Chapter 7
A Little Scala

7.1 Introduction

In the last chapter, you learned a little about the history of Scala and the Scala development environment. In this chapter, you encounter a little of the Scala language, what happens when you compile and run a Scala program, the Scala runtime (Virtual Machine) and the Scala IDE.

7.2 The Scala Environment

There are a number of things that you need in order to develop using the Scala language. First of all you need access to the Scala compiler. The compiler is called `scalac`, and you may use it from the command line to compile Scala code files directly or you may use it via an IDE (such as the Scala IDE) that can compile your code for you automatically. It can also be used via various application build tools such as the Simple Build Tool (SBT) for Scala.

When you compile Scala source files (ones with a `.scala` extension) you create byte code files. These files have a `.class` extension. They are not executable files directly (these are files which have a .exe. or .bin type extension and can be run natively by the operating system). Instead a byte code file is a compressed version of your Scala code that is run via a Virtual Machine. This Virtual Machine acts like a computer that executes byte code files but is actually a software application that can be ported to numerous different operating systems.

Scala can run on the Scala Virtual Machine (which is the Java Virtual Machine, or JVM allied with the Scala runtime libraries that provide for the various concepts, functionality and frameworks that Scala uses). This combination of the JVM and the Scala libraries can be referred to as the Scala runtime and can be used in either interpreter mode (in which case you can type in Scala code and immediately see the

© Springer International Publishing AG 2018

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_7

result) or batch mode in which case you can run applications as you would any other type of environment.

To summarise then, to run the Scala environment you need:

- The Scala compiler
- The Scala Virtual Machine
- The Scala runtime libraries.

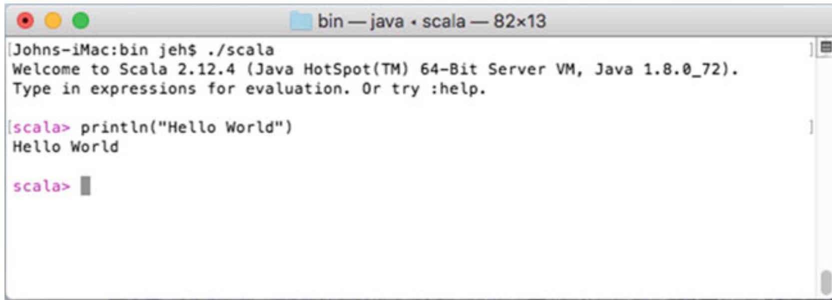
If you use the Scala IDE described later in this chapter then you will automatically obtain these. However, you can install Scala without the need for an IDE if you wish. This can be obtained from the main Scala website.

7.3 The Scala Shell

Scala provides an interactive interpreter or *shell* that you can use to try out pieces of Scala code. This interactive shell allows you to type in Scala code and for that Scala code to be immediately evaluated and the results presented to you. This shell is often referred to as REPL for 'R'ead, 'E'valuate, 'P'rint 'L'oop.

To start up the interactive shell all you need to do is to open a command window (on Windows) or a terminal (on Mac or Unix) and to type in *scala* (assuming that the Scala `bin` directory is on your path or you are in the bin directory). When you type in *scala* you will enter the interactive interpreter and can enter Scala expressions and immediately see the result. For example, the expression `2 + 3` is shown below.

In this example, we asked the interpreter to add 2 and 3 together. The Scala interpreter evaluated the expression and printed out the result. The result (res0) indicates that the result is an Int (integer) of the value 5. This is because we entered an expression that returned a result. We could also have entered an operation that returned nothing, such as *println*; this is shown in the next figure.



Here you can see that the result is that the string is printed out.

To leave the Scala interpreter shell use CTRL-D or use the command *exit*.

7.4 The Scala IDE

You will need a Scala environment on your local machine in order to develop, compile, test and run Scala applications. As Scala is a JVM language this also means that you must have a Java environment on your machine. In theory, you could install each of these components yourself and use whatever editor you wished (including Emacs, or TextEdit). However, the easiest way to get started with Scala is to install one of the Scala IDEs available free on the Web.

There are several to choose from with the Intellij IDEA and Eclipse-based Scala IDEs being the most popular.

