Atomic Scala

Bruce Eckel

Dianne Marsh

Mindview LLC, Crested Butte, CO

Contents

How to Use This Book

This book teaches the Scala language to both programming beginners and to those who have already programmed in another language.

Beginners: Start with the Introduction and move forward through each chapter (we call chapters *atoms*) just as you would any other book – including the Summary atoms, which will solidify your knowledge.

Experienced Programmers: Because you already understand the fundamentals of programming, we have prepared a "fast track" for you:

1. Read the Introduction.
2. Perform the installation for your platform following the appropriate atom. We assume you already have a programming editor and you can use a shell; if not read Editors and The Shell.
3. Read Running Scala and Scripting.
4. Jump forward to Summary 1; read it and solve the exercises.
5. Jump forward to Summary 2; read it and solve the exercises.
6. At this point, continue normally through the book, starting with Pattern Matching.

Introduction

This should be your first Scala book, not your last. We show you enough to become familiar and comfortable with the language – competent, but not expert. You'll write useful Scala code, but you won't necessarily be able to read all the Scala code you encounter.

When you're done, you'll be ready for more complex Scala books, several of which we recommend at the end of this one.

This is a book for a dedicated novice. "Novice" because you don't need prior programming knowledge, but "dedicated" because we're giving you just enough to figure it out on your own. We give you a foundation in programming and in Scala but we don't overwhelm you with the full extent of the language.

Beginning programmers should think of it as a game: You *can* get through, but you'll need to solve a few puzzles along the way. Experienced programmers can move rapidly through the book and find the place where they need to slow down and start paying attention.

Atomic Concepts

All programming languages consist of features that you apply to produce results. Scala is powerful: not only does it have more features, but you can usually express those features in numerous different ways. The combination of more features and more ways to express them can, if everything is dumped on you too quickly, make you flee, declaring that Scala is "too complicated."

It doesn't have to be.

If you know the features, you can look at any Scala code and tease out its meaning. Indeed, it's often easier to understand one page of Scala that produces the same effect as many pages of code in some other language, simply because you can see all the Scala code in one place.

Because it's easy to get overwhelmed, we teach you the language carefully and deliberately, using the following principles:

1. **Baby steps and small wins**. We cast off the tyranny of the chapter. Instead, we present each small step as an *atomic concept* or simply *atom*, which looks like a tiny chapter. A typical atom contains one or more small, runnable pieces of code and the output it produces. We describe what's new and different. We try to present only one new concept per atom.

2. **No forward references**. It often helps authors to say, "These features are explained in a later chapter." This confuses the reader, so we don't do it.

3. **No references to other languages**. We almost never refer to other languages (only when absolutely necessary). We don't know what languages you've learned (if any), and if we make an analogy to a feature in a language you don't understand, it just frustrates you.

4. **Show don't tell**. Instead of verbally describing a feature, we prefer examples and output that demonstrate what the feature does. It's better to see it in code.

5. **Practice before theory**. We try to show the mechanics of the language first, then tell why those features exist. This is backwards from "traditional" teaching, but it often seems to work better.

We've worked hard to make your learning experience the best it can be, but there's a caveat: For the sake of making things easier to understand, we will occasionally oversimplify or abstract a concept that you might later discover isn't precisely correct. We don't do this often, and only after careful consideration. We believe it helps you learn more easily now, and that you'll successfully adapt once you know the full story.

Cross-References

Whenever we refer to another atom in the book, the reference will have a grey box around it. A reference to the current atom looks like this: Introduction.

Sample the Book

In order to introduce you to the book and get you going in Scala, we've released a sample of the electronic book as a free distribution, which you can find at **AtomicScala.com**. We tried to make the sample large enough that it is useful by itself.

The complete book is for sale, both in print form and in eBook format. If you like what we've done in the free sample, please support us and help us continue our work by paying for what you use. We hope that the book helps and we greatly appreciate your sponsorship.

In the age of the Internet, it doesn't seem possible to control any piece of information. You'll probably be able to find the complete electronic version of this book in a number of places. If you are unable to pay for the book right now and you do download it from one of these sites, please "pay it forward." For example, help someone else learn the language once you've learned it. Or just help someone in any way that they need it. Perhaps sometime in the future you'll be better off, and you can come and buy something, or just donate to our tip jar at **AtomicScala.com/tip-jar**.

Example Code & Exercise Solutions

Available as a download from **www.AtomicScala.com**.

Seminars

An important goal of this book was to make the material usable in other forms – in particular, live seminars. Indeed, our experience giving seminars and workshops informed the way we created the book, with the intent of producing short lectures (to fit your attention span) and a stepwise learning process with many checkpoints along the way.

Scala is an amazing and elegant language. It's also powerful, and overwhelming if you try to absorb it all at once. Our goal is to present the language in small bites that you can quickly grasp, to give you a foundation on which to build more knowledge.

We want you to finish the course feeling strong and ready to learn more about Scala. To achieve this we select a subset of topics that, once learned, allow you to create useful and interesting programs – a base from which you can increase your understanding. We have carefully trimmed away topics that you don't need to know right away (but that you'll be able to acquire more easily from other books or more advanced courses).

We're careful to introduce topics before we use them, and we don't assume any programming language background. Books typically go "deep" on a topic in a single chapter. Instead, we divide topics into multiple atoms, building your knowledge piece by piece so you can understand and absorb each idea before moving to the next. We think you'll find this is a more natural way to learn.

We're not covering everything in the language – that's too much for a week. We're giving you what we consider the essentials in a way that you can understand them, so you have a strong basis for moving forward with Scala, either through self-study or more advanced courses.

One of the great things about the "Atomic" format is that it produces small lectures – we try to keep them less than 15 minutes each, within the limits of everyone's attention span. Shorter, more energetic lectures keep you engaged.

After each lecture, we'll give you exercises that develop and cement your knowledge.

Check **AtomicScala.com** for our public seminars. You can schedule In-house seminars with the "Contact Us" form on the website.

Consulting

Bruce Eckel believes that the foundation of the art of consulting is understanding the particular needs and abilities of your team and organization, and through that, discovering the tools and techniques that will serve and move you forward in the most optimal way. These include mentoring and assisting in multiple areas: helping you analyze your plan, evaluating strengths and risks, design assistance, tool evaluation and choice, language training, project bootstrapping workshops, mentoring visits during development, guided code walkthroughs, and research and spot training on specialized topics. To find out Bruce's availability and fitness for your needs, you can contact him at **MindviewInc@gmail.com**.

Conferences

Bruce has organized the *Java Posse Roundup*, an Open-Spaces conference, and the *Scala Summit* (www.ScalaSummit.com) a recurring Open-Spaces conference for Scala. Dianne has organized the Ann Arbor Scala Enthusiasts group, and is one of the organizers for *CodeMash*. Join the mailing list at **AtomicScala.com** to stay informed about our activities and where we are speaking.

Support Us

This was a big project, and it took us a lot of time and effort to produce this book and accompanying support materials. If you like this book and would like to see more things like it, please consider supporting us:

- *Blog, tweet, etc. and tell your friends*. This is a grassroots marketing effort so everything you can do will help.
- *Purchase an eBook or print version* of this book at **AtomicScala.com**.
- *Check* ***AtomicScala.com*** *for other support products* or apps.
- *Come to one of our public seminars*.

About Us

Bruce Eckel is the author of the multi-award-winning *Thinking in Java* and *Thinking in C++*, and a number of other books on computer programming. He's been in the computer industry for over 30 years, periodically gets frustrated and tries to quit, then something like Scala comes along, offering hope and drawing him back in. He's given hundreds of presentations around the world and enjoys putting on alternative conferences and events like *The Java Posse Roundup* and *Scala Summit*. He lives in Crested Butte, Colorado where he often acts in the community theatre. Although he will probably never be more than an intermediate-level skier or mountain biker, he finds these very enjoyable pursuits and considers them among his stable of life-projects, along with abstract painting. Bruce has a BS in applied physics, and an MS in computer engineering. He is currently studying organizational dynamics, trying to find a new way to organize companies so that working together becomes a joy; you can read about his struggles at **www.reinventing-business.com**, while his programming work is found at **www.mindviewinc.com**.

Dianne Marsh is the Director of Engineering for Cloud Tools at Netflix. Previously, she co-founded and ran SRT Solutions, a custom software development firm, before selling the company in 2013. Her expertise in software programming and technology includes manufacturing, genomics decision support and real-time processing applications. Dianne started her professional career using C and has since enjoyed languages including C++, Java, and C#, and is currently having a lot of fun using Scala. Dianne helped organize CodeMash (www.codemash.org), an all-volunteer developer conference bringing together programmers of various languages to learn from one another, and has been a board member of the Ann Arbor Hands-On Museum. She is active with local user groups and hosts several. She earned her Master of Science degree in computer science from Michigan Technological University. She's married to her best friend, has two fun young children and she talked Bruce into doing this book.

Acknowledgements

We thank the Programming Summer Camp 2011 attendees for their early comments and participation with the book. We would like to specifically thank Steve Harley, Alf Kristian Stoyle, Andrew Harmel-Law, Al Gorup, Joel Neely, and James Ward, all of whom have been generous with their time and comments. We also thank the many reviewers of this book in Google Docs format.

Bruce thanks Josh Suereth for all his technical help. Also Rumors Coffee and Tea House/Townie Books in Crested Butte for all the time he spent there working on this book, and Mimi and Jay at Bliss Chiropractic for regularly straightening him out during the process.

Dianne thanks her SRT business partner, Bill Wagner, and her employees at SRT Solutions for the time that she's spent away from the business. She also thanks Bruce for agreeing to write the book with her and keeping her on task throughout the process, even as he certainly grew weary of passive voice and punctuation errors. And special thanks go to her husband, Tom Sosnowski, for his tolerance and encouragement throughout this process.

Finally, thanks to Bill Venners and Dick Wall, whose "Stairway to Scala" class helped solidify our understanding of the language.

Dedication

To Julianna and Benjamin Sosnowski. You are amazing.

Copyrights

Editors

To install Scala, you may need to make changes to your system configuration files. To do this you'll need a program called an *editor*. You'll also need an editor to create the Scala program files – the code listings that we show in this book.

Programming editors vary from *Integrated Development Environments* (IDEs, like Eclipse and IntelliJ IDEA) to standalone programs. If you already have an IDE, you're free to use that for Scala, but in the interest of keeping things simple, we use the Sublime Text editor in our seminars and demonstrations. You can find it at www.sublimetext.com.

Sublime Text works on all platforms (Windows, Mac and Linux) and has a built-in Scala mode that is automatically invoked when you open a Scala file. It isn't a heavy-duty IDE so it doesn't get "too helpful," which is ideal for our purposes. On the other hand, it has some very handy editing features that you'll probably come to love. You can find more details on their site.

Although Sublime Text is commercial software, you can freely use it for as long as you like (you'll periodically get a pop-up window asking you to register, but this doesn't prevent you from continuing to use it). If you're like us, you'll soon decide that you want to support them.

There are many other editors; they are a subculture unto themselves and people even get into heated arguments about their merits. If you find one you like better, it's not too hard to change. The important thing is to choose one and get comfortable with it.

The Shell

If you haven't programmed before, you might never have used your operating system *shell* (also called the *command prompt* in Windows). The shell harkens back to the early days of computing when you did everything by typing commands and the computer responded by printing responses back at you – everything was text-based.

Although it might seem primitive in the age of graphical user interfaces, there are still a surprising number of valuable things you can accomplish with a shell, and we will make regular use of it, both as part of the installation process and to run Scala programs.

Starting a Shell

Mac: Click on the *Spotlight* (the magnifying-glass icon in the upper-right corner of the screen) and type "terminal." Click on the application that looks like a little TV screen (you might also be able to hit "Return"). This starts a shell in your home directory.

Windows: First, you must start the Windows Explorer to navigate through your directories. In **Windows 7**, click the "Start" button in the lower left corner of the screen. In the Start Menu search box area type "explorer" and then press the "Enter" key. In **Windows 8**, click Windows+Q, type "explorer" and then press the "Enter" key.

Once the Windows Explorer is running, move through the folders on your computer by double-clicking on them with the mouse. Navigate to the desired folder. Now click in the address bar at the top of the Explorer window, type "powershell" and press the "Enter" key. This will open a shell in the destination directory. (If Powershell doesn't start, go to the Microsoft website and install it from there).

