Atomic Scala

Bruce Eckel

Dianne Marsh

Mindview LLC, Crested Butte, CO

Contents

How to Use This Book

This free sample from TypeSafe.com only contains the first half of the book. To purchase the full book, please go to www.AtomicScala.com.

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 free sample from TypeSafe.com only contains the first half of the book. To purchase the full book, please go to www.AtomicScala.com.

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

All copyrights in this book are the property of their respective holders. See Copyright for full details.

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:

```
// 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:

```
// CompoundExpressions2.scala

val activity = "swimming"
val hour = 10

val isOpen = {
  if(activity == "swimming" ||
     activity == "ice skating") {
    val opens = 9
    val closes = 20
    println("Operating hours: " +
      opens + " - " + closes)
    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:

```
// 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:

```
// 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:

```
// 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):

```
// 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).

```
// 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:

```
// 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
```

Fields

A *field* is a **var** or **val** that's part of an object. Each object gets its own storage for fields:

```
1   // Cup.scala
2   import com.atomicscala.AtomicTest._
3
4   class Cup {
5     var percentFull = 0
6   }
7
8   val c1 = new Cup
9   c1.percentFull = 50
10  val c2 = new Cup
11  c2.percentFull = 100
12  c1.percentFull is 50
13  c2.percentFull is 100
```

Defining a **var** or **val** inside a class looks just like defining it outside the class. However, the **var** or **val** becomes part of that class, and to refer to it, you must specify its object to using the dot notation as on lines 9 and 11-13.

Note that **c1** and **c2** have different values in their **percentFull** vars, which shows that each object has its own piece of storage for **percentFull**.

A method can refer to a field within its object without using a dot (that is, without qualifying it):

```
1  // Cup2.scala
2  import com.atomicscala.AtomicTest._
3
4  class Cup2 {
5    var percentFull = 0
6    val max = 100
7    def add(increase:Int):Int = {
8      percentFull += increase
9      if(percentFull > max) {
10       percentFull = max
11     }
12     percentFull // Return this value
13   }
14 }
15
16 val cup = new Cup2
17 cup.add(50) is 50
18 cup.add(70) is 100
```

The '**+=**' operator on line 8 adds **increase** to **percentFull** and assigns the result to **percentFull** in a single operation. It is equivalent to saying:

```
percentFull = percentFull + increase
```

The **add** method tries to add **increase** to **percentFull** but ensures that it doesn't go past 100%. The method **add**, like the field **percentFull**, is defined inside the class **Cup2**. To refer to either of them from outside the class, as on line 17, you use the dot between the object and the name of the field or method.

Exercises

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

1. What happens in **Cup2**'s **add** method if **increase** is a negative value? Is any additional code necessary to satisfy the following tests:
```
val cup2 = new Cup2
cup2.add(45) is 45
cup2.add(-15) is 30
cup2.add(-50) is -20
```

2. Adding to your solution for Exercise 1, add code to handle negative values to ensure that the total never goes below 0. Satisfy the following tests:
```
val cup3 = new Cup3
cup3.add(45) is 45
cup3.add(-55) is 0
cup3.add(10) is 10
cup3.add(-9) is 1
cup3.add(-2) is 0
```

3. Can you set **percentFull** from outside the class? Try it, like this:
```
cup3.percentFull = 56
cup3.percentFull is 56
```

4. Write methods that allow you to both set and get the value of **percentFull**. Satisfy the following:
```
val cup4 = new Cup4
cup4.set(56)
cup4.get() is 56
```

For Loops

A **for** loop steps through a sequence of values so you can perform operations using each value. You start with the keyword **for**, followed by a parenthesized expression that traverses the sequence of values. Within the parentheses, you first see the identifier that receives each of the values in turn, pointed at by a **<-** (backwards-pointing arrow; you may choose to read this as "gets"), and then an expression that generates the sequence. On lines 5, 11 and 17, we show three equivalent expressions: **0 to 9**, **0 until 10** and **Range(0, 10)** (**to** and **until** are additional examples of *infix notation*). Each produces a sequence of **Int**s, which we append to a **var String** called **result** (using the '**+=**' operator), in order to produce something testable (then we reset **result** to an empty string for the next **for** loop):

```
1   // For.scala
2   import com.atomicscala.AtomicTest._
3
4   var result = ""
5   for(i <- 0 to 9) {
6     result += i + " "
7   }
8   result is "0 1 2 3 4 5 6 7 8 9 "
9
10  result = ""
11  for(i <- 0 until 10) {
12    result += i + " "
13  }
14  result is "0 1 2 3 4 5 6 7 8 9 "
15
16  result = ""
17  for(i <- Range(0, 10)) {
18    result += i + " "
19  }
20  result is "0 1 2 3 4 5 6 7 8 9 "
```

```
21
22  result = ""
23  for(i <- Range(0, 20, 2)) {
24    result += i + " "
25  }
26  result is "0 2 4 6 8 10 12 14 16 18 "
27
28  var sum = 0
29  for(i <- Range(0, 20, 2)) {
30    println("adding " + i + " to " + sum)
31    sum += i
32  }
33  sum is 90
```

On lines 5 and 11, we use **for** loops to generate all of the values, demonstrating both **to** and **until**. Specifying a **Range** with start and end points is more obvious. On line 17, **Range** creates a list of values from 0 up to, but not including 10. If you want to include the endpoint (10), you can use:

```
Range(0, 10).inclusive
```

or

```
Range(0, 11)
```

The first form seems to make the meaning more explicit.

Note the use of type inference for **i** in the various **for** loops.

The expression following the **for** loop is called the *body*. The body is executed for each value of **i**. The body, like any other expression, can contain just one line of code (lines 6, 12, 18 and 24) or more than one line of code (lines 30-31).

Line 23 also uses a **Range** to print a series of values, but the third argument (**2**) steps the sequence by a value of two instead of one (try different step values).

On line 28, we declare **sum** as a **var** instead of a **val**, so we can modify **sum** each time through the loop.

There are more concise ways to write **for** loops in Scala, but we'll start with this form because it's often easier to read.

Exercises

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

1. Create a value of type **Range** that goes from 0 to 10 (not including 10). Satisfy the following tests:
   ```
   val r1 = // fill this in
   r1 is // fill this in
   ```

2. Use **Range.inclusive** to solve the problem above. What changed?

3. Write a **for** loop that adds the values 0 through 10 (including 10). Sum all the values and ensure that it equals 55. Must you use a **var** instead of a **val**? Why? Satisfy the following test:
   ```
   total is 55
   ```

4. Write a **for** loop that adds even numbers between 1 and 10 (including 10). Sum all the values and ensure that it equals 30. Hint: this conditional expression determines whether a number is even:
   ```
   if (number % 2 == 0)
   ```
   The % (modulo) operator checks to see if there is a remainder when you divide **number** by 2. Satisfy the following:
   ```
   totalEvens is 30
   ```

5. Write a **for** loop that adds even numbers between 1 and 10 (including 10) and odd numbers between 1 and 10. Calculate a sum for the even numbers and a sum for the odd numbers. Did you write two **for** loops? If so, try rewriting this with a single **for** loop. Satisfy the following tests:
   ```
   evens is 30
   odds is 25
   (evens + odds) is 55
   ```

6. If you didn't use **Range** for Exercise 5, rewrite using **Range**. If you did use **Range**, rewrite the **for** using **to** or **until**.

Vectors

A **Vector** is a *container* – something that holds other objects. Containers are also called *collections*. **Vector**s are part of the standard Scala package so they're available without any imports. On line 4 in the following example, we create a **Vector** populated with **Int**s by simply stating the **Vector** name and handing it the initialization values:

```
1   // Vectors.scala
2   import com.atomicscala.AtomicTest._
3
4   // A Vector holds other objects:
5   val v1 = Vector(1, 3, 5, 7, 11, 13)
6   v1 is Vector(1, 3, 5, 7, 11, 13)
7
8   v1(4) is 11 // "Indexing" into a Vector
9
10  // Take each element of the Vector:
11  var result = ""
12  for(i <- v1) {
13    result += i + " "
14  }
15  result is "1 3 5 7 11 13 "
16
17  val v3 = Vector(1.1, 2.2, 3.3, 4.4)
18  // reverse is an operation on the Vector:
19  v3.reverse is Vector(4.4, 3.3, 2.2, 1.1)
20
21  var v4 = Vector("Twas", "Brillig", "And",
22                  "Slithy", "Toves")
23  v4 is Vector("Twas", "Brillig", "And",
24          "Slithy", "Toves")
25  v4.sorted is Vector("And", "Brillig",
26          "Slithy", "Toves", "Twas")
```

```
27  v4.head is "Twas"
28  v4.tail is Vector("Brillig", "And",
29          "Slithy", "Toves")
```

Line 6 shows that when you display a **Vector**, it produces the output in the same form as the initialization expression, making it easy to understand.

On line 8, parentheses are used to *index* into the **Vector**. A **Vector** keeps its elements in the order they were initialized, and you can select them individually by number. Most programming languages start indexing at element zero, which in this case produces the value **1**. Thus, the index of **4** produces a value of **11**.

Forgetting that indexing starts at zero is responsible for the so-called *off-by-one* error. If you try to use an index beyond the last element in the **Vector**, Scala will throw one of the exceptions we talked about in Methods. The exception will display an error message telling you it's an *IndexOutOfBoundsException* so you can figure out what the problem is. Try adding the following, any time after line 22:

```
println(v4(5))
```

In a language like Scala we often don't select elements one at a time, but instead *iterate* through a whole container – an approach that eliminates off-by-one errors. On line 12, you can see that **for** loops work very well with **Vector**s: **for(i <- v1)** means "**i** gets each value in **v1**." This is another example of Scala helping you: you don't even declare **val i** or give its type; Scala knows from the context that this is a **for** loop variable, and takes care of that for you. Many other programming languages will force you to do the extra work – this can be annoying because, in the back of your mind, you know that the language *can* figure it out and it seems like it's making you do the extra work out of spite. For this and many other reasons, programmers from other languages find Scala to be a breath of fresh air – it seems to be saying, "How can I serve you?" instead of cracking a whip and forcing you to jump through hoops.

A **Vector** can hold all different types; on line 17 we create a **Vector** of **Double**. On line 19 this is displayed in reverse order.

The rest of the program experiments with a few other operations. Note the use of the word **sorted** instead of just "sort." When you call **sorted** it *produces* a new **Vector** containing the same elements as the old, in sorted order – but it leaves the original **Vector** alone. If they had instead chosen to say "sort," it would imply that the original **Vector** was changed directly (a.k.a. *sorted in place*). Throughout Scala you will see this tendency of "leaving the original thing alone and producing a new thing." For example, the **head** operation produces the first element of the **Vector** but leaves the original alone, and the **tail** operation produces a new **Vector** containing all but the first elements – and leaves the original alone.

You can learn more about **Vector** by looking it up in ScalaDoc.

Note: Since we speak highly of Scala's consistency we also wanted to point out that it's not perfect. The **reverse** method on line 19 produces a new **Vector**, ordered end to beginning. To maintain consistency with **sorted**, that name should have been "reversed."

Exercises

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

1. Use the REPL to create several **Vector**s, each populated by a different type of data. Look at how the REPL responds and guess what it means.

2. Use the REPL to see if you can make a **Vector** containing other **Vector**s. How might you use such a thing?

3. Create a **Vector** and populate it with words (which are **String**s). Add a **for** loop that prints each element in the **Vector**. Building on the previous exercise, append to a **var String** to create a sentence. Satisfy the following test:
   ```
   sentence.toString() is
     "The dog visited the firehouse "
   ```

4. That last space is unexpected. Use **String**'s method **replace** to replace "firehouse " with "firehouse!" Satisfy the following test:
   ```
   theString is
     "The dog visited the firehouse!"
   ```

5. Building from your solution for Exercise 4, write a **for** loop that prints each word, reversed. Your output should match:
   ```
   /* Output:
   ehT
   god
   detisiv
   eht
   esuoherif
   */
   ```

6. Write a **for** loop that prints the words from Exercise 4 in reverse order (last word first, etc.). Your output should match:
   ```
   /* Output:
   firehouse
   the
   visited
   dog
   The
   */
   ```

7. Create and initialize two **Vector**s, one containing **Int**s and one containing **Double**s. Call the **sum**, **min** and **max** operations on each one and see what happens.

8. Create a **Vector** containing **String**s and apply the **sum**, **min** and **max** operations. Explain the results. One of those methods won't work. Why?

9. In For Loops, we added the values in a **Range** to get the sum. Try calling the sum operation on a **Range**. Does this do the entire summation in one step?

10. **List** and **Set** are similar to **Vector**. Use the REPL to discover their operations and compare them to those of **Vector**.

11. Create and initialize a **List** and **Set** with words, then print each one. Try the **reverse** and **sorted** operations and see what happens.

12. Create two **Vector**s of **Int** named **myVector1** and **myVector2**, each initialized to 1, 2, 3, 4, 5, 6. Use **AtomicTest** to show whether they are equivalent. Is this what you expected?

More Conditionals

Let's practice creating methods by writing some that take Boolean arguments (You learned about Booleans in Conditional Expressions):

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

def trueOrFalse(exp:Boolean):String = {
  if(exp) {
    return "It's true!" // Need 'return'
  }
  "It's false"
}

val b = 1
trueOrFalse(b < 3) is "It's true!"
trueOrFalse(b > 3) is "It's false"
```

A Boolean argument **exp** is passed to the method **trueOrFalse**. If the argument is passed as an expression, such as **b < 3**, that expression is first evaluated and the result is then passed to the method. In this case, **exp** is tested and if it is **true**, the lines within the curly braces are executed.