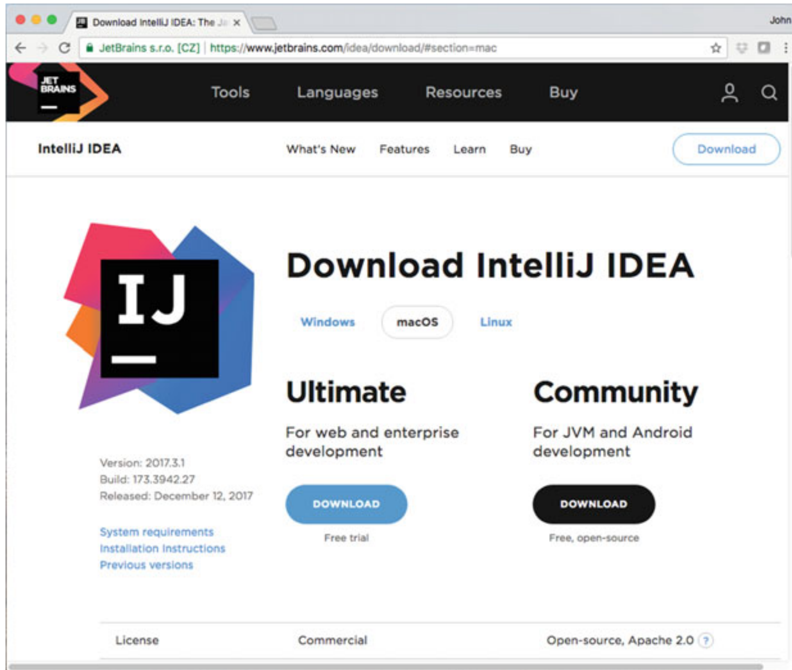
We will be using the IntelliJ-based Scala IDE in the examples throughout this book.

The IntelliJ IDE provides full support for Scala (as well as Java). Support for Scala is built into the IntelliJ IDEA Ultimate version; however, an additional plugin must be installed to use Scala with the IntelliJ IDEA Community Edition (the free version).

We will step through installing the IntelliJ IDEA Community Edition and then add the Scala Plugin to it.

You can download the IntelliJ IDE from:

- https://www.jetbrains.com/idea/download



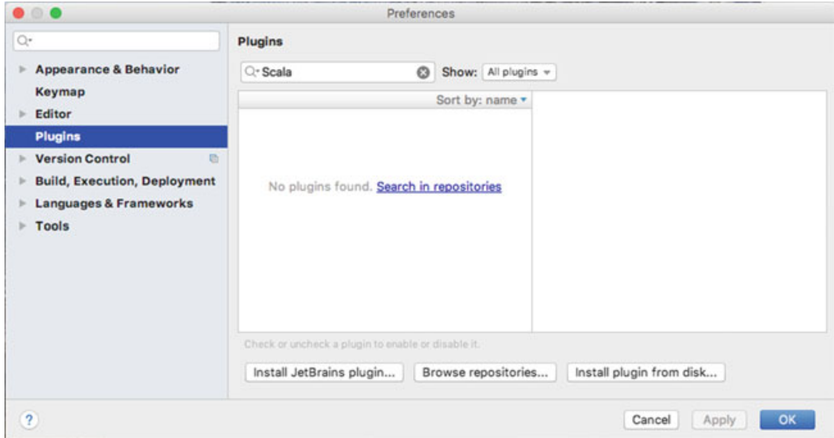
There are versions available for Windows, Linux and Mac systems.

Once you have installed IntelliJ you can install the Scala plugin; see the following link for more information.

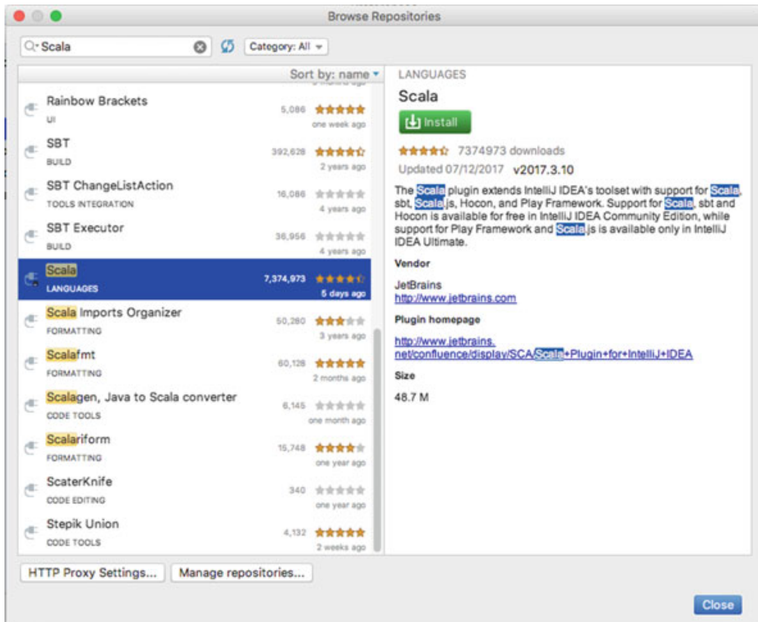
https://www.jetbrains.com/help/idea/discover-intellij-idea-for-scala.html