In order to execute scripts in Powershell (which you must do to test the book examples), you must first change the Powershell *execution policy*.

On **Windows 7**, go to the “Control Panel” … “System and Security” … “Administrative Tools.” Right click on “Windows Powershell Modules” and select “Run as Administrator.”

On **Windows 8**, use Windows+W to bring up “Settings.” Select “Apps” and then type “power” in the edit box. Click on “Windows PowerShell” and then choose “Run as administrator.”

At the Powershell prompt, run the following command:

```
Set-ExecutionPolicy RemoteSigned
```

If asked, confirm that you want to change the execution policy by entering “Y” for Yes.

From now on, in any new Powershells you open, you can run Powershell scripts (files that end with “**.ps1**”) by typing **./** followed by the script’s file name at the Powershell prompt.

Linux: Press ALT-F2. In the dialog box that pops up, type **gnome-terminal** and press “Return.” This opens a shell in your home directory.

Directories

Directories are one of the fundamental elements of a shell. Directories hold files, as well as other directories. You can think about a directory like a tree with branches. If **books** is a directory on your system and it has two other directories as branches, for example **math** and **art**, we say that you have a directory **books** with two *subdirectories* **math** and

art. We refer to them as **books/math** and **books/art** since books is their *parent* directory.

Basic Shell Operations

The shell operations we show here are approximately identical across operating systems. Here are the most essential operations you can do in a shell, ones we will use in this book:

- **Change directory:** Use **cd** followed by the name of the directory where you want to move, or "**cd ..**" if you want to move up a directory. If you want to move to a new directory while remembering where you came from, use **pushd** followed by the new directory name. Then, to return to the previous directory, just say **popd**.
- **Directory listing: ls** shows you all the files and subdirectory names in the current directory. You can also use the wildcard '*' (asterisk) to narrow your search. For example, if you want to list all the files ending in ".scala," you say **ls *.scala**. If you want to list the files starting with "F" and ending in ".scala," you say **ls F*.scala**.
- **Create a directory:** use the mkdir ("make directory") command, followed by the name of the directory you want to create. For example, **mkdir books**.
- **Remove a file:** Use rm ("remove") followed by the name of the file you wish to remove. For example, **rm somefile.scala**.
- **Remove a directory:** use the rm -r command to remove the files in the directory and the directory itself. For example, **rm -r books**.
- **Repeat the last argument of the previous command line** (so you don't have to type it over again in your new command). Within your current command line, type **!$** in Mac/Linux and **$$** in Powershell.

- **Command history: history** in Mac/Linux and **h** in Powershell. This gives you a list of all the commands you've entered, with numbers you can refer to when you want to repeat a command.
- **Repeat a command**: Try the "up arrow" on all three operating systems, which moves through previous commands so you can edit and repeat them. In Powershell, **r** repeats the last command and **r n** repeats the **n**th command, where **n** is a number from the command history. On Mac/Linux, **!!** repeats the last command and **!n** repeats the nth command.
- **Unpacking a zip archive**: A file name ending with **.zip** is an archive containing a number of other files in a compressed format. Both Linux and the Mac have command-line unzip utilities, and it's possible to install a command-line unzip for Windows via the Internet. However, in all three systems you can use the graphical file browser (Windows Explorer, the Mac Finder, or Nautilus or equivalent on Linux) to browse to the directory containing your zip file. Then right-mouse-click on the file and select "Open" on the Mac, "Extract Here" on Linux, or "Extract all …" on Windows.

To learn more about your shell, search Wikipedia for "Windows Powershell," or "Bash_(Unix_shell)" for Mac/Linux.

Installation (Windows)

If you are installing from a USB flash drive for a workshop or seminar, see the special instructions at the end of this atom.

Scala runs on top of Java, so you must first install Java version 1.6 or later (you only need basic Java; the development kit also works but is not required).

Follow the instructions in The Shell to open a Powershell. Try running **java -version** at the prompt (regardless of the subdirectory you're in) to see whether Java is installed on your computer. If it is, you'll see something like the following (version numbers and actual text will vary):

```
java version "1.6.0_37"
Java(TM) SE Runtime Environment (build 1.6.0_37-b06-434-
10M3909)
Java HotSpot(TM) 64-Bit Server VM (build 20.12-b01-434,
mixed mode)
```

If you have at least Version 6 (also known as Java 1.6), you do not need to update.

If you need to install Java, first determine whether you're running 32-bit or 64-bit Windows.

In Windows 7, go to "Control Panel," then "System and Security," then "System." Under "System," you will see "System type," which will say either "32-bit Operating System" or "64-bit Operating System."

In Windows 8, press the Windows+W keys, and then type "System" to open the System application. The System application, titled "View basic information about your computer," will have a "System" area under the large Windows 8 logo. The system type will say either "32-bit Operating System" or "64-bit Operating System."

To install Java, follow the instructions here:

```
java.com/en/download/manual.jsp
```

This will attempt to detect whether it should install a 32-bit or 64-bit version of Java, but you can manually choose the correct version if necessary.

After installation, close all installation windows by pressing "OK," and then verify the Java installation by closing your old Powershell and running **java -version** in a new Powershell.

Set the Path

If your system still can't run **java -version** in Powershell, you must add the appropriate **bin** directory to your *path*, which tells the operating system where to find executable programs. For example, something like this would go at the end of the path:

```
;C:\Program Files\Java\jre6\bin
```

This assumes the default location for the installation of Java. If you put it somewhere else, use that path. Note the semicolon at the beginning – this separates the new directory from previous path directives.

In Windows 7, go to the control panel, select "System," then "Advanced System Settings," then "Environment Variables." Under "System variables," open or create **Path**, then add the installation directory "bin" folder shown above to the end of the "Variable value" string.

In Windows 8, use Windows+W for Settings. Type **env** in the edit box, and then choose "Edit Environment Variables for your account." Choose "Path," if it exists already, or add a new **Path** environment variable if it does not. Then add the installation directory "bin" folder shown above to the end of the "Variable value" string for **Path**.

Close your old Powershell window and start a new one to see the change.

Install Scala

In this book, we use Scala version 2.10, the latest available at the time. In general, the code in this book should also work on versions more recent than 2.10.

The main download site for Scala is:

```
    www.scala-lang.org/downloads
```

Choose the MSI installer which is custom-made for Windows. Once it downloads, execute the resulting file by double-clicking on it, then follow the instructions.

Note: If you are running Windows 8, you may see a message that says "Windows SmartScreen prevented an unrecognized app from starting. Running this app might put your PC at risk." Choose "More info" and then "Run anyway."

When you look in the default installation directory (**C:\Program Files (x86)\scala** or **C:\Program Files\scala**), it should contain:

```
    bin     doc     lib     misc
```

The installer will automatically add the **bin** directory to your path.

Now open a new Powershell and type

```
    scala -version
```

at the Powershell prompt. You'll see the version information for your Scala installation.

Source Code for the Book

We include a way to easily test the Scala exercises in this book with a minimum of configuration and download. Follow the links for the book's source code at **AtomicScala.com** and download the package (this will typically place it in your "Downloads" directory unless you have configured your system to place it elsewhere).

To unpack the book's source code, locate the file using the Windows explorer, then right-click on **AtomicScala.zip** and choose "Extract all …" To create the **C:\AtomicScala** directory and extract the contents into it, enter **C:\** as the destination folder. The **AtomicScala** directory now contains all the examples from the book, and the subdirectory **solutions**.

Set Your CLASSPATH

To run the examples, you'll first need to set your *CLASSPATH*, which is an *environment variable* that is used by Java (Scala runs atop Java) to locate code files. If you want to run code files from a particular directory, you must add that new directory to the CLASSPATH.

In Windows 7, go to "Control Panel," then "System and Security," then "System," then "Advanced System Settings," and finally "Environment Variables."

In Windows 8, open Settings with Windows-W, type “env” in the edit box, then choose “Edit Environment Variables for your account.”

Under “System variables,” open “CLASSPATH,” or create it if it doesn’t exist. Then add to the end of the “Variable value” string:

```
;C:\AtomicScala\code
```

This assumes the aforementioned location for the installation of the Atomic Scala code. If you put it somewhere else, use that path.

Open a Powershell window, change to the **C:\AtomicScala\code** subdirectory, and run:

```
scalac AtomicTest.scala
```

If everything is configured correctly, this creates a subdirectory **com\atomicscala** that includes several files, including:

```
AtomicTest$.class
AtomicTest.class
```

The source-code download package from **AtomicScala.com** includes a Powershell script, **testall.ps1**, that will test all of the code in the book using Windows. Before you run the script for the first time, you will need to tell Powershell that it’s OK. In addition to setting the Execution Policy as described in The Shell, you need to unblock the script. Using the Windows Explorer, go to the **C:\AtomicScala** directory. Right click on **testall.ps1**, choose “Properties” and then check “Unblock.”

Running ./**testall.ps1** tests all the code examples from the book. You will get a couple of errors when you do this and that’s fine; it’s due to things that we explain later in the book.

Exercises

These exercises will verify your installation.

1. Verify your Java version by typing **java –version** in a shell. The version must be 1.6 or greater.
2. Verify your Scala version by typing **scala** in a shell (This starts the REPL). The version must be 2.10 or greater.
3. Quit the REPL by typing **:quit**.

Installation (Mac)

If you are installing from a USB flash drive for a workshop or seminar, see the special instructions at the end of this atom.

Scala runs atop Java, and the Mac comes with Java pre-installed. Use **Software Update** on the Apple menu to check that you have the most up-to-date version of Java for your Mac, and update it if necessary. You need at least Java version 1.6. It is not necessary to update your Mac operating system software.

Follow the instructions in The Shell to open a shell in the desired directory. You can now type "**java -version**" at the prompt (regardless of the subdirectory you're in) and see the version of Java installed on your computer. You should see something like the following (version numbers and actual text will vary):

```
java version "1.6.0_37"
Java(TM) SE Runtime Environment (build 1.6.0_37-b06-434-
10M3909)
Java HotSpot(TM) 64-Bit Server VM (build 20.12-b01-434,
mixed mode)
```

If you see a message that the command is not found or not recognized, there's a problem with your Mac. Java should always be available in the shell.

Install Scala

In this book, we use Scala version 2.10, the latest available at the time. In general, the code in this book should also work on versions more recent than 2.10.

The main download site for Scala is:

```
www.scala-lang.org/downloads
```

Download the version with the **.tgz** extension. Click on the link on the web page, then select “open with archive utility.” This puts it in your “Downloads” directory and un-archives the file into a folder. (If you download without opening, you can open a new Finder window, right-click on the **.tgz** file, then choose “Open With -> Archive Utility”).

Rename the un-archived folder to “Scala” and then drag it to your home directory (the directory with an icon of a home, and is named whatever your user name is). If you don’t see a home icon, open “Finder,” choose “Preferences” and then choose the “Sidebar” icon. Check the box with the home icon next to your name in the list.

When you look at your **Scala** directory, it should contain:

```
bin     doc     examples     lib     man     misc     src
```

Set the Path

Now you need to add the appropriate **bin** directory to your *path*. Your path is usually stored in a file called **.profile** or **.bash_profile**, located in your home directory. We’ll assume that you’re editing **.bash_profile** from this point forward.

If neither file exists, create an empty file by typing:

```
touch ~/.bash_profile
```

Update your path by editing this file. Type:

```
open ~/.bash_profile.
```

Add the following at the end of all other PATH statement lines:

```
PATH="$HOME/Scala/bin/:${PATH}"
export PATH
```

By putting this at the end of the other PATH statements, when the computer searches for Scala it will find your version of Scala first, rather than others that might exist elsewhere in the path.

From that same terminal window, type:

```
source ~/.bash_profile
```

Now open a new shell and type

```
scala -version
```

at the shell prompt. You'll see the version information for your Scala installation.

Source Code for the Book

We include a way to easily test the Scala exercises in this book with a minimum of configuration and download. Follow the links for the book's source code at **AtomicScala.com** and download **AtomicScala.zip** into a convenient location on your computer.