The **return** keyword is new here. It says, "Leave this method and return this value." Normally, the last expression in a Scala method produces the value returned from that method, so we don't usually need the **return** keyword and you won't see it very often. If we just gave the **String** "It's true" without the **return**, nothing would happen; the method would continue and always return "It's false" (Try it – remove the **return** and see what happens).

It's more common to use the **else** keyword:

```
 1  // OneOrTheOther.scala
 2  import com.atomicscala.AtomicTest._
 3
 4  def oneOrTheOther(exp:Boolean):String = {
 5    if(exp) {
 6      "True!" // No 'return' necessary
 7    }
 8    else {
 9      "It's false"
10    }
11  }
12
13  val v = Vector(1)
14  val v2 = Vector(3, 4)
15  oneOrTheOther(v == v.reverse) is "True!"
16  oneOrTheOther(v2 == v2.reverse) is
17  "It's false"
```

The **oneOrTheOther** method is now a single expression, instead of the two expressions inside **trueOrFalse**. The result of that expression – the expression on line 6 if **exp** is true, or line 9 if **exp** is false – becomes the returned value, so the **return** keyword is no longer necessary.

Some people feel strongly that **return** should never be used to exit a method in the middle, but we remain neutral on the subject.

The tests show that if a **Vector** of length one is reversed it is always equal to the original, but if it is longer than one the reverse typically *isn't* equal to the original.

You are not limited to a single test. You can test multiple combinations by combining **else** and **if**:

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

```

def checkTruth(
  exp1:Boolean, exp2:Boolean):String = {
  if(exp1 && exp2) {
    "Both are true"
  }
  else if(!exp1 && !exp2) {
    "Both are false"
  }
  else if(exp1) {
    "First: true, second: false"
  }
  else {
    "First: false, second: true"
  }
}

checkTruth(true || false, true) is
  "Both are true"
checkTruth(1 > 0 && -1 < 0, 1 == 2) is
  "First: true, second: false"
checkTruth(1 >= 2, 1 >= 1) is
  "First: false, second: true"
checkTruth(true && false,false && true) is
"Both are false"
```

The typical pattern is to start with an **if**, followed by as many **else if** clauses as you need, and ending with a final **else** for anything that doesn't match all the previous tests. When an **if** expression reaches a certain size and complexity you'll probably want to use pattern matching, described after Summary 2.

Exercises

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

1. Under what conditions does a **Vector** of length greater than one equal its reverse?

2. Palindromes are words or phrases that read the same forward and backward. Some examples include "mom" and "dad." Write a method to test words or phrases for palindromes. Hint: **String**'s **reverse** method may prove useful here. Use **AtomicTest** to check your solution (*remember to import it!*). Satisfy the following tests:
   ```
   isPalindrome("mom") is true
   isPalindrome("dad") is true
   isPalindrome("street") is false
   ```

3. Building on the previous exercise, ignore case when testing for palindromes. Satisfy the following tests:
   ```
   isPalIgnoreCase("Bob") is true
   isPalIgnoreCase("DAD") is true
   isPalIgnoreCase("Blob") is false
   ```

4. Building on the previous exercise, strip out special characters before palindrome testing. Here's some sample code and tests: (Hint: In integer values, 'A' is 65, 'B' is 66, … 'a' is 97 … 'z' is 122. '0' is 48 … '9' is 57)
   ```
   var createdStr = ""
   for (c <- str) {
     // convert to Int for comparison:
     val theValue = c.toInt
     if (/* check for letters */) {
       createdStr += c
     }
     else if (/* check for numbers */) {
       createdStr += c
     }
   }
   isPalIgnoreSpecial("Madam I'm adam") is
   true
   isPalIgnoreSpecial("trees") is false
   ```

Summary 2

This atom summarizes and reviews the atoms from Methods through More Conditionals. If you're an experienced programmer, this should be your next atom after Summary 1. Beginning programmers should read the 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.

The topics appear in appropriate order for experienced programmers, which is not the same as the order of the atoms in the book. For example, we'll start by introducing packages and imports so that we can use our minimal test framework for the rest of the atom.

Packages, Imports & Testing

Any number of reusable library components can be bundled under a single library name using the **package** keyword:

```
// ALibrary.scala
package com.yoururl.libraryname
// Components to reuse ...
class X
```

You can put multiple components in a single file, or spread components out among multiple files under the same package name. Here we've defined an empty **class** called **X** as the sole component.

You must compile libraries using the **scalac** command:

```
scalac ALibrary.scala
```

The package name conventionally begins with your reversed domain name in order to make it unique. On line 2, the domain name is **yoururl.com**. If the package name contains periods, each part of the name becomes a subdirectory. So when you compile **ALibrary.scala**, you'll produce the directory structure (beneath the current directory):

```
com/yoururl/libraryname
```

The **libraryname** directory will contain a compiled file with a name ending in **.class** for each component in your library.

The **import** statement allows you to use a library:

```
// UseALibrary.scala
import com.yoururl.libraryname._
new X
```

The underscore after the library name tells Scala to bring in all the components of a library. Now we can refer to **X** without producing an error. You can also select components individually; details are in Imports and Packages.

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 ALibrary.scala
```

An important library in this book is **AtomicTest**, our very simple testing framework. Once it's imported, you can use "**is**" almost as if it were a language keyword:

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

```

```
val pi = 3.14
val pie = "A round dessert"

pi is 3.14
pie is "A round dessert"
pie is "Square" // Produces error
```

The ability to use **is** without any dots or parentheses is called *infix notation*, a fundamental language feature. **AtomicTest** makes **is** an assertion of truth which also prints the result on the left side of the **is** statement, and an error message if the expression on the right of the **is** doesn't agree. This way you see verified results in the source code.