To install the Scala plugin start up your IntelliJ IDE. If this is the first time you are running IntelliJ then it will prompt you to download featured plugins. At this point you can select the Scala plugin.

If you have already run IntelliJ then you can use the Plugins page for the installation. To find this page go to the menu bar and select IntelliJ IDEA->Preferences… This will display the 'Preferences' dialog. On the left-hand side, select the option in which the 'Plugins' are listed. Select this and the available plugins are listed on the right-hand side and select 'Search in repositories' as shown below:



Enter Scala into the search box and find the Scala Plugin in the list displayed, as shown below:



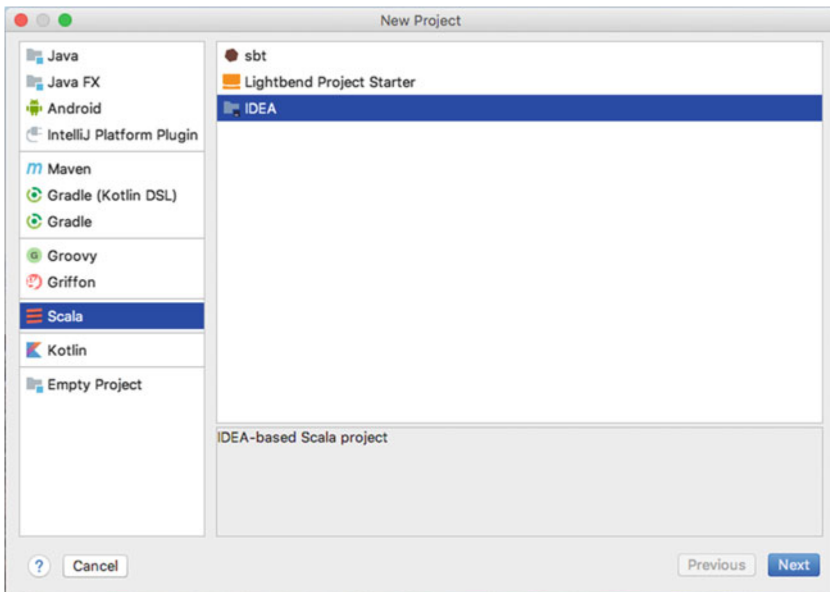
Now click on the ‘Install’ button on the right-hand side. Once the plugin is downloaded and installed you will need to restart IntelliJ (this is indicated by a button stating ‘Restart IntelliJ IDEA’).

You are now ready to start working with Scala and IntelliJ.

7.4.1 *Creating an IntelliJ Project*

IntelliJ is oriented around the idea of a project containing one or more modules. The project is associated with a directory location, and *typically* modules reside under that project location.

Personally I create a directory called *projects*, in which I place a particular project for a particular task or tool and inside this I have my modules. The project and the Scala environment to use with the project are specified when you create a new Project from the File->New->Projects … menu. This causes a dialog wizard to be displayed that will take you through the steps required to create a new Scala project as shown in below figure.