Unpack the book's source code by double clicking on **AtomicScala.zip**. This will make an **AtomicScala** folder. Drag that to your home directory, using the directions above (for installing Scala).

The **~/AtomicScala** directory now contains all the examples from the book in the subdirectory **code**, and the subdirectory **solutions**.

Set Your CLASSPATH

The *CLASSPATH* is an *environment variable* that is used by Java (Scala runs atop Java) to locate Java program files. If you want to place code files in a new directory, you must add that new directory to the CLASSPATH.

Edit your **~/.profile** or **~/.bash_profile**, depending on where your path information is located, and add the following:

```
CLASSPATH="$HOME/AtomicScala/code:${CLASSPATH}"
export CLASSPATH
```

Open a new terminal window and change to the **AtomicScala** subdirectory by typing:

```
cd ~/AtomicScala/code
```

Now run:

```
scalac AtomicTest.scala
```

If everything is configured correctly, this creates a subdirectory **com/atomicscala** that includes several files, and in particular:

```
AtomicTest$.class
AtomicTest.class
```

Finally, test all the code in the book by running the **testall.sh** file that you will find there (part of the book's source-code download from **AtomicScala.com**) with:

```
./testall.sh
```

You will get a couple of errors when you do this and that's fine; it's due to things that we explain later in the book.

Exercises

These exercises will verify your installation.

1. Verify your Java version by typing **java –version** in a shell. The version must be 1.6 or greater.

2. Verify your Scala version by typing **scala** in a shell (This starts the REPL). The version must be 2.10 or greater.

3. Quit the REPL by typing **:quit**.

Installation (Linux)

If you are installing from a USB flash drive for a workshop or seminar, see the special instructions at the end of this atom.

In this book, we use Scala version 2.10, the latest available at the time. In general, the examples in this book should also work on versions more recent than 2.10.

Standard Package Installation

Important: The standard package installer may not install the most recent version of Scala. There is often a significant delay between a release of Scala and its inclusion in the standard packages. If the resulting version is not what you need, follow the instructions in the section titled "Install Recent Version From tgz File."

Ordinarily, you can use the standard package installer, which will also install Java if necessary, using one of the following shell commands (see The Shell):

Ubuntu/Debian: **sudo apt-get install scala**

Fedora/Redhat release 17+: **sudo yum install scala**

(Prior to release 17, Fedora/Redhat contains an old version of Scala, incompatible with this book).

Now follow the instructions in the next section to ensure that both Java and Scala are installed and that you have the right versions.

Verify Your Installation

Open a shell (see The Shell) and type "**java -version**" at the prompt. You should see something like the following (Version numbers and actual text will vary):

```
java version "1.7.0_09"
Java(TM) SE Runtime Environment (build 1.7.0_09-b05)
Java HotSpot(TM) Client VM (build 23.5-b02, mixed mode)
```

If you see a message that the command is not found or not recognized, add the java directory to the computer's execution path using the instructions in the section "Set the Path."

Test the Scala installation by starting a shell and typing "**scala -version**." This should produce Scala version information; if it doesn't, add Scala to your path using the following instructions.

Configure your Editor

If you already have an editor that you like, skip this section. If you chose to install Sublime Text 2, as we described in Editors, you must tell Linux where to find the editor. Assuming you have installed Sublime Text 2 in your home directory, create a symbolic link with the shell command:

```
sudo ln -s   ~/"Sublime Text 2"/sublime_text
/usr/local/bin/sublime
```

This allows you to edit a file named **filename** using the command:

```
sublime filename
```

Set the Path

If your system can't run **java -version** or **scala -version** from the console (terminal) command line, you may need to add the appropriate **bin** directory to your path.

Your path is usually stored in a file called **.profile** located in your home directory. We'll assume that you're editing **.profile** from this point forward.

Run **ls -a** to see if the file exists. If not, create a new file using the sublime editor, as described above, by running:

```
    sublime ~/.profile.
```

Java is typically installed in **/usr/bin**. Add Java's **bin** directory to your path if your location is different. The following **PATH** directive includes both **/user/bin** (for Java) and Scala's **bin**, assuming your Scala is in a **Scala** subdirectory off of your home directory (note that we use a fully qualified path name – not ~ or **$HOME** – for your home directory):

```
    export PATH=/usr/bin:/home/`whoami`/Scala/bin/:$PATH:
```

\`whoami\` (note the back quotes) inserts your username.

Note: Add this line at the end of the **.profile** file, after any other lines that set the **PATH**.

Next, type:

```
    source ~/.profile
```

to get the new settings (or close your shell and open a new one). Now open a new shell and type

```
    scala -version
```

at the shell prompt. You'll see the version information for your Scala installation.

If you get the desired version information from both **java -version** and **scala -version**, then you can skip the next section.

Install Recent Version from tgz File

Try running **java -version** to see if you already have Java 1.6 or greater installed. If not, go to **www.java.com/getjava**, click "Free Java Download" and scroll down to the download for "Linux" (there is also a "Linux RPM" but we'll just use the regular version). Start the download and ensure that you are getting a file that starts with **jre-** and ends with **.tar.gz** (You must also verify that you get the 32-bit or 64-bit version depending on which Linux you've installed).

That site contains detailed instructions via help links.

Move the file to your home directory, then start a shell in your home directory and run the command:

```
    tar  zxvf   jre-*.tar.gz
```

This creates a subdirectory starting with **jre** and ending with the version of Java you just installed. Below that is a **bin** directory. Edit your **.profile** (following the instructions earlier in this atom) and locate the very last **PATH** directive, if there is one. Add or modify your **PATH** so that Java's **bin** directory is the first one in your **PATH** (there are more "proper" ways to do this but we're being expedient). For example, the beginning of the **PATH** directive in your **~/.profile** file might look like:

```
export set PATH=/home/`whoami`/jre1.7.0_09/bin:$PATH: …
```

This way, if there are any other versions of Java on your system, the version you just installed will always be seen first.

Reset your **PATH** with the command:

```
source ~/.profile
```

(Or just close your shell and open a new one). Now you should be able to run **java -version** and see a version number that agrees with what you've just installed.

Install Scala

The main download site for Scala is **www.scala-lang.org/downloads**. Scroll through this page to locate the desired release number, and then download the one marked "Unix, Max OSX, Cygwin." The file will have an extension of **.tgz**. After it downloads, move the file into your home directory.

Start a shell in your home directory and run the command:

```
tar  zxvf   scala-*.tgz
```

This creates a subdirectory starting with **scala-** and ending with the version of Scala you just installed. Below that is a **bin** directory. Edit your **.profile** file and locate the **PATH** directive. Add the **bin** directory to your **PATH**, again before the **$PATH**. For example, the **PATH** directive in your **~/.profile** file might look like this:

```
export set
PATH=/home/`whoami`/jre1.7.0_09/bin:/home/`whoami`/scala-
2.10.0-RC3/bin:$PATH:
```

Reset your **PATH** with the command

```
    source ~/.profile
```

(Or just close your shell and open a new one). Now you should be able to run **scala -version** and see a version number that agrees with what you've just installed.

Source Code for the Book

We include a way to easily test the Scala exercises in this book with a minimum of configuration and download. Follow the links for the book's source code at **AtomicScala.com** into a convenient location on your computer.

Move **AtomicScala.zip** to your home directory using the shell command:

```
    cp AtomicScala.zip ~
```

Unpack the book's source code by running **unzip AtomicScala.zip**. This will make an **AtomicScala** folder.

The **~/AtomicScala** directory now contains all the examples from the book in a subdirectory **code**, and the subdirectory **solutions**.

Set Your CLASSPATH

Note: Sometimes (on Linux, at least) you don't need to set the CLASSPATH at all and everything still works right. Before setting your CLASSPATH, try running the **testall.sh** script (see below) and see if it's successful.

The *CLASSPATH* is an *environment variable* that is used by Java (Scala runs atop Java) to locate code files. If you want to place code files in a new directory, then you must add that new directory to the CLASSPATH. For example, this adds **AtomicScala** to your CLASSPATH when added to your **~/.profile**, assuming you installed into the **AtomicScala** subdirectory located off your home directory:

```
export
CLASSPATH="/home/`whoami`/AtomicScala/code:$CLASSPATH"
```

The changes to CLASSPATH will take effect if you run:

```
source ~/.profile
```

or if you open a new shell.

Verify that everything is working by changing to the **AtomicScala/code** subdirectory. Then run:

```
scalac AtomicTest.scala
```

If everything is configured correctly, this will create a subdirectory **com/atomicscala** that includes several files, and in particular:

```
AtomicTest$.class
AtomicTest.class
```

Finally, test all the code in the book by running:

```
chmod +x testall.sh
./testall.sh
```

You will get a couple of errors when you do this and that's fine; it's due to things that we explain later in the book.

Exercises

These exercises will verify your installation.

1. Verify your Java version by typing **java –version** in a shell. The version must be 1.6 or greater.
2. Verify your Scala version by typing **scala** in a shell (This starts the REPL). The version must be 2.10 or greater.
3. Quit the REPL by typing **:quit**.

Running Scala

The Scala interpreter is also called the REPL (for *Read-Evaluate-Print-Loop*). You get the REPL when you type **scala** by itself on the command line. You should see something like the following (it can take a few moments to start):

```
Welcome to Scala version 2.10.0-RC3 (Java HotSpot(TM) 64-
Bit Server VM, Java 1.7.0_09).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

The exact version numbers will vary depending on the versions of Scala and Java you've installed, but make sure that you're running Scala 2.10 or greater.

The REPL gives you immediate interactive feedback. For example, you can do arithmetic:

```
scala> 42 * 11.3
res0: Double = 474.6
```

res0 is the name Scala gave to the result of the calculation. **Double** means "double precision floating point number." A floating-point number can hold fractional values, and "double precision" refers to the number of significant places to the right of the decimal point that the number is capable of representing.

You can find out more by typing **:help** at the Scala prompt. To exit the REPL, type:

```
scala> :quit
```

Comments

A *Comment* is illuminating text that is ignored by Scala. There are two forms of comment. The **//** (two forward slashes) begins a comment that goes to the end of the current line:

```
47 * 42  // Single-line comment
47 + 42
```

Scala will evaluate the multiplication, but will ignore the **//** and everything after it until the end of the line. On the following line, it will pay attention again and perform the sum.

The multiline comment begins with a **/*** (a forward slash followed by an asterisk) and continues – including line breaks (usually called *newlines*) – until a ***/** (an asterisk followed by a forward slash) ends the comment:

```
47 + 42 /* A multiline comment
Doesn't care
about newlines */
```

It's possible to have code on the same line *after* the closing ***/** of a comment but it's confusing so people don't usually do it. In practice, you'll see the **//** comment used a lot more than the multiline comment.

Comments should add new information that isn't obvious from reading the code. If the comments just repeat what the code says, it becomes annoying (and people start ignoring your comments). When the code changes, programmers often forget to update comments, so it's a good practice to use comments judiciously, mainly for highlighting tricky aspects of your code.

Scripting

A *script* is a file filled with Scala code that you can run from the command-line prompt. Suppose you have a file named **myfile.scala** containing a Scala script. To execute that script from your operating system shell prompt, you enter:

```
scala myfile.scala
```

Scala will then execute all the lines in your script. This is more convenient than typing all of those lines into the Scala REPL.

Scripting makes it easy to quickly put together simple programs, so we will use it throughout much of this book (thus, you'll run the examples via The Shell). Scripting solves basic problems, such as making utilities for your computer. More sophisticated programs require the use of the *compiler*, which we'll explore when the time comes.

Using Sublime Text (from Editors), type in the following lines and save the file as **ScriptDemo.scala**:

```
// ScriptDemo.scala
println("Hello, Scala!")
```

We always begin a code file with a comment that contains the name of the file.

Assuming you've followed the instructions in the "Installation" atom for your computer's operating system, the book's examples are in a directory called **AtomicScala**. Although you can download the code, we urge you to type in the code from the book, since the hands-on experience may help you learn.

The above script has a single executable line of code.

