDrawingActor: $response")
      sender ! response
  }
}
```

❶

Declare an object `Messages` that defines most of the messages we'll send between actors. They work like "signals" to trigger behavior. Putting them in an object is a common encapsulation idiom.

❷

`Exit` and `Finished` have no state of their own, so they function like "flags."

❸

The case class `Response` is used to send an arbitrary string message to a sender in response to a message received from the sender.

❹

Import the `akka.actor.Actor` type, an abstract base class, which we'll subclass to define our actors.

❺

Define an actor for drawing shapes.

❻

Import the three messages defined in `Messages` here. Nesting imports, which is permitted in Scala, scopes these values to where they are used.

❼

Implement the one abstract method, `Actor.receive`, that we have to implement, which defines how to handle incoming messages.

In most actor systems, including Akka, there is a *mailbox* associated with each actor where messages are stored until they can be processed by the actor. Akka guarantees that messages are processed in the order in which they are received, and it guarantees that while each message is being processed, the code won't be preempted by another thread, so the handler code is inherently thread-safe.

Note that the `receive` method has a curious form. It takes no arguments and the body is just a sequence of expressions starting with the `case` keyword:

```
def receive = {
  case first_pattern =>
    first_pattern_expressions
  case second_pattern =>

second_pattern_expressions
}
```

This body is the *literal* syntax for a special kind of function called a `PartialFunction`. The actual type is `PartialFunction[Any,Unit]`, which means it takes an argument of type `Any` and returns `Unit`. `Any` is the root class of the type hierarchy in Scala, so the function accepts any argument. Because it returns `Unit`, the body must be *purely side-effecting*. This is necessary for actor systems, because the messaging is asynchronous. There is nothing to "return" to, in the usual sense. So, our code blocks will usually send other messages, including replies to the sender.

A `PartialFunction` consists only of `case` clauses, which do *pattern matching* on the message that will be passed to the function. There is no function argument shown for the message. It's handled internally by the implementation.

When one of the patterns matches, the expressions after the arrow (`=>`) up to the next `case` keyword (or the end of the function) are evaluated. The expressions don't need to be wrapped in curly braces, because the arrow and the next `case` keyword (or the end of the function) provide unambiguous boundaries. Also, if there is just one short expression, it can go on the same line after the arrow.

A *partial function* sounds complicated, but it's actually a simple idea. Recall that a one-argument function takes a value of some type and returns a value of another or the same type. A partial function explicitly says, "I might not be able to do anything with every value you give me." A classic example from mathematics is division, `x/y`, which is undefined when the denominator `y` is 0. Hence, division is a *partial function*.

So, each message is tried against the three pattern match expressions and the first one that matches wins. Let's break down the details of `receive`:

```
def receive = {
  case s: Shape =>
//  ❶
    ...
  case Exit =>
//  ❷
    ...
  case unexpected =>
//  ❸
    ...
}
```

❶

If the message is a `Shape` instance, the first `case` clause matches. The variable `s` is assigned to refer to the shape; i.e., `s` will be of type `Shape`, while the input message has the type `Any`.

❷

The message equals `Exit`. This will be a signal that we're done.

❸

This is the "default" clause that matches anything. It is equivalent to `unexpected: Any`, so it matches anything passed in that didn't match the preceding pattern matches. The message is assigned to the variable `unexpected`.

Because the last match works for all messages, it must be the last one. If you tried moving it before the others you would get an error message about "unreachable code" for the subsequent `case` clause.

Note that because we have a default clause, it means that our "partial" function is actually "total"; it successfully handles all inputs.

Now let's look at the expressions invoked for each match:

```
def receive = {
  case s: Shape =>
    s.draw(str => println(s"ShapesDrawingActor: $str"))                //
❶
                   "ShapesDrawingActor: $s
    sender ! Response(sdrawn"                        )                 //
❷
  case Exit =>
         "ShapesDrawingActor:
    println(sexiting..."                     )                         //
❸
    sender ! Finished
//  ❹
  case unexpected =>
                         "ERROR: Unknown message:
    val response = Response(s$unexpected"                        )   //
❺
    println(s"ShapesDrawingActor: $response")
    sender ! response
//  ❻
}
```

❶

Call `draw` on the shape `s`, passing it an anonymous function that knows what to do with the string generated by `draw`. In this case, it just prints the string.

❷

Send a message to the "sender" with a response.

❸

Print a message that we're quitting.

❹

Send the `Finished` message to the "sender."

❺

Create an error message as a `Response`, then print it.

❻

Send a message to the "sender" with the response.

Lines like `sender ! Response(s"ShapesDrawingActor: $s drawn")` construct a reply and send it to the sender of the shape. `Actor.sender` is a function that returns a reference to whichever actor sent the message and the `!` is a method for sending *asynchronous* messages. Yes, `!` is a method name. The choice of `!` is a convention adopted from *Erlang*, a language that popularized the actor model.

We are also using a bit of syntactic sugar that Scala allows. The following two lines of code are equivalent:

```
                   "ShapesDrawingActor: $s
sender ! Response(sdrawn"                        )
                   "ShapesDrawingActor: $s
sender.!(Response(sdrawn"                        ))
```

When a method takes a single argument, you can drop the period after the object and drop the parentheses around the argument. Note that the first line has a cleaner appearance, which is why this syntax is supported. This notation is called *infix notation*, because the “operator” `!` is between the object and argument.

Tip

Scala method names can use operator symbols. When methods are called that take a single argument, the period after the object and the parentheses around the argument can be dropped. However, there will sometimes be expressions with parsing ambiguities that require you to keep the period or the parentheses or both.

One last note before we move on to the last actor. One of the commonly taught tenets of object-oriented programming is that you should never use case statements that match on instance type, because inheritance hierarchies evolve, which breaks these case statements. Instead, polymorphic functions should be used. So, is the pattern-matching code just discussed an *antipattern*?

Recall that we defined `Shape.draw` to call the `toString` method on the `Shape`, which is implemented in each concrete subclass because they are case classes. Hence, the code in the first `case` statement invokes a polymorphic `toString` operation and we don’t match on specific subtypes of `Shape`. This means our code won’t break if we change the `Shape` class hierarchy. The other case clauses match on unrelated conditions that also won’t change frequently, if at all.

Hence, we have combined polymorphic dispatch from object-oriented programming with pattern matching, a workhorse of functional programming. This is one way that Scala elegantly integrates these two programming paradigms.

Finally, here is the `ShapesDrawingDriver` that runs the example:

```
//
src/main/scala/progscala2/introscala/shapes/ShapesActorDriver.scala
package progscala2.introscala.shapes
import akka.actor.{Props, Actor, ActorRef, ActorSystem}
import com.typesafe.config.ConfigFactory

// Message used only in this
file:
private object Start                                                   //
❶

object ShapesDrawingDriver {                                           //
❷
  def main(args: Array[String]) {                                      // ❸
    val system = ActorSystem("DrawingActorSystem", ConfigFactory.load())
    val drawer = system.actorOf(
      Props(new ShapesDrawingActor), "drawingActor")
    val driver = system.actorOf(
       Props(new ShapesDrawingDriver(drawer)), "drawingService")
    driver ! Start                                                     //
❹
  }
}

class ShapesDrawingDriver(drawerActor: ActorRef) extends Actor {      // ❺
  import Messages._