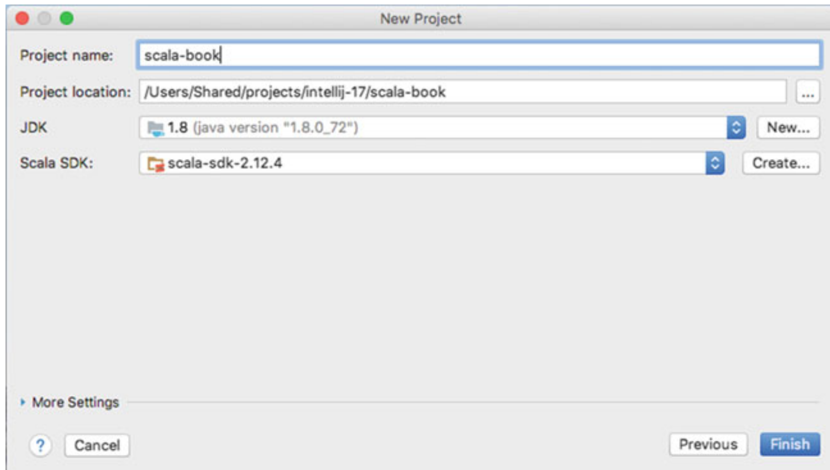


If this is the first tie that you have created a Scala project you may also need to set up the Scala SDK.

On the next screen give your project a name, and select the location of the project (an appropriate directory).

If on this screen, the Scala SDK field is blank (or does not contain the version of Scala you wish to use) click on the 'Create…'. On the resulting dialog, Browse to the location in which you have installed the Scala SDK and select it.

My set-up is shown below:



Now click on 'Finish'.

7.4.2 *Inside IntelliJ*

You will now be presented with an empty editor (see Fig. 7.1). This editor is made up of Windows providing access to the modules and source files you are working with. The current display shows the Project Window on the left. The currently blank central area is where your code will be displayed.

7.4.3 *Creating a Module*

Although it is not strictly necessary to have multiple modules within an IntelliJ project, it does help to organise your work. We will therefore create an initial module to hold our first examples.

To do this select File->New->Module… Make sure that the Scala module type is selected in the left-hand list of the resulting dialog, and click Next.

You will now be presented with the New Module dialog as shown below. Enter a name for your module, for example 'HelloWorld'. Assuming that you want the module to be located under the project directory you can accept the defaults for the remaining fields.

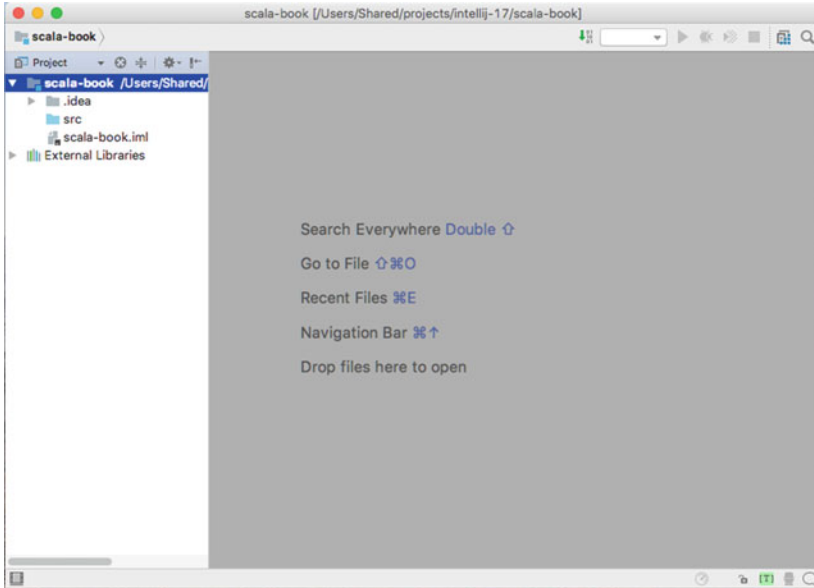
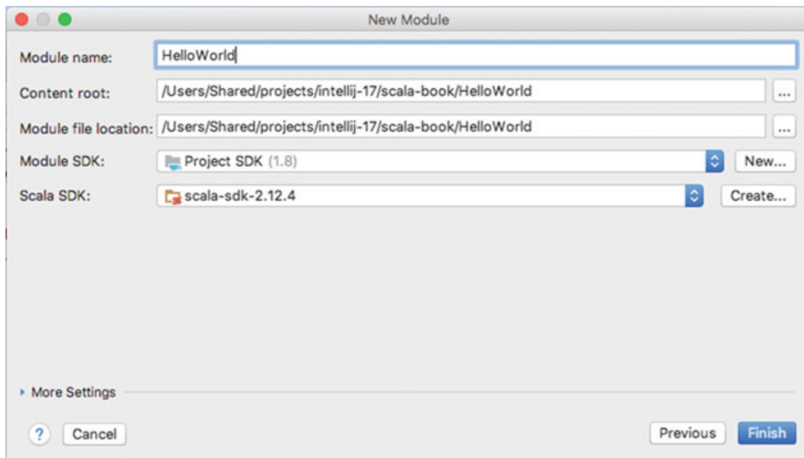


Fig. 7.1 IntelliJ IDEA IDE



Now click Finish.