```
println("Hello, Scala!")
```

When you run this script by typing (at the shell prompt):

```
scala ScriptDemo.scala
```

You should see:

```
Hello, Scala!
```

Now we’re ready to start learning about Scala.

Values

A *value* holds a particular type of information. You define a value like this:

```
val name = initialization
```

That is, the **val** keyword followed by the name (that you make up), an equals sign and the initialization value. The name begins with a letter or an underscore, followed by more letters, numbers and underscores. The dollar sign (**$**) is used for internal use, so don't use it in names you make up. Upper and lower case are distinguished (so **thisvalue** and **thisValue** are different).

Here are some value definitions:

```
// Values.scala

val whole = 11
val fractional = 1.4
val words = "A value"

println(whole, fractional, words)

/* Output:
(11,1.4,A value)
*/
```

The first line of each example in this book contains the name of the source code file as you will find it in the **AtomicScala** directory that you set up in your appropriate "Installation" atom. You will also see line numbers on all of our code samples. Line numbers do not appear in legal Scala code, so don't add them in your code. We use them merely as a convenience when describing the code.

We also format the code in this book to so it fits on an eBook reader page, which means that we sometimes add line breaks – to shorten the lines – where they would not otherwise be necessary in code.

On line 3, we create a value named **whole** and store 11 in it. Similarly, on line 4, we store the "fractional number" 1.4, and on line 5 we store some text (a *string*) in the value **words**.

Once you initialize a **val**, you can't change it (it is *constant*). Once we set **whole** to 11, for example, we can't later say:

```
whole = 15
```

If we do this, Scala complains, saying "error: reassignment to val."

It's important to choose descriptive names for your identifiers. This makes it easier for people to understand your code and often reduces the need for comments. Looking at the code snippet above, you have no idea what **whole** represents. If your program is storing the number 11 to represent the time of day when you get coffee, it's more obvious to others if you name it **coffeetime** and easier to read if it's **coffeeTime**.

In the first few examples of this book, we show the output at the end of the listing, inside a multiline comment. Note that **println** will take a single value, or a comma-separated sequence of values.

We include exercises with each atom from this point forward. The solutions are available at **AtomicScala.com**. The solution folders match the names of the atoms.

Exercises

Solutions are available at **www.AtomicScala.com**.

1. Store (and print) the value **17**.
2. Using the value you just stored (**17**), try to change it to **20**. What happened?
3. Store (and print) the value "**ABC1234**."
4. Using the value you just stored ("**ABC1234**"), try to change it to "**DEF1234**." What happened?
5. Store the value **15.56**. Print it.

Data Types

Scala distinguishes between different *types* of values. When you're doing a math problem, you just write the computation:

```
5.9 + 6
```

You know that when you add those numbers together, you will get another number. Scala does that too. You don't care that one is a fractional number (5.9), which Scala calls a **Double**, and the other is a whole number (6), which Scala calls an **Int**. When you do math by hand, you know that you will get a fractional number as a result, but you probably don't think about it very much. Scala categorizes these different ways of representing data into 'types' so it knows whether you're using the right kind of data. In this case, Scala creates a new value of type **Double** to hold the result.

Using types, Scala either adapts to what you need, as above, or if you ask it to do something silly it gives you an error message. For example, what if you use the REPL to add a number and a **String**:

```
scala> 5.9 + "Sally"
res0: String = 5.9Sally
```

Does that make any sense? In this case, Scala has rules that tell it how to add a **String** to a number. The types are important because Scala uses them to figure out what to do. Here, it appends the two values together and creates a **String** to hold the result.

Now try multiplying a **Double** and a **String**:

```
5.9 * "Sally"
```

Combining types this way doesn't make any sense to Scala, so it gives you an error.

In Values, we stored several different types, from numbers to letters. Scala figured out the type for us, based on how we used it. This is called *type inference*.

We can be more verbose and specify the type:

```
val name:type = initialization
```

That is, the **val** keyword followed by the name (that you make up), a colon, the type, and the initialization value. So instead of saying:

```
val n = 1
val p = 1.2
```

You can say:

```
val n:Int = 1
val p:Double = 1.2
```

When you specify the type explicitly, you are telling Scala that **n** is an **Int** and **p** is a **Double**, rather than letting it figure out the type.

Here are Scala's basic types:

```
// Types.scala

val whole:Int = 11
val fractional:Double = 1.4
// true or false:
val trueOrFalse:Boolean = true
val words:String = "A value"
val lines:String = """Triple quotes let
```

```
9    you have many lines
10   in your string"""
11
12   println(whole, fractional,
13     trueOrFalse, words)
14   println(lines)
15
16   /* Output:
17   (11,1.4,true,c,A value)
18   Triple quotes allow
19   you to have many lines
20   in your string
21   */
```

The **Int** data type is an *integer*, which means it only holds whole numbers. You can see this on line 3. To hold fractional numbers, as on line 4, use a **Double**.

A **Boolean** data type, as on line 6, can only hold the two special values **true** and **false**.

A **String** holds a sequence of characters. You can assign a value using a double-quoted string as on line 7, or if you have many lines and/or special characters, you can surround them with triple-double-quotes, as on lines 8-10 (this is a *multiline string*).

Scala uses type inference to figure out what you mean when you mix types. When you mix **Int**s and **Double**s using addition, for example, Scala decides the type to use for the resulting value. Try the following in the REPL:

```
scala> val n = 1 + 1.2
n: Double = 2.2
```

This shows that when you add an **Int** to a **Double**, the result becomes a **Double**. With type inference, Scala determines that **n** should be a

Double and ensures that it follows all the rules for **Double**s. It does this seemingly trivial bit of work for you, so you can focus on the more meaningful code.

Scala does a lot of type inference for you, as part of its strategy of doing work for the programmer. You can usually try leaving out the type declaration and see whether Scala will pick up the slack. If not, it will give you an error message. We'll see more of this as we go.

Exercises

Solutions are available at **www.AtomicScala.com**.

1. Store the value **5** as an **Int** and print the value.
2. Store (and print) the value "**ABC1234**" as a **String**.
3. Store the value **5.4** as a **Double**. Print it.
4. Store the value **true**. What type did you use? What did it print?
5. Store a multiline **String**. Does it print in multiple lines?
6. What happens if you try to store the **String** "**maybe**" in a **Boolean**?
7. What happens if you try to store the number **15.4** in an **Int** value?
8. What happens if you try to store the number **15** in a **Double** value? Print it.

Variables

In Values, you learned to create values that you set once and don't change. When this is too restrictive, you can use a *variable* instead of a value.

Like a value, a *variable* holds a particular type of information. But with a variable, you are allowed to change the data that is stored. You define a variable in exactly the same way as you define a value, except that you use the **var** keyword in place of the **val** keyword:

```
var name:type = initialization
```

The word *variable* describes something that can change (a **var**), while *value* indicates something that cannot change (a **val**).

Variables come in handy when data must change as the program is running. Choosing when to use variables (**var**) vs. values (**val**) comes up a lot in Scala. In general, your programs are easier to extend and maintain if you use **val**s. Sometimes, it's too complex to solve a problem using only **val**s, and for that reason, Scala gives you the flexibility of **var**s.

Note: Most programming languages have style guidelines, intended to help you write code that is easy for you and others to understand. When you define a value, for example, Scala style recommends that you leave a space between the **name:** and the **type**. Books have limited space and we've chosen to make the book more readable at the cost of some style guidelines. Scala doesn't care about this space. You can follow the Scala style guidelines, but we don't want to burden you with that before you're comfortable with the language. From this point forward in the book, we will conserve space by omitting the space.

Exercises

Solutions are available at **www.AtomicScala.com**.

1. Create an **Int** value (**val**) that is set to **5**. Try to update that number to **10**. What happened? How would you solve this problem?

2. Create an **Int** variable (**var**) named **v1** that is set to **5**. Update it to **10** and store in a **val** named **constantv1**. Did this work? Can you think of a time that this might be useful?

3. Using **v1** and **constantv1** from above, now set **v1** to **15**. Did the value of **constantv1** change?

4. Create a new **Int** variable (**var**) called **v2** initialized to **v1**. Set **v1** to **20**. Did the value of **v2** change?

Expressions

The smallest useful fragment of code in many programming languages is either a *statement* or an *expression*. They're different in a simple way.

Suppose you speak to a crowd of people about recycling, but don't actually do anything about it. You've made a "statement," but you haven't produced a result. In the same way, a statement in a programming language does not produce a result. In order for the statement to do something interesting, it must change the state of its surroundings. Another way of putting this is "a statement is called for its *side effects*" (that is, what it does *other* than producing a result). Making a statement about recycling might have the side effect of influencing others to recycle. As a memory aid, you can say that

> *A statement changes state*

One definition of "express" is "to force or squeeze out," as in "to express the juice from an orange." So

> *An expression expresses*

That is, it produces a result.

Essentially, everything in Scala is an expression. The easiest way to see this is in the REPL:

```
scala> val i = 1; val j = 2
i: Int = 1
j: Int = 2

scala> i + j
```

```
res1: Int = 3
```

Semicolons allow you to put more than one statement or expression on a line. The expression **i + j** produces a value – the sum.

You can also have multiline expressions surrounded by curly braces, as seen on lines 3-7:

```
1   // Expressions.scala
2
3   val c = {
4     val i1 = 2
5     val j1 = 4/i1
6     i1 * j1
7   }
8   println(c)
9   /* Output:
10  4
11  */
```

Line 4 is an expression that sets a value to the number 2. Line 5 is an expression that divides 4 by the value stored in **i1** (that is, 4 "divided by" 2) resulting in 2. Line 6 multiplies those values together, and the resulting value is stored in **c**.

What if an expression doesn't produce a result? The REPL answers the question via type inference:

```
scala> val e = println(5)
e: Unit = ()
```

The call to **println** doesn't produce a value, so the expression doesn't either. Scala has a special type for an expression that doesn't produce a value: **Unit**. You can get the same result with an empty set of curly braces:

```
scala> val f = {}
f: Unit = ()
```

As with the other data types, you can explicitly declare something as **Unit** when necessary.

Exercises

Solutions are available at **www.AtomicScala.com**.

1. Create an expression that initializes **feetPerMile** to **5280**.
2. Create an expression that initializes **yardsPerMile** by dividing **feetPerMile** by 3.0.
3. Create an expression that divides 2000 by **yardsPerMile** to calculate miles for someone who swam 2000 yards.
4. Combine the above three expressions into a multiline expression that returns miles.

Conditional Expressions

A *conditional* makes a choice. It tests an expression to see whether it's **true** or **false** and does something based on the result. A true-or-false expression is called a *Boolean*, after the mathematician George Boole who invented the logic behind such expressions. Here's a very simple conditional that uses the **>** (greater than) sign and shows the use of Scala's **if** keyword:

```
// If.scala

if(1 > 0) {
  println("It's true!")
}

/* Output:
It's true!
*/
```

The expression inside the parentheses of the **if** must evaluate to **true** or **false**. If it is **true**, the lines within the curly braces are executed.

We can create a Boolean expression separately from where it is used:

```
// If2.scala

val x:Boolean = { 1 > 0 }

if(x) {
  println("It's true!")
}

/* Output:
It's true!
*/
```

You can test for the opposite of the Boolean expression using the “not” operator ‘!’:

```
// If3.scala

val y:Boolean = { 11 > 12 }

if(!y) {
  println("It's false")
}

/* Output:
It's false
*/
```

Because **y** is **Boolean**, the **if** can test it directly by saying **if(y)**. If we want the opposite, we put the “not” operator in front, so **if(!y)** reads “if not y.”

The **else** keyword allows you to deal with both the **true** and **false** possibilities:

```
// If4.scala

val z:Boolean = false

if(z) {
  println("It's true!")
} else {
  println("It's false")
}

/* Output:
It's false
*/
```

The **else** keyword is only used in conjunction with **if**.

The entire **if** is itself an expression, which means it can produce a result:

```
// If5.scala

val result = {
  if(99 > 100) { 4 }
  else { 42 }
}
println(result)

/* Output:
42
*/
```

We'll learn more about conditionals in later atoms.

Exercises

Solutions are available at **www.AtomicScala.com**.

1. Set the values **a** and **b** to **1** and **5**, respectively. Write a conditional expression that checks to see if **a** is less than **b**. Print "a is less than b" or "a is not less than b."

2. Using **a** and **b**, above, write some conditional expressions to check if the values are less than 2 or greater than 2. Print the results.

3. Set the value **c** to 5. Modify the first exercise, above, to check if **a < c**. Then, check if **b < c** (where '**<**' is the less-than operator). Print the results.

Evaluation Order

Programming languages define the order in which operations are performed. Remember that if you mix addition, multiplication, subtraction, and division, there are rules about the order of evaluation. Thus,

```
45 + 5 * 6
```

is calculated as 5 times 6 plus 45, which equals 75, because the multiplication operation 5 * 6 is performed first, followed by the addition 30 + 45 to produce 75.

If you want 45 + 5 to be performed first, use parentheses:

```
(45 + 5) * 6
```

For a result of 300.

The same is true with programming languages. For example, let's calculate *body mass index* (BMI), which is weight in kilograms divided by height in meters squared. If you have a BMI of less than 18.5, you are underweight. Between 18.5-24.9 is normal weight. BMI of 25 and over is overweight.