AtomicTest is defined in Appendix A; you must compile it with the command line **scalac AtomicTest.scala** before the above code will work.

Methods

Almost all named subroutines in Scala are created as *methods*. The basic form is:

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

The **def** keyword is followed by the method name and the argument list in parentheses. Each argument must have a type (Scala cannot infer argument types). The method itself has a type, which is defined in the same way as a type for a **var** or **val**: a colon followed by the type name. The method's type is the type of the result that is returned by that method.

The method signature is followed by an "**=**" and the method body, which is effectively just an expression; this is typically a compound expression surrounded by curly braces as you see above. The result of the body – the last line of the compound expression above – becomes the return value of the method.

Here's a method that produces the cube of its argument, and another one that adds an exclamation point to a **String**:

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

def cube(x:Int):Int = { x * x * x }
cube(3) is 27

def bang(s:String):String = { s + "!" }
bang("pop") is "pop!"
```

In each case, the method body is a single expression that produces the method's return value.

Classes & Objects

Scala is a *hybrid object-functional* language: it supports both object-oriented and functional programming paradigms.

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. When you create a **val** or **var** of a class, it's called *creating an object* or sometimes *creating an instance of an object*. Even instances of what would be built-in types in other languages (like **Double** or **String**) are objects in Scala.

An especially useful type of object is the *container* or *collection*: an object that holds other objects. In this book, we primarily use the **Vector** because it's the most general-purpose sequence. Here we create a **Vector** holding **Double**s and perform several operations on it:

```
1   // VectorCollection.scala
2   import com.atomicscala.AtomicTest._
3
4   val v1 = Vector(19.2, 88.3, 22.1)
5   v1 is Vector(19.2, 88.3, 22.1)
6   v1(1) is 88.3 // Indexing
7   v1.reverse is Vector(22.1, 88.3, 19.2)
8   v1.sorted is Vector(19.2, 22.1, 88.3)
9   v1.max is 88.3
10  v1.min is 19.2
```

No **import** statement is required to use a **Vector**. On line 6 you can see that Scala uses parentheses for indexing into sequences (indexing is zero-based) rather than square brackets as many languages do.

Lines 7 - 10 show examples of the numerous methods available for Scala collections. The REPL is useful as an investigation tool here; create a **Vector** object:

```
scala> val v = Vector(1)
```

Now type **v.** (**v** followed by a period) as if you're about to call a method for **v**, but instead, press the TAB key. This works for any type of object; the REPL produces a list of possible methods to call.

To find out what all those methods mean, use the Scala documentation, which is available as a download or online. See the ScalaDoc atom for details.

When you call **reverse** and **sorted** as on lines 7 and 8, the **Vector v1** is not modified. Instead, a new **Vector** is created and returned, containing the desired result. This approach of never modifying the original object is consistent throughout Scala libraries and you should endeavor to follow this pattern whenever you can.

Creating Classes

A class definition consists of the **class** keyword, a name for the class, and an optional body. The body can contain:

1. Field definitions (**val**s and **var**s)
2. Method definitions
3. Code that is executed during the creation of each object

This example shows both fields and initialization code:

```
1  // ClassBodies.scala
2
3  class NoBody
4  val nb = new NoBody
5
6  class SomeBody {
7    val name = "Janet Doe"
8    println(name + " is SomeBody")
9  }
10 val sb = new SomeBody
11
12 class EveryBody {
13   val all = Vector(new SomeBody,
14     new SomeBody, new SomeBody)
15 }
16 val eb = new EveryBody
```

A class without a body simply has a name, as on line 3. To create an instance of a class, you use the **new** keyword as on lines 4, 10 and 16.

Lines 7 and 13 show fields within class bodies. Fields can be any type; here we see a **String** on line7 and a **Vector** holding **SomeBody** objects on line 13. Fields with fixed contents are of limited use; things will get more interesting later on.

Note line 8, which is not part of a field or method. When you run this script, you'll see that line 8 executes every time a **SomeBody** object is created.

Here's a class with methods:

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

class Temperature {
  var current = 0.0
  var scale = "f"
  def setFahrenheit(now:Double):Unit = {
    current = now
    scale = "f"
  }
  def setCelsius(now:Double):Unit = {
    current = now
    scale = "c"
  }
  def getFahrenheit():Double = {
    if(scale == "f")
      current
    else
      current * 9.0/5.0 + 32.0
  }
21    def getCelsius():Double = {
22      if(scale == "c")
23        current
24      else
25        (current - 32.0) * 5.0/9.0
26    }
27  }
28  
29  val temp = new Temperature
30  temp.setFahrenheit(98.6)
31  temp.getFahrenheit() is 98.6
32  temp.getCelsius is 37.0
33  temp.setCelsius(100.0)
34  temp.getFahrenheit is 212.0
```

These methods are just like those we've defined *outside* of classes, except that they belong to the class and have unqualified access to the other members of the class – such as **current** and **scale** (methods can also call other methods in the class without qualification).

Notice that line 29 uses a **val** for **temp**, but lines 30 and 33 modify the **Temperature** object. The **val** declaration simply prevents the reference **temp** from being reassigned to a new object; it does not restrict the behavior of the object itself.

On lines 31, 32 and 34 you can see that, if a method has an empty argument list, Scala allows you to call it with or without parentheses.

The following two classes are the foundation of a tic-tac-toe game. They also further demonstrate conditional expressions:

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

```

```
 4  class Cell {
 5    var entry = ' '
 6    def set(e:Char):String = {
 7      if(entry==' ' && (e=='X' || e=='O')) {
 8        entry = e
 9        "successful move"
10      } else
11        "invalid move"
12    }
13  }
14
15  class Grid {
16    val cells = Vector(
17      Vector(new Cell, new Cell, new Cell),
18      Vector(new Cell, new Cell, new Cell),
19      Vector(new Cell, new Cell, new Cell)
20    )
21    def play(e:Char, x:Int, y:Int):String = {
22      if(x < 0 || x > 2 || y < 0 || y > 2)
23        "invalid move"
24      else
25        cells(x)(y).set(e)
26    }
27  }
28
29  val grid = new Grid
30  grid.play('X', 1, 1) is "successful move"
31  grid.play('X', 1, 1) is "invalid move"
32  grid.play('O', 1, 3) is "invalid move"
```

The **entry** field in **Cell** is a **var** so that it can be modified. The single quotes in the initialization on line 5 produce a **Char** type, so all assignments to **entry** must also be **Char**s.

The **set** method starting on line 6 tests that the space is available and that you've passed it the right character; it returns a **String** result to indicate success or failure.

The **Grid** class contains a **Vector** containing three **Vector**s, each containing three **Cell**s – a matrix. The **play** method checks to see if the **x** and **y** indices are within range, then indexes into the matrix on line 25, relying on the tests performed by the **set** method.

For Loops

All programming languages have looping constructs and virtually always have a **for** loop, but often resort to counting through integers which you then use as an index into a sequence. Scala's **for** focuses on the sequence rather than the numbers. For example, this **for** selects each element in a **Vector**:

```
// ForVector.scala
val v = Vector("Somewhere", "over",
  "the", "rainbow")
for(word <- v) {
  println(word)
}
```

The left arrow **<-** selects each element from the generator expression on the right. Here the generator expression is just a **Vector** but it can be more complex. Note that you don't have to declare **word** as a **var** or **val** – it's automatically a **val**. Unlike the integral-indexing approach used by many languages, Scala's **for** automatically keeps track of the number of elements in the generator expression, eliminating the errors that come from accidentally indexing off the end of a sequence.

It's still possible to step through integral values using a **Range** object as a generator:

```
1  // ForWithRanges.scala
2  import com.atomicscala.AtomicTest._
3
4  var result = ""
5  for(i <- Range(0, 10)) {
6    result += i + " "
7  }
8  result is "0 1 2 3 4 5 6 7 8 9 "
9
10 result = ""
11 for(i <- Range(1, 21, 3)) {
12   result += i + " "
13 }
14 result is "1 4 7 10 13 16 19 "
```

The end value is excluded, as you see on line 8. The optional third argument to **Range** is the step value, used on line 11.

Scala provides some readable shorthand to produce **Range**s:

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

var result = ""
for(i <- 0 until 10) {
  result += i + " "
}
result is "0 1 2 3 4 5 6 7 8 9 "

result = ""
for(i <- 0 to 10) {
  result += i + " "
}
result is "0 1 2 3 4 5 6 7 8 9 10 "

result = ""
for(i <- 'a' to 'h') {
```

```
18    result += i + " "
19  }
20  result is "a b c d e f g h "
```

The effect of **until** is the same as **Range**, but **to** includes the endpoint. Note the clarity of creating a **Range** of characters on line 17.

Exercises

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

Whenever possible, use **AtomicTest** to test the solutions for these exercises.

1. Create a **Vector** filled with **Char**s, one filled with **Int**s, and one filled with **String**s. Sort each **Vector** and produce the **min** and **max** for each. Write a **for** loop for each sorted **Vector** that appends its elements, separated by spaces, to a **String**.

2. Create a **Vector** containing all the **Vector**s from Exercise 1. Write a **for** loop within a **for** loop to move through this **Vector** of **Vector**s and append all the elements to a single **String**.

3. In the REPL, create a single **Vector** containing a **Char**, an **Int**, a **String** and a **Double**. What type does this **Vector** contain? Try to find the **max** of your **Vector**. Does this make sense?

4. Modify **BasicMethods.scala** so the two methods are part of a class. Put the class in a **package** and compile it. Import the resulting library into a script and test it.

5. Create a **package** containing the classes in **ClassBodies.scala**. Compile this package, then import it into a script. Modify the classes by adding methods that produce results that can be tested with **AtomicTest**.

6. Add Kelvin temperature units to **Temperature.scala** (Kelvin is Celsius + 273.15). When writing the new code, call the existing methods whenever possible.

7. Add a method to **TicTacToe.scala** that displays the game board (hint: use a **for** loop within a **for** loop). Call this method automatically for each move.

8. Add a method to **TicTacToe.scala** that determines whether there is a winner or if the game is a draw. Call this method automatically for each move.

Pattern Matching

A large part of computer programming involves making comparisons and taking action based on whether something matches. Anything that makes this task easier is a boon for programmers, so Scala provides extensive language support in the form of *pattern matching*.

A *match expression* compares a value against a selection of possibilities. All match expressions begin with the value you want to compare, followed by the keyword **match**, an opening curly brace, and then a set of possible matches and their associated actions, and ends with a closing curly brace. Each possible match and its associated action begins with the keyword **case** followed by an expression. The expression is evaluated and compared to the target value. If it matches, the expression to the right of the **=>** ("rocket") produces the result of the **match** expression.

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

def matchColor(color:String):String = {
  color match {
    case "red" => "RED"
    case "blue" => "BLUE"
    case "green" => "GREEN"
    case _ => "UNKNOWN COLOR: " + color
  }
}

matchColor("white") is
  "UNKNOWN COLOR: white"
matchColor("blue") is "BLUE"
```

Line 5 begins the match expression: The value name **color** followed by the **match** keyword and a set of expressions in curly braces, representing things to match against. Line 6-8 compare the value **color** to **"red"**, **"blue"** and **"green"**. The first successful match finishes the execution of the pattern match – in this case, a **String** is produced by the pattern match which becomes the return value of **matchColor**.

Line 9 is another special use of the "_" (underscore). Here, it is a *wildcard*, and matches anything that isn't matched above. When we test against **"white"** on line 13, it doesn't match red, blue, or green, and hits the *wildcard pattern*, which always appears last in the match list. If you do not include it, you will get an error when you try to match on something other than the listed patterns.

The example shown here only matches against a simple type (**String**) but you'll learn in later atoms that pattern matching can be much more sophisticated.

You'll notice that pattern matching can overlap with the functionality of **if** statements. Because pattern matching is more flexible and powerful, we prefer it over **if** statements when there's a choice.

Exercises

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

1. Rewrite **matchColor** using **if/else**. Which approach seems more straightforward? Satisfy the following tests:
```
matchColor("white") is
"UNKNOWN COLOR: white"
matchColor("blue") is "BLUE"
```

2. Rewrite **oneOrTheOther** from More Conditionals using pattern matching. Satisfy the following tests:
```
val v = Vector(1)
val v2 = Vector(3, 4)
```

```
oneOrTheOther(v == v.reverse) is "True!"
oneOrTheOther(v2 == v2.reverse) is
"It's false"
```

3. Rewrite **checkTruth** from More Conditionals with pattern matching. Satisfy the following tests:

```
checkTruth(true || false, true) is
  "Both are true"
checkTruth(1 > 0 && -1 < 0, 1 == 2) is
  "First: true, second: false"
checkTruth(1 >= 2, 1 >= 1) is
  "First: false, second: true"
checkTruth(true && false, false && true) is
"Both are false"
```

4. Create a method **forecast** that represents the percentage of cloudiness, and use it to produce a "weather forecast" string such as "Sunny" (100), "Mostly Sunny" (80), "Partly Sunny" (50), "Mostly Cloudy" (20), and "Cloudy" (0). For this exercise, only match for the legal values 100, 80, 50, 20, and 0 (we will revisit this exercise later). Everything else should produce "Unknown." Satisfy the following tests:

```
forecast(100) is "Sunny"
forecast(80) is "Mostly Sunny"
forecast(50) is "Partly Sunny"
forecast(20) is "Mostly Cloudy"
forecast(0) is "Cloudy"
forecast(15) is "Unknown"
```

5. Create a **Vector** named **sunnyData** that holds the values (100, 80, 50, 20, 0, 15). Use a **for** loop to call **forecast** with the contents of **sunnyData**. Display the answers and ensure that they match the responses above.

Class Arguments

When you create a new object, you'll typically want to pass information into that object in order to initialize it. You do this using *class arguments*. The class argument list looks like a method argument list, but placed after the class name:

```
1   // ClassArg.scala
2   import com.atomicscala.AtomicTest._
3
4   class ClassArg(a:Int) {
5     println(f)
6     def f():Int = { a * 10 }
7   }
8
9   val ca = new ClassArg(19)
10  ca.f() is 190
11  // ca.a // error
```

Now the **new** expression requires an argument (try it without one). The initialization of **a** happens before we enter the class body, so it's always set to the expected value. And even though the **println** on line 5 appears to happen *before* **f** is defined, all the definitions (values and methods) are actually initialized before the rest of the body is executed, so it doesn't matter that line 5 appears first – **f** is still available at that point.

Notice that **a** is not accessible outside the class body, as shown by the error which comes from uncommenting line 11. If you want **a** to be visible outside the class body, declare it as a **var** or **val** in the argument list:

```
1   // VisibleClassArgs.scala
2   import com.atomicscala.AtomicTest._
3
```

```
4  class ClassArg2(var a:Int)
5  class ClassArg3(val a:Int)
6
7  val ca2 = new ClassArg2(20)
8  val ca3 = new ClassArg3(21)
9
10 ca2.a is 20
11 ca3.a is 21
12 ca2.a = 24
13 ca2.a is 24
14 // Can't do this: ca3.a = 35
```

These class definitions have no explicit class bodies (the bodies are implied). Because **a** is declared using **var** or **val**, it becomes visible outside the class body as seen on lines 10-13. Class arguments that are declared with **val** cannot be changed outside of the class, but those that are declared with **var** can, as you might expect. You can see this on lines 12-14.

Note that **ca2** is a **val** (lines 7). Does the fact that you can change the value of **a** on line 12 surprise you? It might help to think of an analogy. Consider a house as a **val**, and a sofa inside the house as a **var**. You can change the sofa inside the house because it's a **var**. You can't change the house, though – it's a **val**. Here, making **ca2** and **ca3** **val**s means you can't point them at other objects. But the **val** doesn't control the insides of the object.

Your class can have many arguments:

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

class Sum3(a1:Int, a2:Int, a3:Int) {
  def result():Int = { a1 + a2 + a3 }
}

```

```
new Sum3(13, 27, 44).result() is 84
```

You can support any number of arguments using a *variable argument list*, denoted by a trailing '*':

```
1   // VariableClassArgs.scala
2   import com.atomicscala.AtomicTest._
3
4   class Sum(args:Int*) {
5     def result():Int = {
6       var total = 0
7       for(n <- args) {
8         total += n
9       }
10      total
11    }
12  }
13
14  new Sum(13, 27, 44).result() is 84
15  new Sum(1, 3, 5, 7, 9, 11).result() is 36
```

The trailing '*' on **args** (line 4) turns the passed arguments into a sequence that can be traversed using a '**<-**' in a **for** expression. Methods can also have variable argument lists.

Exercises

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

1. Create a new class **Family** that takes a variable argument list representing the names of family members. Satisfy the following tests:

```
val family1 = new Family("Mom",
  "Dad", "Sally", "Dick")
family1.familySize() is 4
val family2 =
  new Family("Dad", "Mom", "Harry")
family2.familySize() is 3
```

2. Adapt the **Family** class definition to include class arguments for a mother, father, and a variable number of children. What changes did you have to make? Satisfy the following tests:
   ```
   val family3 = new FlexibleFamily(
     "Mom", "Dad", "Sally", "Dick")
   family3.familySize() is 4
   val family4 =
     new FlexibleFamily("Dad", "Mom", "Harry")
   family4.familySize() is 3
   ```

3. Does it work to leave out the kids altogether? Do you need to modify your **familySize** method? Satisfy the following test:
   ```
   val familyNoKids =
     new FlexibleFamily("Mom", "Dad")
   familyNoKids.familySize() is 2
   ```

4. Can you use a variable argument list for both parents and children?

5. Can you put the variable argument list first, and the parents last?

6. Fields contained a class **Cup2** with a field **percentFull**. Rewrite that class definition, using a class argument instead of defining a field.

7. Using your solution for Exercise 6, can you get and set the value of **percentFull** without writing any new methods? Try it!

8. Continue working with the **Cup2** class. Modify the **add** method to take a variable argument list. Specify any number of pours (increase) and spills (decrease = increase with a negative value) and return the resulting value. Satisfy the following tests:
   ```
   val cup5 = new Cup5(0)
   cup5.increase(20, 30, 50,
     20, 10, -10, -40, 10, 50) is 100
   ```

```
cup5.increase(10, 10, -10, 10,
  90, 70, -70) is 30
```

9. Write a method that squares a variable argument list of numbers and returns the sum. Satisfy the following tests:
```
squareThem(2) is 4
squareThem(2, 4) is 20
squareThem(1, 2, 4) is 21
```

Named & Default Arguments

When creating an instance of a class that has an argument list, you can specify the argument names, as on lines 4 and 5:

```
1  // NamedArguments.scala
2
3  class Color(red:Int, blue:Int, green:Int)
4  new Color(red = 80, blue = 9, green = 100)
5  new Color(80, 9, green = 100)
```

All the argument names are specified on line 4, and on line 5 you can see how to choose the ones you want to name.

Named arguments are useful for code readability, and this is especially true for long and complex argument lists – named arguments can make it clear enough that the reader doesn't need to check the documentation.

Named arguments are even more useful when combined with *default arguments*: default values for arguments in the class definition:

```
// NamedAndDefaultArgs.scala

class Color2(red:Int = 100,
  blue:Int = 100, green:Int = 100)
new Color2(20)
new Color2(20, 17)
new Color2(blue = 20)
new Color2(red = 11, green = 42)
```

Any argument you don't specify gets its default value. With default arguments, the person creating the object need only specify the arguments that are different from the defaults. If you have a long argument list, this can greatly simplify the resulting code, making it far easier to write and (more importantly) to read.