You will now see the module listed under your project in the IDE.

We will add a new Scala object to this module. This is done by using the right mouse menu, from the `src` node in the Project Window tree (i.e. click on HelloWorld–>src and bring up the right mouse menu). Select New–>Scala Class as shown in Fig. 7.2.

Fig. 7.2 Selecting the Scala Class Wizard

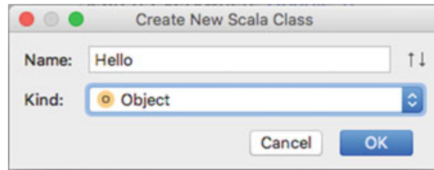


Fig. 7.3 Scala New Object Wizard

This will display the *Create New Scala Class* dialog. Enter the name of the Scala object to be created (e.g. 'Hello') and change the 'Kind' to Object. This is illustrated in Fig. 7.3.

And now click 'OK.

This results in a new file Hello.scala being created under *src,* and the basic structure of a Scala object will be displayed in the central code area (see Fig. 7.4).

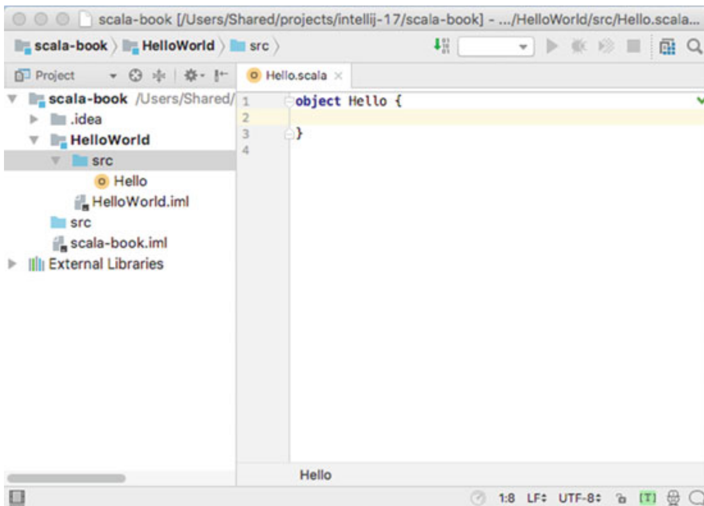


Fig. 7.4 Hello Object in the IDE

7.5 Implementing the Object

To implement the Hello Object we created above, type in the definition for the main method as shown below.

```
object Hello {
  def main(args: Array[String]): Unit = {
   println("Hello Scala")
  }
}
```

This says that the object *Hello* defines a 'main' method. A 'main' method is the entry point for this application. The main method takes an array of Strings (that is the type), and the parameter that this array is placed into is called *args* (although you can call the parameter whatever you like). The parameter is placed in parentheses.

Following the parameters is a ':' followed by the return type of 'Unit'. This indicates that this *method* does not return a value (Unit indicates no returned value). This is followed by an '=' and the definition of the body of the method (within the '{…}' brackets).

In this case all that the body of the method does is to printout the string "Hello Scala" using the function *println*, which is automatically made available to all code by the Scala environment.

7.6 Running the Application

To run the application, select the file *Hello.scala* in the Project Window. Using the right mouse menu select the *Run* menu option followed by the Scala Application option. This is illustrated in Fig. 7.5.