```
// BMI.scala

val kg = 72.57 // 160 lbs
val heightM = 1.727 // 68 inches

val bmi = kg/(heightM * heightM)
if(bmi < 18.5) println("Underweight")
else if(bmi < 25) println("Normal weight")
else println("Overweight")
```

If you remove the parentheses on line 6, you divide **kg** by **heightM** then multiply that result by **heightM**. That's a much larger number, and the wrong answer.

Evaluation order is important for more than just math equations. Here, different evaluation order produces different results:

```scala
// EvaluationOrder.scala

val sunny = true
val hoursSleep = 6
val exercise = false
val temp = 55

val happy1 = sunny && temp > 50 ||
  exercise && hoursSleep > 7
println(happy1) // true

val sameHappy1 = (sunny && temp > 50) ||
  (exercise && hoursSleep > 7)
println(sameHappy1) // true

val notSame =
  (sunny && temp > 50 || exercise) &&
  hoursSleep > 7
println(notSame) // false
```

We introduce more *Boolean Algebra* here: The **&&** means "and" and it requires that *both* the Boolean expression on the left and the one on the right are **true** to produce a **true** result. In this case, the Boolean expressions are **sunny, temp > 50, exercise,** and **hoursSleep > 7**. The **||** means "or" and produces **true** if either the expression on the left or right of the operator is **true** (or if both are **true**).

Lines 8-9 read “It’s sunny *and* the temperature is greater than 50 *or* I’ve exercised *and* had more than 7 hours of sleep.” But does “and” have precedence over “or,” or vice versa?

Lines 8-9 uses Scala’s default evaluation order. This produces the same result as lines 12-13 (so, without parentheses, the “ands” are evaluated first, then the “or”). Lines 16-18 use parentheses to produce a different result; in that expression we will only be happy if we get at least 7 hours of sleep.

When you’re not sure what evaluation order that Scala will choose, use parentheses to force your intention. This also makes it clear to anyone reading your code.

BMI.scala uses **Double**s for the weight and height. Here’s a version using **Int**s (for English units instead of metric):

```
1   // IntegerMath.scala
2
3   val lbs = 160
4   val height = 68
5
6   val bmi = lbs / (height * height) * 703.07
7
8   if (bmi < 18.5) println("Underweight")
9   else if (bmi < 25) println("Normal weight")
10  else println("Overweight")
```

Scala implies that both **lbs** and **height** are integers (**Int**s) because the initialization values are integers (they have no decimal points). When you divide an integer by another integer, Scala produces an integer result. The standard way to deal with the remainder during integer division is *truncation*, which means “chop it off and throw it away” (there’s no rounding). So, if you divide 5 by 2 you get 2, and 7/10 is zero. When Scala calculates **bmi** on line 6, it divides 160 by 68*68 and gets zero. It then multiplies zero by 703.07 to get zero. We get unexpected results because of integer math. To avoid the problem, simply declare either **lbs** or **height** (or both, if you prefer) as **Double**. You can also tell Scala to infer **Double** by adding '**.0**' at the end of the initialization values.

Exercises

Solutions are available at **www.AtomicScala.com**.

1. Write an expression that evaluates to **true** if the sky is "sunny" and the temperature is more than 80 degrees.

2. Write an expression that evaluates to **true** if the sky is either "sunny" or "partly cloudy" and the temperature is more than 80 degrees.

3. Write an expression that evaluates to **true** if the sky is either "sunny" or "partly cloudy" and the temperature is either more than 80 degrees or less than 20 degrees.

4. Convert Fahrenheit to Celsius. Hint: first subtract 32, then multiply by 5/9. If you get 0, check to make sure you didn't do integer math.

5. Convert Celsius to Fahrenheit. Hint: first multiply by 9/5, then add 32. Use this to check your solution for the previous exercise.

Compound Expressions

In Expressions, you learned that nearly everything in Scala is an expression, and that expressions may contain one line of code, or multiple lines of code surrounded with curly braces. Now we'll differentiate between basic expressions, which don't need curly braces, and *compound expressions*, which must be surrounded by curly braces. A compound expression can contain any number of other expressions, including other curly-braced expressions.

Here's a simple example of a compound expression:

```
scala> val c = { val a = 11; a + 42 }
c: Int = 53
```

Notice that **a** is defined within this expression. The result of the last expression becomes the result of the compound expression; in this case, the sum of 11 and 42 as reported by the REPL. But what about **a**? Once you leave the compound expression (move outside the curly braces), you can't access **a**. It is a *temporary variable*, and is discarded once you exit the *scope* of the expression.

Here's a compound expression to determine whether a business is open or closed, based on the **hour**:

```
// CompoundExpressions1.scala

val hour = 6

val isOpen = {
  val opens = 9
  val closes = 20
  println("Operating hours: " +
    opens + " - " + closes)
10    if(hour >= opens && hour <= closes) {
11      true
12    } else {
13      false
14    }
15  }
16  println(isOpen)
17
18  /* Output:
19  Operating hours: 9 - 20
20  false
21  */
```

Notice on lines 8 and 9 that strings can be assembled using '+' signs. The Boolean **>=** operator returns **true** if the expression on the left side of the operator is *greater than or equal* to that on the right. Likewise, the Boolean operator **<=** returns **true** if the expression on the left side is *less than or equal* to that on the right. On line 10, we're checking to see if the hour that we define is between the opening time and closing time, so we combine the expressions with the Boolean **&&** (and).

This expression contains an additional level of curly-braced nesting:

```
1   // CompoundExpressions2.scala
2
3   val activity = "swimming"
4   val hour = 10
5
6   val isOpen = {
7     if(activity == "swimming" ||
8        activity == "ice skating") {
9       val opens = 9
10      val closes = 20
11      println("Operating hours: " +
12        opens + " - " + closes)
13      if(hour >= opens && hour <= closes) {
```

```
14        true
15      } else {
16        false
17      }
18    } else {
19      true
20    }
21  }
22
23  println(isOpen)
24  /* Output:
25  Operating hours: 9 - 20
26  true
27  */
```

The compound expression used by **CompoundExpressions1.scala** is inserted into lines 9-17, adding another expression layer, with an **if** expression on line 7 to verify whether we even need to check business hours. The Boolean **==** operator returns **true** if the expressions on each side of the operator are equivalent.

Expressions like **println** don't produce a result. Compound expressions don't necessarily produce a result, either:

```
scala> val e = { val x = 0 }
e: Unit = ()
```

Defining **x** doesn't produce a result, so the compound expression doesn't either; the REPL shows that the type of such an expression is **Unit**.

Here, expressions that produce results simplify the code:

```
// CompoundExpressions3.scala
val activity = "swimming"
val hour = 10
```

```
4
5   val isOpen = {
6     if(activity == "swimming" ||
7        activity == "ice skating") {
8       val opens = 9
9       val closes = 20
10      println("Operating hours: " +
11        opens + " - " + closes)
12      (hour >= opens && hour <= closes)
13    } else {
14      true
15    }
16  }
17
18  println(isOpen)
19  /* Output:
20  Operating hours: 9 - 20
21  true
22  */
```

Line 12 is the last expression in the "true" part of the **if** statement, so it becomes the result when the **if** evaluates to **true**.

Exercises

Solutions are available at **www.AtomicScala.com**.

1. In Exercise 3 of Conditional Expressions, you checked to see if **a** was less than **c**, and then if **b** was less than **c**. Repeat that exercise but this time use less than or equal.

2. Adding to your solution for the previous exercise, check first to see if both **a** and **b** are less than or equal to **c** using a single **if**. If they are not, then check to see if either one is less than or equal to **c**. If you set **a** to 1, **b** to 5, and **c** to 5, you should see "both are!" If, instead, you set **b** to 6, you should see "one is and one isn't!"

3. Modify **CompoundExpressions2.scala** to add a compound expression for **goodTemperature**. Pick a temperature (low and high) for each of the activities and determine whether you want to do each activity based on both temperature and whether a facility is open. Print the results of the comparisons to match the output described below. You can do this with the following code, once you define the expression for **goodTemperature**.

```
val doActivity = isOpen && goodTemperature
println(activity + ":" + isOpen + " && " +
goodTemperature + " = " + doActivity)
/* Output
(run 4 times, once for each activity):
swimming:false && false = false
walking:true && true = true
biking:true && false = false
couch:true && true = true
*/
```

4. Create a compound expression that determines whether you will do an activity. For example, you might say that you will do the running activity if the distance is less than 6 miles, the biking activity if the distance is less than 20 miles, and the swimming activity if the distance is less than 1 mile. You choose, and set up the compound expression. Test against various distances and various activities, and print your results. Here's some code to get you started.

```
val distance = 9
val activity = "running"
val willDo = // fill this in
/* Output
(run 3 times, once for each activity):
running: true
walking: false
biking: true
*/
```

Summary 1

This atom summarizes and reviews the atoms from Values through Compound Expressions. If you're an experienced programmer, this should be your first atom after installation. Beginning programmers should read this atom and perform the exercises as review.

If any information here isn't clear to you, you can go back and study the atom for that particular topic.

Values, Data Types, & Variables

Once a *value* is assigned, it cannot be reassigned. To create a value, use the **val** keyword followed by an identifier you choose, a colon, and the type for that value. Next, there's an equals sign and whatever you're assigning to the **val**:

```
val name:type = initialization
```

Scala's *type inference* can usually determine the type automatically based on the initialization. This produces a simpler definition:

```
val name = initialization
```

Thus, both of the following are valid:

```
val daysInFebruary = 28
val daysInMarch:Int = 31
```

A *variable* definition looks the same, with **var** substituted for **val**:

```
var name = initialization
var name:type = initialization
```

Unlike values, you can change the assignment to a variable, so the following are valid:

```
var hoursSpent = 20
hoursSpent = 25
```

However, the type can't be changed, so you'll get an error if you say:

```
hoursSpent = 30.5
```

Expressions & Conditionals

The smallest useful fragment of code in most programming languages is either a *statement* or an *expression*. They're different in a simple way.

> *A statement changes state*
> *An expression expresses*

That is, an expression produces a result, while a statement does not. Because it doesn't return anything, a statement must change the state of its surroundings in order to do anything useful.

Almost everything in Scala is an expression. Using the REPL:

```
scala> val hours = 10
scala> val minutesPerHour = 60
scala> val minutes = hours * minutesPerHour
```

In each case, everything to the right of the '**=**' is an expression, which produces a result that is assigned to the **val** on the left.

Some expressions, like **println**, don't seem to produce a result. Scala has a special **Unit** type for these:

```
scala> val result = println("???")
???
result: Unit = ()
```

Conditional expressions can have both **if** and **else** expressions. The entire **if** is itself an expression, which means it can produce a result:

```
scala> if (99 < 100) { 4 } else { 42 }
res0: Int = 4
```

Because we didn't create a **var** or **val** identifier to hold the result of this expression, the REPL assigned the result to the temporary variable **res0**. You can specify your own value:

```
scala> val result = if (99 < 100) { 4 } else { 42 }
result: Int = 4
```

When entering multiline expressions in the REPL, it's helpful to put it into *paste mode* with the **:paste** command. This delays interpretation until you enter **CTRL-D**. Paste mode is especially useful when copying and pasting chunks of code into the REPL.

Evaluation Order

When you're not sure what order Scala will choose for evaluating expressions, use parentheses to force your intention. This also makes it clear to anyone reading your code. Understanding evaluation order helps you to decipher what a program does, both with logical operations (Boolean expressions) and with mathematical operations.

When you divide an **Int** with another **Int**, Scala produces an **Int** result, and any remainder is truncated. So 1/2 produces 0. If a **Double** is involved, the **Int** is *promoted* to **Double** before the operation, so 1.0/2 produces 0.5.

You might expect the following to produce 3.4:

```
scala> 3.0 + 2/5
res1: Double = 3.0
```

But it doesn't. Because of evaluation order, Scala divides 2 by 5 first, and integer math produces 0, yielding a final answer of 3.0. The same evaluation order *does* produce the expected result here:

```
scala> 3 + 2.0/5
res3: Double = 3.4
```

2.0 divided by 5 produces 0.4. The 3 is promoted to a **Double** because we add it to a **Double** (0.4), which gives 3.4.

Compound Expressions

Compound expressions are surrounded by curly braces. A compound expression can contain any number of other expressions, including other curly-braced expressions. For example:

```
1   // CompoundBMI.scala
2   val lbs = 150.0
3   val height = 68.0
4   val weightStatus = {
5     val bmi = lbs/(height * height) * 703.07
6     if(bmi < 18.5) "Underweight"
7     else if(bmi < 25) "Normal weight"
8     else "Overweight"
9   }
10  println(weightStatus)
```

When you define a value inside an expression, such as **bmi** on line 5, the value is not accessible outside the scope of the expression. Notice, that **lbs** and **height** are accessible inside the compound expression.

The result of the compound expression is whatever is produced by its last expression; in this case, the **String** "Normal weight."

Experienced programmers should go to Summary 2 after working the following exercises.

Exercises

Solutions are available at **www.AtomicScala.com**.

Work exercises 1-8 in the REPL.

1. Store and print an **Int** value.
2. Try to change the value. What happened?
3. Create a **var** and initialize it to an **Int**, then try reassigning to a **Double**. What happened?
4. Store and print a **Double**. Did you use type inference? Try declaring the type.
5. What happens if you try to store the number **15** in a **Double** value?
6. Store a multiline **String** (see Data Types) and print it.
7. What happens if you try to store the **String** "**maybe**" in a **Boolean**?
8. What happens if you try to store the number **15.4** in an **Int** value?
9. Modify **weightStatus** in **CompoundBMI.scala** to return **Unit** instead of **String**.
10. Modify **CompoundBMI.scala** to return an **idealWeight** based on a BMI of 22.0. Hint: idealWeight = bmi * (height * height) / 703.07

Methods

A *method* is a mini-program packaged under a name. When you use a method (usually described as *invoking a method*), this mini-program is executed. A method combines a group of activities into a single name, and is the most basic way to organize your programs.

Ordinarily, you pass information into a method, and the method uses that information to calculate a result, which it returns to you. The basic form of a method in Scala is:

```
def methodName(arg1:Type1, arg2:Type2, …):returnType = {
  lines of code
  result
}
```

All method definitions begin with the keyword **def**, followed by the method name and the *argument list* in parentheses. The arguments are the information that you pass into the method, and each one has a name followed by a colon and the type of that argument. The closing parenthesis of the argument list is followed by a colon and the type of the result that the method produces when you call it. Finally, there's an equal sign, to say "here's the method body itself." The lines of code in the method body are enclosed in curly braces, and the last line is the result that the method returns to you when it's finished. Note that this is the same behavior we just described in Compound Expressions: a method body is an expression.

You don't need to say anything special to produce the result; it's just whatever is on the last line in the method. Here's an example:

```
// MultiplyByTwo.scala

def multiplyByTwo(x:Int):Int = {
```

```
4     println("Inside multiplyByTwo")
5     x * 2 // Return value
6   }
7
8   val r = multiplyByTwo(5) // Method call
9   println(r)
10  /* Output:
11  Inside multiplyByTwo
12  10
13  */
```

On line 3 you see the **def** keyword, the method name, and an argument list consisting of a single argument. Note that declaring arguments is just like declaring **val**s: the argument name, a colon, and the type returned from the method. Thus, this method takes an **Int** and returns an **Int**. Lines 4 and 5 are the body of the method. Note that line 5 performs a calculation and since it's the last line, the result of that calculation becomes the result of the method.

Line 8 runs the method by *calling* it with an appropriate argument, and then captures the result into the value **r**. You can see how the method call mimics the form of its declaration: the method name, followed by arguments inside parentheses.

Observe that **println** is also a method call – it just happens to be a method defined by Scala.

All the lines of code in a method (and you can put in a lot of code) are now executed by a single call, using the method name **multiplyByTwo** as an abbreviation for that code. This is why methods are the most basic form of simplification and code reuse in programming. You can also think of a method as an expression with substitutable values (the arguments).

Let's look at two more method definitions:

```
 1  // AddMultiply.scala
 2
 3  def addMultiply(x:Int,
 4    y:Double, s:String):Double = {
 5    println(s)
 6    (x + y) * 2.1
 7  }
 8
 9  val r2:Double = addMultiply(7, 9,
10    "Inside addMultiply")
11  println(r2)
12
13  def test(x:Int, y:Double,
14    s:String, expected:Double):Unit = {
15    val result = addMultiply(x, y, s)
16    assert(result == expected,
17      "Expected " + expected +
18      " Got " + result)
19    println("result: " + result)
20  }
21
22  test(7, 9, "Inside addMultiply", 33.6)
23
24  /* Output:
25  Inside addMultiply
26  33.6
27  Inside addMultiply
28  result: 33.6
29  */
```

addMultiply takes three arguments of three different types. It prints its third argument, a **String**, and returns a **Double** value, the result of the calculation on line 6.

Line 13 begins another method, defined only to test the **addMultiply** method. In previous atoms, we printed the output and relied on ourselves to catch any discrepancies. That's unreliable; even in a book where we scrutinize the code over and over we've learned that visual inspection can't be trusted to find errors. So the **test** method compares the result of **addMultiply** with an expected result and complains loudly if the two don't agree.

The call to **assert** on line 16 is a method defined by Scala. It takes a Boolean expression and a **String** message (which we build up using **+**'s). If the expression is **false**, Scala prints the message and stops executing code in the method. This is *throwing an exception*, and Scala prints out a lot of information to help you figure out what happened, including the line number where the exception happened. Try it – on line 22 change the last argument (the expected value) to 40.1. You should see something like the following:

```
Inside addMultiply
33.6
Inside addMultiply
java.lang.AssertionError: assertion failed: Expected 40.1
Got 33.6
 at scala.Predef$.assert(Predef.scala:173)
 at Main$$anon$1.test(AddMultiply.scala:16)
 at Main$$anon$1.<init>(AddMultiply.scala:22)
 at Main$.main(AddMultiply.scala:1)
 at Main.main(AddMultiply.scala)
         [many more lines deleted here]
```

Notice that if the **assert** fails then line 19 never runs; that's because the exception aborts the program's execution.

There's more to know about exceptions, but for now just treat them as something that produces error messages.

Note that **test** returns nothing, so we explicitly declare the return type as **Unit** on line 14. A method that doesn't return a result is called for its side effects – whatever it does *other* than returning something that you can use.

When writing methods, you should choose descriptive names to make reading the code easier and to reduce the need for code comments. Often we won't be as explicit as we would prefer in this book because we're constrained by line widths.

If you read other Scala code, you'll see many different ways to write methods in addition to the form shown in this atom. Scala is very expressive this way and it saves effort when writing and reading code. However, it can be confusing to see all these forms right away, when you're just learning the language, so for now we'll use this form and introduce the others after you're more comfortable with Scala.

Exercises

Solutions are available at **www.AtomicScala.com**.

1. Create a method **getSquare** that takes an **Int** argument and returns its square. Print your answer. Test using the following code.

```
val a = getSquare(3)
assert(/* fill this in */)
val b = getSquare(6)
assert(/* fill this in */)
val c = getSquare(5)
assert(/* fill this in */)
```

2. Create a method **getSquareDouble** that takes a **Double** argument and returns its square. Print your answer. How does this differ from Exercise 1? Satisfy the following code to check your solutions.

```
val sd1 = getSquareDouble(1.2)
assert(1.44 == sd1, "Your message here")
val sd2 = getSquareDouble(5.7)
assert(32.49 == sd2, "Your message here")
```

3. Create a method **isArg1GreaterThanArg2** that takes two **Double** arguments. Return **true** if the first argument is greater than the second. Return **false** otherwise. Print your answer. Satisfy the following:
   ```
   val t1 = isArg1GreaterThanArg2(4.1, 4.12)
   assert(/* fill this in */)
   val t2 = isArg1GreaterThanArg2(2.1, 1.2)
   assert(/* fill this in */)
   ```

4. Create a method **getMe** that takes a **String** and returns the same **String**, but all in lowercase letters (There's a **String** method called **toLowerCase**). Print your answer. Satisfy the following:
   ```
   val g1 = getMe("abraCaDabra")
   assert("abracadabra" == g1,
     "Your message here")
   val g2 = getMe("zyxwVUT")
   assert("zyxwvut"== g2, "Your message here")
   ```

5. Create a method **addStrings** that takes two **String**s as arguments, and returns the **String**s appended (added) together. Print your answer. Satisfy the following:
   ```
   val s1 = addStrings("abc", "def")
   assert(/* fill this in */)
   val s2 = addStrings("zyx", "abc")
   assert(/* fill this in */)
   ```

6. Create a method **manyTimesString** that takes a **String** and an **Int** as arguments and returns the **String** duplicated that many times. Print your answer. Satisfy the following:
   ```
   val m1 = manyTimesString("abc", 3)
   assert("abcabcabc" == m1,
     "Your message here")
   val m2 = manyTimesString("123", 2)
   assert("123123" == m2, "Your message here")
   ```

7. In the exercises for Evaluation Order, you calculated body mass index (BMI) using weight in pounds and height in inches. Rewrite

as a method. Satisfy the following:

```
val normal = bmiStatus(160, 68)
assert("Normal weight" == normal,
  "Expected Normal weight, Got " + normal)
val overweight = bmiStatus(180, 60)
assert("Overweight" == overweight,
  "Expected Overweight, Got " +
  overweight)
val underweight = bmiStatus(100, 68)
assert("Underweight" == underweight,
  "Expected Underweight, Got " +
  underweight)
```

Classes & Objects

Objects are the foundation for numerous modern languages, including Scala. In an *object-oriented* (OO) programming language, you think about “nouns” in the problem you’re solving, and translate those nouns to objects, which can hold data and perform actions. An object-oriented language is oriented towards the creation and use of objects.

Scala isn’t just object-oriented; it’s also *functional*. In a functional language, you think about “verbs,” the actions that you want to perform, and you typically describe these as mathematical equations.

Scala differs from many other programming languages in that it supports both object-oriented and functional programming. This book focuses on objects and only introduces a few of the functional subjects.

Objects contain **val**s and **var**s to store data (these are called *fields*) and they perform operations using Methods. A *class* defines fields and methods for what is essentially a new, user-defined data type. Making a **val** or **var** of a class is called *creating an object* or *creating an instance of an object*. We even refer to instances of built-in types like **Double** or **String** as objects.

Consider Scala’s **Range** class:

```
val r1 = Range(0, 10)
val r2 = Range(5, 7)
```

Each object has its own piece of storage in memory. For example, **Range** is a class, but a particular range **r1** from 0 to 10 is an object. It’s distinct from another range **r2** from 5 to 7. So we have a single **Range** *class*, of which there are two objects or instances.

Classes can have many operations/methods. In Scala, it's easy to explore classes using the REPL, which has the valuable feature of *code completion*. This means that if you start typing something and then hit the TAB key, the REPL will attempt to complete what you're typing. If it can't complete it, it will give you a list of options. We can find the possible operations on any class this way (the REPL will give lots of information – you can ignore the things you see here that we haven't talked about yet).

Let's look at **Range** in the **REPL**. First, we create an object called **r** of type **Range**:

```
scala> val r = Range(0, 10)
```

Now if we type the identifier name followed by a dot, then press TAB, the REPL will show us the possible completions:

```
scala> r.(PRESS THE TAB KEY)
++                     ++:
+:                     /:
/:\                    :+
:\                     addString
aggregate              andThen
apply                  applyOrElse
asInstanceOf           by
canEqual               collect
collectFirst           combinations
companion              compose
contains               containsSlice
copyToArray            copyToBuffer
corresponds            count
diff                   distinct
drop                   dropRight
dropWhile              end
endsWith               exists
```

```
filter                filterNot
find                  flatMap
flatten               fold
foldLeft              foldRight
forall                foreach
genericBuilder        groupBy
grouped               hasDefiniteSize
head                  headOption
inclusive             indexOf
indexOfSlice          indexWhere
indices               init
inits                 intersect
isDefinedAt           isEmpty
isInclusive           isInstanceOf
isTraversableAgain    iterator
last                  lastElement
lastIndexOf           lastIndexOfSlice
lastIndexWhere        lastOption
length                lengthCompare
lift                  map
max                   maxBy
min                   minBy
mkString              nonEmpty
numRangeElements      orElse
padTo                 par
partition             patch
permutations          prefixLength
product               reduce
reduceLeft            reduceLeftOption
reduceOption          reduceRight
reduceRightOption     repr
reverse               reverseIterator
reverseMap            run
runWith               sameElements
scan                  scanLeft
scanRight             segmentLength
```

```
seq                 size
slice               sliding
sortBy              sortWith
sorted              span
splitAt             start
startsWith          step
stringPrefix        sum
tail                tails
take                takeRight
takeWhile           terminalElement
to                  toArray
toBuffer            toIndexedSeq
toIterable          toIterator
toList              toMap
toSeq               toSet
toStream            toString
toTraversable       toVector
transpose           union
unzip               unzip3
updated             validateRangeBoundaries
view                withFilter
zip                 zipAll
```

There are a surprising number of operations available for a **Range**; some are simple and obvious, like **reverse**, and others appear to require more learning before you can use them. If you try calling some of those, the REPL will tell you that you need arguments. To know enough to call those operations, you can look them up in the Scala documentation, which we introduce in the following atom.

A warning is in order here. Although the REPL is a very useful tool, it has its flaws and limits. In particular, it will often *not* show every possible completion. Lists like the above are helpful when getting started, but you shouldn't assume that it's exhaustive – the Scala documentation might include other features. In addition, the REPL and scripts will sometimes have behavior that is not proper for regular Scala programs.

A **Range** is a kind of object, and a defining characteristic of objects is that you can perform operations on them. Instead of "performing an operation," we sometimes say *sending a message* or *calling a method*. You call an operation on an object by giving the object identifier, then a dot, then the name of the operation. Since **reverse** is a method that the REPL says is defined for range, you can call it by saying **r.reverse**, which just reverses the order of the **Range** we previously created, resulting in (9,8,7,6,5,4,3,2,1,0).

For now, it's enough to know what an object is and how to use it. Soon you'll learn to define your own classes.

Exercises

Solutions are available at **www.AtomicScala.com**.

1. Create a **Range** object and print the **step** value. Create a second **Range** object with a step value of 2 and then print the **step** value. What's different?

2. Create a **String** object initialized to **"This is an experiment"** and call the **split** method on it, passing a space (**" "**) as the argument to the **split** method.

3. Create a **String** object **s1** (as a **var**) initialized to **"Sally"**. Create a second **String** object **s2** (as a **var**) initialized to **"Sally"**. Use **s1.equals(s2)** to determine if the two **String**s are equivalent. If they are, print "s1 and s2 are equal," otherwise print "s1 and s2 are not equal."

4. Building from Exercise 3, set **s2** to **"Sam"**. Do the strings match? If they match, print "s1 and s2 are equal." If they do not match, print "s1 and s2 are not equal." Is **s1** still set to **"Sally"**?

5. Building from Exercise 3, create a **String** object **s3** as a result of calling **toUpperCase** on **s1**. Call **contentEquals** to compare the Strings **s1** and **s3**. If they match, print “s1 and s3 are equal.” If they do not match, print “s1 and s3 are not equal.” Hint: use **s1.toUpperCase**.

ScalaDoc

Scala provides an easy way to get documentation about classes. While the REPL provides you with a way to see all the available operations for a class, ScalaDoc provides much more detail. It's helpful to keep a window open with the REPL running so you can do quick experiments when you have a question, and another window with the documentation so you can rapidly look things up.

The documentation can be installed on your machine (see below), or you can find it online at:

```
www.scala-lang.org/api/current/index.html
```

Try typing **Range** into the upper-left search box to see the results directly below. You'll see several items that contain the word "Range." Click on **Range**; this will cause the right-hand window to display all the documentation for the **Range** class. Note that the right-hand window also has its own search box partway down the page. Type one of the operations you discovered in the previous atom into **Range**'s search box, and scroll down to see the results. Although you won't understand most of it at this time, it's very helpful to get used to the Scala documentation so you can become comfortable looking things up.

If the installation process you used didn't give you the option to install the documentation locally, you can download it from **www.scala-lang.org** and select the "Documentation" menu item, then "Scala API" and "Download Locally." On the page that comes up, look for "Scala API." (The abbreviation *API* stands for *Application Programming Interface*).

Note: As of this writing, there's an error of omission in the ScalaDoc. Some of the classes Scala uses are Java classes, and they were dropped from the ScalaDoc as of 2.8. **String** is an example of a Java class that we often use in this book, and that Scala programmers use as if it were a Scala class. Here's a link to the corresponding (Java) documentation for **String**:

```
docs.oracle.com/javase/6/docs/api/java/lang/String.html
```

Creating Classes

As well as using predefined types like **Range**, you can create your own types of objects. Indeed, creating new types comprises much of the activity in object-oriented programming. You create new types by defining *classes*.

An object is a piece of the solution for a problem you're trying to solve. Start by thinking of objects as expressing concepts. As a first approximation, if you discover a "thing" in your problem, you can represent that thing as an object in your solution. For example, suppose you are creating a program that manages animals in a zoo. Each animal becomes an object in your program.

It makes sense to categorize the different types of animals based on how they behave, their needs, animals they get along with and those they fight with – everything (that you care about for your solution) that is different about a species of animal should be captured in the classification of that animal's object. Scala provides the **class** keyword to create new types of objects:

```
// Animals.scala

// Create some classes:
class Giraffe
class Bear
class Hippo

// Create some objects:
val g1 = new Giraffe
val g2 = new Giraffe
val b = new Bear
val h = new Hippo

// Each new object is unique:
```

```
15  println(g1)
16  println(g2)
17  println(h)
18  println(b)
```

Begin with **class**, followed by the name – that you make up – of your new class. The class name must begin with a letter (A-Z, upper or lower case), but can include things like numbers and underscores. Following convention, we capitalize the first letter of a class name, and lowercase the first letter of all **val**s and **var**s.

Lines 4-6 define three new classes, and lines 9-12 create objects (also known as *instances*) of those classes using **new**. The **new** keyword creates a new object, given a class.

Giraffe is a class, but a particular five-year-old male giraffe that lives in Arizona is an *object*. Every time you create a new object, it's different from all the others, so we give them names like **g1** and **g2.** You see their uniqueness in the rather cryptic output of lines 15-18, which looks something like:

```
Main$$anon$1$Giraffe@53f64158
Main$$anon$1$Giraffe@4c3c2378
Main$$anon$1$Hippo@3cc262
Main$$anon$1$Bear@14fdb00d
```

If we remove the common **Main$$anon$1$** part, we see:

```
Giraffe@53f64158
Giraffe@4c3c2378
Hippo@3cc262
Bear@14fdb00d
```

The part before the '@' is the class name, and the number (yes, that's a number even though it includes some letters – it's called "hexadecimal notation" and you can learn about it in Wikipedia) indicates the address where the object is located in your computer's memory.

The classes defined here (**Giraffe**, **Bear**, and **Hippo**) are as simple as they can be; the entire class definition is a single line. More complex classes use curly braces '{' and '}' to describe the characteristics and behaviors for that class. This can be as trivial as telling us an object is being created:

```scala
// Hyena.scala

class Hyena {
  println("This is in the class body")
}
val hyena = new Hyena
```

The code inside the curly braces is the *class body*, and is executed when an object is created.

Exercises

Solutions are available at **www.AtomicScala.com**.

1. Create classes for **Hippo**, **Lion**, **Tiger**, **Monkey**, and **Giraffe**, then create an instance of each one of those classes. Display the objects. Do you see five different ugly-looking (but unique) strings? Count and inspect them.

2. Create a second instance of **Lion** and two more **Giraffes**. Print those objects. How do they differ from the original objects that you created?

3. Create a class **Zebra** that prints "I have stripes" when you create it. Test it.

Methods Inside Classes

When you define methods inside a class, the method belongs to that class. Here, the **bark** method belongs to the **Dog** class:

```
// Dog.scala
class Dog {
  def bark():String = { "yip!" }
}
```

Methods are called (invoked) with the object name, followed by a '.' (dot/period), followed by the method name. Here, we call the **meow** method on line 7, and we use **assert** to validate the result:

```
1  // Cat.scala
2  class Cat {
3    def meow():String = { "mew!" }
4  }
5
6  val cat = new Cat
7  val m1 = cat.meow()
8  assert("mew!" == m1,
9    "Expected mew!, Got " + m1)
```

Methods have special access to the other elements within a class. For example, you can call another method within the class without using a dot (that is, without *qualifying* it). Here, the **exercise** method calls the **speak** method without qualification:

```
// Hamster.scala
class Hamster {
  def speak():String = { "squeak!" }
  def exercise():String = {
    speak() + " Running on wheel"
6     }
7   }
8
9   val hamster = new Hamster
10  val e1 = hamster.exercise()
11  assert(
12    "squeak! Running on wheel" == e1,
13    "Expected squeak! Running on wheel" +
14    ", Got " + e1)
```

Outside the class, you must say **hamster.exercise** (as on line 10) and **hamster.speak**.

The methods that we created in Methods didn't appear to be inside a class definition, but it turns out that everything in Scala is an object. When we use the REPL or run a script, Scala takes any methods that aren't inside classes and invisibly bundles them inside of an object.

Exercises

Solutions are available at **www.AtomicScala.com**.

1. Create a **Sailboat** class with methods to raise and lower the sails, printing "Sails raised," and "Sails lowered," respectively. Create a **Motorboat** class with methods to start and stop the motor, returning "Motor on," and "Motor off," respectively. Make an object (instance) of the **Sailboat** class. Use **assert** for verification:

```
val sailboat = new Sailboat
val r1 = sailboat.raise()
assert(r1 == "Sails raised",
  "Expected Sails raised, Got " + r1)
val r2 = sailboat.lower()
assert(r2 == "Sails lowered",
  "Expected Sails lowered, Got " + r2)
val motorboat = new Motorboat
val s1 = motorboat.on()
```

```
assert(s1 == "Motor on",
  "Expected Motor on, Got " + s1)
val s2 = motorboat.off()
assert(s2 == "Motor off",
  "Expected Motor off, Got " + s2)
```

2. Create a new class **Flare**. Define a **light** method in the **Flare** class. Satisfy the following:

```
val flare = new Flare
val f1 = flare.light
assert(f1 == "Flare used!",
  "Expected Flare used!, Got " + f1)
```

3. In each of the **Sailboat** and **Motorboat** classes, add a method **signal** that creates a **Flare** object and calls the **light** method on the **Flare**. Satisfy the following:

```
val sailboat2 = new Sailboat2
val signal = sailboat2.signal()
assert(signal == "Flare used!",
  "Expected Flare used! Got " + signal)
val motorboat2 = new Motorboat2
val flare2 = motorboat2.signal()
assert(flare2 == "Flare used!",
  "Expected Flare used!, Got " + flare2)
```

Imports & Packages

One of the most fundamental principles in programming is the acronym DRY: *Don't Repeat Yourself*. Whenever you duplicate code, you are not just doing extra work. You're also creating multiple identical pieces of code that you must change whenever you need to fix or improve it, and every duplication is a place to make another mistake.

Scala's **import** reuses code from other files. One way to use **import** is to specify the class name you want to use:

```
import packagename.classname
```

A package is an associated collection of code; each package is usually designed to solve a particular kind of problem, and often contains multiple classes. For example, the **util** package includes **Random**, which generates a random number:

```
1   // ImportClass.scala
2   import util.Random
3
4   val r = new Random
5   println(r.nextInt(10))
6   println(r.nextInt(10))
7   println(r.nextInt(10))
```

After creating a **Random** object, lines 5-7 use **nextInt** to generate random numbers between 0 and 10, not including 10.

The **util** package contains other classes and objects as well, such as the **Properties** object. We can import more than one class using multiple **import** statements:

```
// ImportMultiple.scala
import util.Random
import util.Properties

val r = new Random
val p = Properties
```

You can import more than one item within the same **import** statement:

```
// ImportSameLine.scala
import util.Random, util.Properties

val r = new Random
val p = Properties
```

However, Scala has syntax for combining multiple classes in a single **import** statement:

```
// ImportCombined.scala
import util.{Random, Properties}

val r = new Random
val p = Properties
```

You can even change the name as you import:

```
// ImportNameChange.scala
import util.{ Random => Bob,
  Properties => Jill }

val r = new Bob
val p = Jill
```

If you want to import everything in a package, use the underscore:

```scala
// ImportEverything.scala
import util._

val r = new Random
val p = Properties
```

Finally, if you only use something in a single place, you may choose to skip the **import** statement and fully qualify the name:

```scala
// FullyQualify.scala

val r = new util.Random
val p = util.Properties
```

So far in this book we've been able to use simple scripts for our examples, but eventually you'll need to write some code for use in multiple places. Rather than duplicating the code, Scala allows you to create and import packages. You create your own packages using the **package** keyword (which must be the first non-comment statement in the file) followed by the name of your package (package names can be more complex than this):

```scala
// TheRoyalty.scala
package royals

class Royalty(name:String,
characteristic:String) {
  def title():String = {
    "Sir " + characteristic + "alot"
  }
  def fancyTitle():String = {
    "Sir " + name +
    " " + characteristic + "alot"
  }
}
```

On line 2, we name the package **royals**, and then define the class **Royalty** in the usual way. Notice there's no requirement to name the source-code file anything special.

To make the package accessible to a script, we must *compile* the package using the **scalac** command at the shell prompt:

```
scalac TheRoyalty.scala
```

Packages cannot be scripts – they can only be compiled.

Once **scalac** is finished, you will discover that there's a new directory with the same name as the package; here the directory name is **royalty**. This directory contains a file for each class defined in the **royalty** package, each with **.class** at the end of the file name.

Now the elements in the **royalty** package are available to any script in our directory by using an **import**:

```
// ImportRoyalty.scala
import royals.Royalty

val royal = new Royalty("Henry", "Laughs")
val title = royal.title()
assert("Sir Laughsalot" == title,
  "Expected Sir Laughsalot, Got " + title)
```

You can run the script as usual with:

```
scala ImportRoyalty.scala
```

Note: there is a bug in Scala 2.10 and below that causes a delay between compiling and making the classes available for import. To get around this bug, use the **nocompdaemon** flag:

```
scala -nocompdaemon ImportRoyalty.scala
```

Package names should be unique, and the Scala community has a convention of using the reversed-domain package name of the creator to ensure this. Since our domain name is **Atomicscala.com**, for our package to be part of a distributed library it should be named **com.atomicscala.royals** rather than just **royals**. This helps us avoid name collisions with other libraries that might also use the name **royals**.

Exercises

Solutions are available at **www.AtomicScala.com**.

1. Rename the package **royals** using the reverse domain-name standard described above. Build it with **scalac**, following the previously described steps, and ensure that a directory hierarchy is created on your computer to hold these classes. Revise **ImportRoyalty.scala,** above, and save as **ImportTests.scala**. Remember to update the package import to use your new class. Ensure that the tests run properly.

2. Add another class **Crest** to your solution for Exercise 1. Pass in a **name** (as a **String**) and a **year** (as a **String**) for the crest. Create a method **description** that returns “Bear in the year 1875,” when “Bear” is the name and “1875” is the year.

3. Modify **ImportRoyalty.scala** to use the various different importing methods shown in this atom.

4. Create your own package containing three trivial classes (just define the classes, don’t give them bodies). Use the techniques in this atom to import one class, two classes, and all classes, and show that you’ve successfully imported them in each case.

Testing

For robust code, you must test it constantly – every time you make changes. This way, if you make a change in one part of your code that unexpectedly has an effect somewhere else, you know immediately, as soon as you make the change, and you know which change caused things to break. If you don't find out immediately, then changes accumulate and you don't know which one caused the problem – which means you'll spend a *lot* longer tracking it down. Constant testing is therefore essential for rapid program development.

Because testing is a crucial practice, we introduce it early and use it throughout the rest of the book. This way, you'll become accustomed to testing as a standard part of the programming process.

Using **println** to verify code correctness is a rather weak approach; it requires us to pay attention to the output every time and consciously assure that it's right. Using **assert** is better because it happens automatically. However, a failed **assert** produces very noisy output that's often less than clear. In addition, we'd like a more natural syntax for writing tests.

To simplify your experience using this book, we created our own tiny testing system. The goal is a minimal approach that:

- Allows you to see the expected result of expressions right next to those expressions, for easier comprehension.
- Shows some output so you can see the program is running, even when all the tests succeed.
- Ingrains the concept of testing early in your practice.
- Requires no extra downloads or installations to work.

Although useful, this is *not* a testing system for use in the workplace. Others have worked long and hard to create such test systems – in particular, Bill Venners' *ScalaTest* (www.scalatest.org) has become the de facto standard for Scala testing, and you should reach for that when you start producing real Scala code.

This example shows our testing framework, imported on line 2:

```
1   // TestingExample.scala
2   import com.atomicscala.AtomicTest._
3
4   val v1 = 11
5   val v2 = "a String"
6
7   // "Natural" syntax for test expressions:
8   v1 is 11
9   v2 is "a String"
10  v2 is "Produces Error" // Show failure
11  /* Output:
12  11
13  a String
14  a String
15  [Error] expected:
16  Produces Error
17  */
```

Before you can run a Scala script that uses **AtomicTest**, you must follow the instructions in your appropriate "Installation" atom to compile the **AtomicTest** object (or just run the "testall" script, also described in that atom).

We don't intend that you understand the code for **com.atomicscala.AtomicTest** because it uses some tricks that are beyond the scope of this book. If you'd like to see the code, it's in Appendix A.

In order to produce a clean, comfortable appearance, **AtomicTest** uses a Scala feature that you haven't seen before: the ability to write a method call **a.method(b)** in the text-like form:

```
a method b
```

This is called *infix notation*. **AtomicTest** uses this feature by defining an **is** method:

```
expression is expected
```

You can see this used on lines 8 and 9 in the previous example.

This system is very flexible – almost anything works as a test expression. If *expected* is a string, then *expression* is converted to a string and the two strings are compared. Otherwise, *expression* and *expected* are just compared directly (without converting them first). In either case, *expression* is displayed on the console so you can see something happening when the program runs. If *expression* and *expected* are not equivalent, **AtomicTest** prints an error message when the program runs (and records it in the file **_AtomicTestErrors.txt**).

Lines 12-14 show the output; the output from lines 8 and 9 are on lines 12 and 13; even though the tests succeeded you still get output showing the contents of the object on the left of **is**. Line 10 intentionally fails so you can see the output. Line 14 shows what the object is, followed by the error message, followed by what the program expected to see for that object.

That's all there is to it. The **is** method is the only operation defined for **AtomicTest** – it truly is a minimal testing system. Now you can put "**is**" expressions anywhere in a script to produce both a test and some console output.

From now on we won't need commented output blocks because the testing code will do everything we need (and better, because you can see the results right there rather than scrolling to the bottom and detecting which line of output corresponds to a particular **println**).

Anytime you run a program that uses **AtomicTest**, you'll automatically verify the correctness of that program. Ideally, by seeing the benefits of using testing throughout the rest of the book, you'll become addicted to the idea of testing and will feel uncomfortable when you see code that doesn't have tests. You will probably start feeling that code without tests is broken by definition.

Testing as Part of Programming

There's another benefit to writing testably – it changes the way you think about and design your code. In the above example, we could have just displayed the results to the console. But the test mindset makes you think, "How will I test this?" When you create a method, you begin thinking that you should return something from the method, if for no other reason than to test that result. It turns out that methods that take one thing and transform it into something else tend to produce better designs, as well.

Testing is most effective when it's built into your software development process. Writing tests ensures that you're getting the results you expect. Many people advocate writing tests before writing the implementation code – to be rigorous, you first make the test fail before you write the code to make it pass. This technique, called *Test Driven Development* (TDD), is a way to make sure that you're really testing what you think you are. You can find a more complete description of TDD on Wikipedia (search for "Test_driven_development").

Here’s a simplified example using TDD to implement the BMI calculation from Evaluation Order. First, we write the tests, along with an initial implementation that fails (because we haven’t yet implemented the functionality).

```scala
// TDDFail.scala
import com.atomicscala.AtomicTest._

calculateBMI(160, 68) is "Normal weight"
calculateBMI(100, 68) is "Underweight"
calculateBMI(200, 68) is "Overweight"

def calculateBMI(lbs: Int,
  height: Int):String = { "Normal weight" }
```

Only the first test passes. Next we add code to determine which weights are in which categories:

```scala
// TDDStillFails.scala
import com.atomicscala.AtomicTest._

calculateBMI(160, 68) is "Normal weight"
calculateBMI(100, 68) is "Underweight"
calculateBMI(200, 68) is "Overweight"

def calculateBMI(lbs:Int,
  height:Int):String = {
  val bmi = lbs / (height*height) * 703.07
  if (bmi < 18.5) "Underweight"
  else if (bmi < 25) "Normal weight"
  else "Overweight"
}
```

Now *all* the tests fail because we’re using **Int**s instead of **Double**s, producing a zero result. The tests guide us to the fix:

```
// TDDWorks.scala
import com.atomicscala.AtomicTest._

calculateBMI(160, 68) is "Normal weight"
calculateBMI(100, 68) is "Underweight"
calculateBMI(200, 68) is "Overweight"

def calculateBMI(lbs:Double,
  height:Double):String = {
  val bmi = lbs / (height*height) * 703.07
  if (bmi < 18.5) "Underweight"
  else if (bmi < 25) "Normal weight"
  else "Overweight"
}
```

You may choose to add additional tests to ensure that we have tested the boundary conditions completely.

Wherever possible in the remaining exercises of this book, we include tests your code must pass. Feel free to test additional cases.

Exercises

Solutions are available at **www.AtomicScala.com**.

1. Create a value named **myValue1** initialized to 20. Create a value named **myValue2** initialized to 10. Use “is” to test that they do not match.

2. Create a value named **myValue3** initialized to 10. Create a value named **myValue4** initialized to 10. Use “is” to test that they do match.

3. Compare **myValue2** and **myValue3**. Do they match?

4. Create a value named **myValue5** initialized to the **String** “10”. Compare it to **myValue2**. Does it match?

5. Use Test Driven Development (write a failing test, and then write the code to fix it) to calculate the area of a quadrangle. Start with the following sample code and fix the intentional bugs:

```
def squareArea(x: Int):Int = x * x
def rectangleArea(x:Int, y:Int):Int = x * x
def trapezoidArea(x:Int, y:Int,
  h:Int):Double = h/2 * (x + y)

squareArea(1) is 1
squareArea(2) is 4
squareArea(5) is 25
rectangleArea(2, 2) is 4
rectangleArea(5, 4) is 20
trapezoidArea(2, 2, 4) is 8
trapezoidArea(3, 4, 1) is 3.5
```

End of Sample

We hope you've enjoyed this sample of *Atomic Scala*. If you've found it useful, you can purchase the rest of the book either in print form or as an eBook in HTML, PDF, mobi (for Kindle) and ePub (for Apple iBooks, Nook, Kobo, etc.) from:

www.AtomicScala.com

Copyright

Created in the United States in Crested Butte, Colorado and Ann Arbor, Michigan.

ISBN 978-0-9818725-1-3
Text printed in the United States
First printing (Version 1.0), March 2013
Version 1.1, September 2013

Front and back cover illustrations by John Lucas.
Cover and interior design by Daniel Will-Harris, www.will-harris.com

Visit us at AtomicScala.com