Named and default arguments also work in method argument lists.

Named and default arguments work with variable argument lists (introduced in Class Arguments); however (as is always the case) the variable argument list must appear last. Also, the variable argument list itself cannot support default arguments.

Warning: Named and default arguments currently have an idiosyncrasy when combined with a variable argument list – you cannot vary the order of the named arguments from their definition. For example:

```
// Family.scala

class Family(mom:String, dad:String,
  kids:String*)

new Family(mom="Mom", dad="Dad")
// Doesn't work:
// new Family(dad="Dad", mom="Mom")

new Family(mom="Mom", dad="Dad",
  kids="Sammy", "Bobby")
// Doesn't work:
/* new Family(dad="Dad", mom="Mom",
  kids="Sammy", "Bobby") */
```

Ordinarily, named arguments would allow us to change the order of the parents, so we would be able to specify first **dad**, then **mom**. When you add a variable argument list, however, you can no longer reorder arguments by naming them. The reason for this restriction is beyond the scope of this book; we'll simply recommend that you avoid using named arguments with variable argument lists.

Exercises

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

1. Define a class **SimpleTime** that takes two arguments: an **Int** that represents hours, and an **Int** that represents minutes. Use named arguments to create a **SimpleTime** object. Satisfy the following tests:

```
val t = new SimpleTime(hours=5, minutes=30)
t.hours is 5
t.minutes is 30
```

2. Using the solution for **SimpleTime** above, default **minutes** to 0 so that you don't have to specify them. Satisfy the following tests:

```
val t2 = new SimpleTime2(hours=10)
t2.hours is 10
t2.minutes is 0
```

3. Create a class **Planet** that has, by default, a single moon. The **Planet** class should have a name (**String**) and description (**String**). Use named arguments to specify the name and description, and a default for the number of moons. Satisfy the following tests:

```
val p = new Planet(name = "Mercury",
  description = "small and hot planet",
  moons = 0)
p.hasMoon is false
```

4. Modify your solution for the previous exercise by changing the order of the arguments that you use to create the **Planet**. Did you have to change any code? Satisfy the following tests:
```
val earth = new Planet(moons = 1,
  name = "Earth",
  description = "a hospitable planet")
earth.hasMoon is true
```

5. Can you modify your solution for Exercise 2 in Class Arguments to default the mother's name to "Mom" and the father's name to "Dad?" Why do you get an error? Hint: Scala does a good job of telling you what the problem is.

6. Demonstrate that named and default arguments can be used with methods. Create a class **Item** that takes two class arguments: A **String** for **name** and a **Double** for **price**. Add a method **cost** which has named arguments for **grocery** (**Boolean**), **medication** (**Boolean**), and **taxRate** (**Double**). Default **grocery** and **medication** to **false**, **taxRate** to **0.10**. In this scenario, groceries and medications are not taxable. Return the total cost of the item by calculating the appropriate tax. Satisfy the following tests:
```
val flour = new Item(name="flour", 4)
flour.cost(grocery=true) is 4
val sunscreen = new Item(
  name="sunscreen", 3)
sunscreen.cost() is 3.3
val tv = new Item(name="television", 500)
tv.cost(rate = 0.06) is 530
```

Overloading

The term *overload* refers to the name of a method: You can use the same name ("overload" that name) for different methods as long as the argument lists differ.

```
 1  // Overloading.scala
 2  import com.atomicscala.AtomicTest._
 3
 4  class Overloading1 {
 5    def f():Int = { 88 }
 6    def f(n:Int):Int = { n + 2 }
 7  }
 8
 9  class Overloading2 {
10    def f():Int = { 99 }
11    def f(n:Int):Int = { n + 3 }
12  }
13
14  val mo1 = new Overloading1
15  val mo2 = new Overloading2
16  mo1.f() is 88
17  mo1.f(11) is 13
18  mo2.f() is 99
19  mo2.f(11) is 14
```

On lines 5 and 6 you see two methods with the same name, **f**. We say that the method's *signature* is comprised of the name, argument list and return type. Scala distinguishes one method from another by comparing the signatures. The only difference between the two signatures on lines 5 and 6 is the argument list, and that's all Scala needs in order to decide that the two methods are different. The calls on lines 16 and 17 show that they are indeed different methods. A method signature also includes information about the enclosing class.

Thus, the overloaded **f** methods in **Overloading1** don't clash with the **f** methods in **Overloading2**.

Why is overloading useful? It allows you to express "variations on a theme" more clearly than if you were forced to use different method names. Let's say you want method for adding:

```
1   // OverloadingAdd.scala
2   import com.atomicscala.AtomicTest._
3
4   def addInt(i:Int, j:Int):Int = { i + j }
5   def addDouble(i:Double, j:Double):Double ={
6     i + j
7   }
8
9   def add(i:Int, j:Int):Int = { i + j }
10  def add(i:Double, j:Double):Double = {
11    i + j
12  }
13
14  addInt(5, 6) is add(5, 6)
15
16  addDouble(56.23, 44.77) is
17    add(56.23, 44.77)
```

addInt takes two **Int**s and returns an **Int**, while **addDouble** takes two **Double**s and returns a **Double**. Without overloading, you can't just name the operation, so programmers typically conflate *what* with *how* in order to produce unique names (this is not required; you can create unique names using random characters but the typical pattern is to use more meaningful information like argument types). In contrast, the overloaded **add** on lines 9 and 10 is much clearer.

The lack of overloading in a language is not a terrible hardship, but it provides a valuable simplification that produces more readable code. With overloading, you can just say *what*, which raises the level of abstraction and puts less mental load on the reader. If you want to know *how*, you can look at the arguments. Notice also that overloading reduces redundancy: If we must say **addInt** and **addDouble**, then we essentially repeat the argument information in the method name.

Overloading doesn't work in the REPL. If you define the methods above, the second **add** overwrites rather than overloads the first **add**. The REPL is great for simple experimentation, but even a little more complexity can produce inconsistent results. To overcome this limitation, use the REPL's **:paste** mode described in Summary 1 (in "Expressions & Conditionals").

Exercises

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

1. Modify **Overloading.scala** so that the argument lists for all the methods are identical. Observe the error messages.