  def receive = {
    case Start =>                                                      //
❻
      drawerActor ! Circle(Point(0.0,0.0), 1.0)
      drawerActor ! Rectangle(Point(0.0,0.0), 2, 5)
      drawerActor ! 3.14159
      drawerActor ! Triangle(Point(0.0,0.0), Point(2.0,0.0), Point(1.0,2.0))
      drawerActor ! Exit
    case Finished =>                                                   //
❼
              "ShapesDrawingDriver: cleaning
      println(sup..."                              )
      context.system.shutdown()
    case response: Response =>                                         //
❽
             "ShapesDrawingDriver: Response =
      println("                                 + response)
    case unexpected =>                                                 //
❾
      println(
"ShapesDrawingDriver: ERROR: Received an unexpected message =
"
        + unexpected)
  }
}
```

❶

A message used only in this file (`private`) is used to start everything. Using a special start message is a common idiom.

❷

The “driver” actor.

❸

The `main` method that is run to drive the application. It constructs an `akka.actor.ActorSystem` and then builds the two actors, the `ShapesDrawingActor` we discussed earlier and the `ShapesDrawingDriver` we’ll discuss shortly. We’ll defer discussing the details of Akka setup logic until Robust, Scalable Concurrency with Actors. For now, note that we need to pass `ShapesDrawingActor` to `ShapesDrawingDriver`. Actually, we pass an `akka.actor.ActorRef` (“actor reference”) that points to the actual instance.

❹

Send `Start` to the driver to begin!

❺

The `ShapesDrawingDriver` actor.

❻

When its `receive` handler gets the `Start` message, it fires off five asynchronous messages to `ShapesDrawingActor`: three shapes, the value of Pi (which will be considered an error), and `Exit`. So, this will be a short-lived actor system!

❼

When `Finished` is received as a reply to `Exit` (recall what `ShapesDrawingDriver` does with `Exit`), we shut down the actor system, accessed through a `context` field in `Actor`.

❽

Simply print any other expected responses.

❾

A similar default clause for unexpected messages as we used previously.

Let’s try it! At the `sbt` prompt, type `run`, which will compile the code if necessary and then present you with a list of all the code examples that have a `main` method:

```
> run
[info] Compiling ...

Multiple main classes detected, select one to run:

 [1] progscala2.introscala.shapes.ShapesDrawingDriver
 ...

Enter number:
```

Enter `1` and you should see output similar to the following (output wrapped to fit):

```
...
Enter number: 1

[info] Running progscala2.introscala.shapes.ShapesDrawingDriver
ShapesDrawingActor: draw: Circle(Point(0.0,0.0),1.0)
ShapesDrawingActor: draw: Rectangle(Point(0.0,0.0),2.0,5.0)
ShapesDrawingActor: Response(ERROR: Unknown message: 3.14159)
ShapesDrawingActor: draw: Triangle(
  Point(0.0,0.0),Point(2.0,0.0),Point(1.0,2.0))
ShapesDrawingActor: exiting...
ShapesDrawingDriver: Response = Response(
  ShapesDrawingActor: Circle(Point(0.0,0.0),1.0) drawn)
ShapesDrawingDriver: Response = Response(
  ShapesDrawingActor: Rectangle(Point(0.0,0.0),2.0,5.0) drawn)
ShapesDrawingDriver: Response = Response(ERROR: Unknown message: 3.14159)
ShapesDrawingDriver: Response = Response(
  ShapesDrawingActor: Triangle(
    Point(0.0,0.0),Point(2.0,0.0),Point(1.0,2.0)) drawn)
ShapesDrawingDriver: cleaning up...
[success] Total time: 10 s, completed Aug 2, 2014 7:45:07 PM
>
```

Because all the messages are sent asynchronously, note how responses printed by the driver are interleaved with output from the drawing actor, but messages were handled in the order sent. The output will vary from run to run.

So now you have a taste of actor-based concurrency and a few more powerful Scala features.

Recap and What's Next

We introduced Scala, then dove into some nontrivial Scala code, including a taste of Akka's actor library for concurrency.

As you explore Scala, you will find other useful resources that are available on http://scala-lang.org. You will find links for libraries, tutorials, and various papers that describe features of the language.

Typesafe, Inc. is the commercial company that supports Scala and several JVM-based development tools and frameworks, including Akka, Play, and others. You'll find many useful resources on the company's website. In particular, the Typesafe Activator is a tool for exploring, downloading, and building from templates for different kinds of applications using Scala and Java tools. Finally, Typesafe offers support subscriptions, consulting, and training.

Next we'll continue our introduction to Scala features, emphasizing the various concise and efficient ways of getting lots of work done.