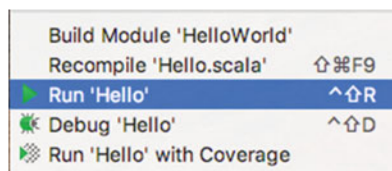


Fig. 7.5 Menu option to run the application

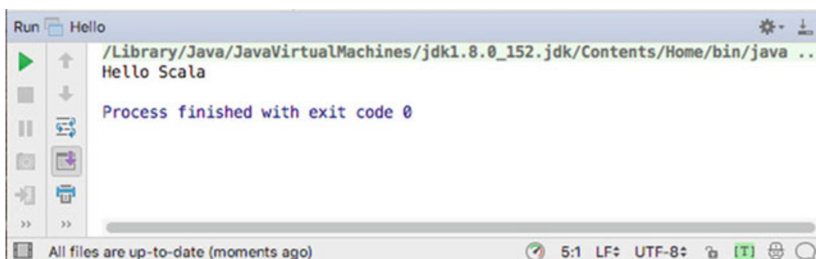


Fig. 7.6 Output from the Hello application

This will look for a *main* method in the object in *Hello.scala* and run that method. This will result in the "Hello Scala" string being printed out in the Console window (which is the output window for running applications within Eclipse). This is illustrated in Fig. 7.6.

7.6.1 Scala Interpreter Console

The Scala Interpreter Console is also available within the IntelliJ IDEA. This allows you to use the Scala interpreter (The REPL) directly from IntelliJ IDEA IDE. It automatically picks up your project contents and allows you to run Scala code (including the code you have written in your classes and objects) in the interpreter. This can be quiet useful for trying things out.

The Scala Console can be run from the right mouse menu for a Scala file. For example, select the Hello Object in the Project Window, and the from the right mouse menu, select 'Run Scala Console', as shown in Fig. 7.7.

The Scala interpreter is shown in Fig. 7.8. This interpreter is linked to the *helloworld* project and shows that you can type in an expression to the Evaluate box. This expression is evaluated, and the expression and the result are presented in the main window above the Evaluate box.

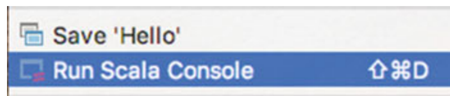


Fig. 7.7 Selecting the Scala Interpreter Console

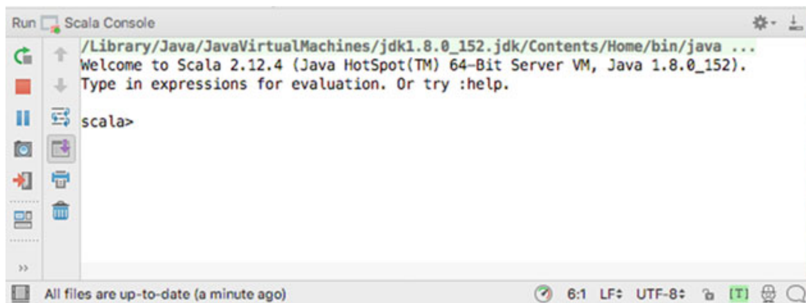


Fig. 7.8 Using the Scala Interpreter Console

7.7 Scala Classpath

Whatever your platform, you should be aware of the CLASSPATH environment variable. This variable tells Scala (actually the JVM) where to look for class definitions, so it should at least point to the runtime library and the current directory. It may also point to other directories in which you have defined classes. On Windows, you may change CLASSPATH in the autoexec.bat file by adding the following declaration:

```
SET CLASSPATH= .;c:\Scala\lib\classes.zip
```

Note that on a Windows machine the Classpath is a ';' separated list, whereas on a Unix machine it is a ':' separated list. Thus, for a Unix box you can define the CLASSPATH as:

```
setenv CLASSPATH
.:/usr/local/misc/Scala/lib/classes.zip
```

You should also add the Scala `bin` directory to your path. You should now be ready to use the Scala tools.

7.8 Compiling and Executing Scala

It is useful and instructive to actually look at what happens when you *compile* and *execute* a Scala program.

If your code is compiled successfully, the compiler will generate class files representing your Scala code such as `Hello.class` and Hello$.class. The files are located (by default) under the out directory of your project; for example, on a Mac, the directory listing is as shown in Fig. 7.9.

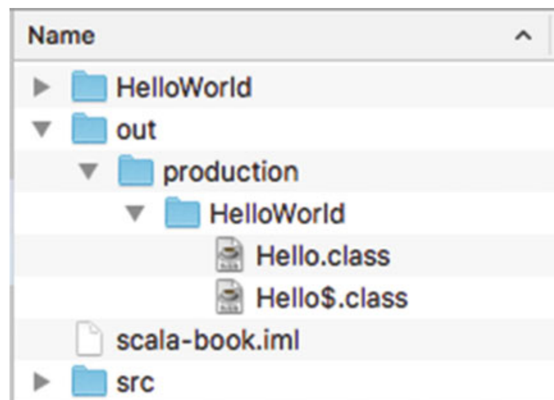


Fig. 7.9 Multiple .class files generated for a scala type

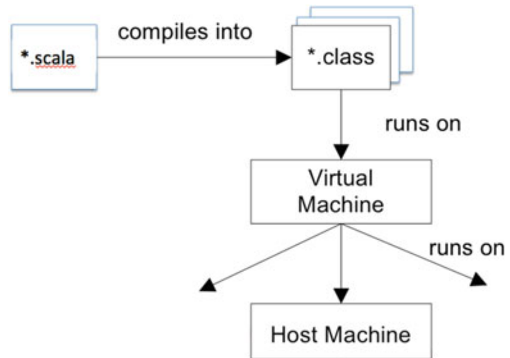


Fig. 7.10 Relationship between the scala source files and the Virtual Machine environment

The *.class* files represent the *compiled* version of the Scala object Hello.

So what has happened here—we haven't created any form of executable, rather we have created a set of class files. The effects of compiling your Scala code are illustrated in Fig. 7.10.

As can be seen from this figure, the compiler compiles the `.scala` files into.-class files. These in turn run on the Virtual Machine (this is what runs when you type in *scala* to the command prompt). The Virtual Machine (actually the JVM plus the Scala libraries) reads the class files and then runs them. This may involve interpreting the class files, compiling the class files to native code or a combination of the two depending upon the JVM being used. If we take the original approach the JVM interprets the class files. Thus the class files run on the JVM.

The JVM can be viewed as a virtual computer (that is one that exists only in software). Thus, Scala runs on a software computer. This software computer must then execute on the underlying host machine. This means that in effect the JVM has two aspects to it: the part that interprets Scala programs and a back end that must be ported to different platforms as required. This means that although the Scala programs you write are indeed "write once, run anywhere" the JVMs they run on need to be ported to each platform on which Scala will execute.

Although your Scala programs do not need to be re-written to run on Unix, NT, Linux, etc., the JVM does. This means that different JVMs on different platforms can (and do) have different bugs in them. It is therefore essential to test your Scala programs on all platforms that they are to be used on. In reality Scala is actually "write once, run anywhere, test everywhere" that you will use your Scala programs.

Note that there are multiple languages that can be compiled to JVM byte codes (and Scala is just one example, others include Ada, C#, Python), but that the byte code language was originally designed for Scala and thus must directly support the Scala language. As Scala was not the original source language for byte codes there must be a mapping from the concepts in Scala to these byte codes. Thus one Scala concept may generate one, two or three different implementations at the byte code level. In this case our hello world object results in two-byte code elements being

created called `Hello.class` and `Hello$.class`. For the most part you can ignore these details; they will typically become relevant only if you need to integrate Java into a Scala application (or Scala code into a Java application).

7.9 Memory Management

7.9.1 Why Have Automatic Memory Management?

Any discussion of Scala needs to consider how Scala handles memory. One of the many advantages of languages such as Java, C# and Scala over languages such as C ++ is that they automatically manage memory allocation and reuse.

It is not uncommon to hear C++ programmers complaining about spending many hours attempting to track down a particularly awkward bug only to find it was a problem associated with memory allocation or pointer manipulation. Similarly, a regular problem for C++ developers is that of memory creep, which occurs when memory is allocated but is not freed up. The application either uses all available memory or runs out of space and produces a runtime error.

Most of the problems associated with memory allocation in languages such as C ++ occur because programmers must concentrate not only on the (often complex) application logic but also on memory management. They must ensure that they allocate only the memory that is required and de-allocate it when it is no longer required. This may sound simple, but it is no mean feat in a large complex application.

An interesting question to ask is "why do programmers have to manage memory allocation?". There are few programmers today who would expect to have to manage the registers being used by their programs, although 20 or 30 years ago the situation was very different. One answer to the memory management question, often cited by those who like to manage their own memory, is that "it is more efficient, you have more control, it is faster and leads to more compact code". Of course, if you wish to take these comments to their extreme, then we should all be programming in assembler. This would enable us all to produce faster, more efficient and more compact code than that produced by Pascal or C++.

The point about high-level languages, however, is that they are more productive, introduce fewer errors, are more expressive and are efficient enough (given modern computers and compiler technology). The memory management issue is somewhat similar. If the system automatically handles the allocation and de-allocation of memory, then the programmer can concentrate on the application logic. This makes the programmer more productive, removes problems due to poor memory management and, when implemented efficiently, can still provide acceptable performance.

7.9.2 Memory Management in Scala

Scala provides automatic memory management. Essentially, it allocates a portion of memory as and when required. When memory is short, it looks for areas that are no longer referenced. These areas of memory are then freed up (de-allocated) so that they can be reallocated. This process is often referred to as "garbage collection".

The Virtual Machine (the JVM) uses an approach known as *mark and sweep* to identify objects that can be freed up. The garbage collection process searches from any root objects, i.e. objects from which the main method has been run, marking all the objects it finds. It then examines all the objects currently held in memory and deletes those objects that are not marked.

A second process invoked with garbage collection is memory compaction. This involves moving all the allocated memory blocks together so that free memory is contiguous rather than fragmented.

7.9.3 When Is Garbage Collection Performed?

Thegarbage collection process runs in its own thread. That is, it runs at the same time as other processes within the JVM. It is initiated when the ratio of free memory versus total memory passes a certain point.

You can also explicitly indicate to the JVM that you wish the garbage collector to run. This can be useful if you are about to start a process that requires a large amount of memory and you think that there may be unneeded objects in the system. You can do this using the sys object:

```
sys.runtime.gc()
```

However, calling `sys.runtime.gc` is only an indication to the compiler that you would like garbage collection to happen. There is no guarantee that it will run.

7.9.4 Checking the Available Memory

You can find out the current state of your system (with regard to memory) using the Runtime environment object. This object allows you to obtain information about the current free memory, total memory, etc.:

```
package com.jjh.scala.memory

object TestRuntime {

  def main(args: Array[String]): Unit = {

    val runtime = sys.runtime
    val freeMemory = runtime.freeMemory()
    val totalMemory = runtime.totalMemory()

    println("Total memory is " + totalMemory +
      " and free memory is " + freeMemory)

    println("Requesting system gc")
    sys.runtime.gc()

    val newFreeMemory = runtime.freeMemory()
    println("Total memory is " + totalMemory +
      " and free memory is now " + newFreeMemory)

  }
}
```

The result of running this application is shown in Fig. 7.11.

References

Scala
To download Scala (without the use of an IDE see www.scala-lang.org/downloads)

The Scala programming language home page
see http://www.scala-lang.org

The Scala mailing list
see http://listes.epfl.ch/cgi-bin/doc_en?liste=scala

The Scala wiki
see http://scala.sygneca.com/

Scala and.Net
http://www.scala-lang.org/old/node/10299

```
Run  TestRuntime
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/bin/java ..
Total memory is 192937984 and free memory is 184869224
Requesting system gc
Total memory is 192937984 and free memory is now 189514808

Process finished with exit code 0

Compilation completed successfully in 2s 114ms (a minute ago)   7:1  LF  UTF-8
```

Fig. 7.11 Output from memory application

Maven: Build tool for Scala and Scala
http://maven.apache.org/
https://code.google.com/p/esmi/wiki/ScalaMavenSupport

Ant: Build tool for Scala and Scala
http://ant.apache.org/
http://www.scala-lang.org/old/node/98

SBT (Simple Build Tool): Build tool for Scala and Scala
http://www.scala-sbt.org

Scala IDE for Eclipse
http://scala-ide.org/

Plugin for Existing Eclipse
If you already have an Eclipse installation and wish to add Scala to that then see the Scala plugin for Eclipse
see http://www.scala-lang.org/downloads/eclipse/index.html

Scala IDE IntelliJ
http://www.jetbrains.com/idea/features/scala.html

Scala IDE for NetBeans
http://sourceforge.net/projects/erlybird/files/nb-scala/

Further Reading

The Scala Language Specification 2.12
See http://scala-lang.org/files/archive/spec/2.12

The busy Scala developer's guide to Scala: Of traits and behaviours Using Scala's version of Java interfaces
see http://www.ibm.com/developerworks/Scala/library/j-scala04298.html

Scala for Java Programmers
http://docs.scala-lang.org/tutorials/scala-for-java-programmers.html