2. Create five overloaded methods that add their arguments together. Create the first with no arguments, the second with one argument, etc. Satisfy the following tests:
   ```
   f() is 0
   f(1) is 1
   f(1, 2) is 3
   f(1, 2, 3) is 6
   f(1, 2, 3, 4) is 10
   ```

3. Modify Exercise 2 to define the methods inside of classes.

4. Modify your solution for Exercise 3 to add a method with the same name and arguments, but a different return type. Does that work? Does it matter if you use an explicit return type or type inference for the return type?

Constructors

A significant stumbling block in programming is initialization. Your code can do all the right things, but if you don't set up the proper initial conditions it won't work correctly.

Each object is its own isolated world. A program is a collection of objects, so correct initialization of each object solves a large part of the initialization problem. Scala provides mechanisms to guarantee proper object initialization, some of which you've seen in the last few atoms.

The *constructor* is the code that "constructs" a new object. The constructor is the combined effect of the class argument list – initialized before entering the class body – and the class body, whose statements execute from top to bottom.

The simplest form of a constructor is a single line class definition, with no class arguments and no executable lines of code, such as:

```
class Bear
```

In Fields, the constructor initializes the fields to the values specified, or to defaults if no values were specified. In Class Arguments, the constructor quietly initializes those values and makes them accessible to other objects; it also unravels a variable argument list.

In those cases, we didn't write constructor code – Scala did it for us. For more customization, add your own constructor code. For example:

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

class Coffee(val shots:Int = 2,
             val decaf:Boolean = false,
             val milk:Boolean = false,
             val toGo:Boolean = false,
             val syrup:String = "") {
  var result = ""
  println(shots, decaf, milk, toGo, syrup)
  def getCup():Unit = {
    if(toGo)
      result += "ToGoCup "
    else
      result += "HereCup "
  }
  def pourShots():Unit = {
    for(s <- 0 until shots)
      if(decaf)
        result += "decaf shot "
      else
        result += "shot "
  }
  def addMilk():Unit = {
    if(milk)
      result += "milk "
  }
  def addSyrup():Unit = {
    result += syrup
  }
  getCup()
  pourShots()
  addMilk()
  addSyrup()
}

```

```
37 val usual = new Coffee
38 usual.result is "HereCup shot shot "
39 val mocha = new Coffee(decaf = true,
40   toGo = true, syrup = "Chocolate")
41 mocha.result is
42 "ToGoCup decaf shot decaf shot Chocolate"
```

Notice that the methods have access to the class arguments without explicitly passing them, and that **result** is available as an object field. Although all the methods are called at the end of the class body, they could actually be called at the beginning or any other place in the body (try it).

When the **Coffee** constructor completes, it guarantees that the class body has successfully run and all proper initialization has occurred; the **result** field captures all the operations.

We're using default arguments here, just as we can use them in any other method. If all arguments have defaults, you can say **new Coffee** without using parentheses, as on line 37.

Exercises

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

1. Modify **Coffee.scala** so you can specify some caffeinated shots and some decaf shots. Satisfy the following tests:

```
val doubleHalfCaf =
  new Coffee(shots=2, decaf=1)
val tripleHalfCaf =
  new Coffee(shots=3, decaf=2)
doubleHalfCaf.decaf is 1
doubleHalfCaf.caf is 1
doubleHalfCaf.shots is 2
tripleHalfCaf.decaf is 2
tripleHalfCaf.caf is 1
tripleHalfCaf.shots is 3
```

2. Create a new class **Tea** that has 2 methods: **describe**, which includes information about whether the tea includes milk, sugar, is decaffeinated, and includes the name; and **calories**, which adds 100 calories for milk and 16 calories for sugar. Satisfy the following tests:

```
val tea = new Tea
tea.describe is "Earl Grey"
tea.calories is 0

val lemonZinger = new Tea(
  decaf = true, name="Lemon Zinger")
lemonZinger.describe is
  "Lemon Zinger decaf"
lemonZinger.calories is 0

val sweetGreen = new Tea(
  name="Jasmine Green", sugar=true)
sweetGreen.describe is
  "Jasmine Green + sugar"
sweetGreen.calories is 16

val teaLatte = new Tea(
  sugar=true, milk=true)
teaLatte.describe is
  "Earl Grey + milk + sugar"
  teaLatte.calories is 116
```

3. Use your solution for Exercise 2 as a starting point. Make decaf, milk, sugar and name accessible outside of the class. Satisfy the following tests:

```
val tea = new Tea2
tea.describe is "Earl Grey"
tea.calories is 0
tea.name is "Earl Grey"

val lemonZinger = new Tea2(decaf = true,
  name="Lemon Zinger")
lemonZinger.describe is
  "Lemon Zinger decaf"
lemonZinger.calories is 0
lemonZinger.decaf is true

val sweetGreen = new Tea2(
  name="Jasmine Green", sugar=true)
sweetGreen.describe is
  "Jasmine Green + sugar"
sweetGreen.calories is 16
sweetGreen.sugar is true

val teaLatte = new Tea2(sugar=true,
  milk=true)
teaLatte.describe is
  "Earl Grey + milk + sugar"
teaLatte.calories is 116
teaLatte.milk is true
```

Auxiliary Constructors

Named and default arguments in the class argument list can construct objects in multiple ways. We can also use *constructor overloading* by creating multiple constructors (the name is overloaded in this case because you're making different ways to create objects of the same class). To create an overloaded constructor you define a method (with a distinct argument list) called **this** (a keyword). Overloaded constructors have a special name in Scala: *auxiliary constructors*.

Because constructors are responsible for the very important act of initialization, constructor overloading has an additional constraint: all auxiliary constructors must first call the *primary constructor*, which is the constructor produced by the class argument list together with the class body. To call the primary constructor within an auxiliary constructor, you don't use the class name, but instead the **this** keyword:

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

class GardenGnome(val height:Double,
  val weight:Double, val happy:Boolean) {
  println("Inside primary constructor")
  var painted = true
  def magic(level:Int):String = {
    "Poof! " + level
  }
  def this(height:Double) {
    this(height, 100.0, true)
  }
14    def this(name:String) = {
15      this(15.0)
16      painted is true
17    }
18    def show():String = {
19      height + " " + weight +
20      " " + happy + " " + painted
21    }
22  }
23
24  new GardenGnome(20.0, 110.0, false).
25  show() is "20.0 110.0 false true"
26  new GardenGnome("Bob").show() is
27  "15.0 100.0 true true"
```

The first auxiliary constructor begins on line 11. You do not declare a return type for an auxiliary constructor, and it doesn't matter whether you include an '=' (line 14) or leave it out (line 11). The first line in any auxiliary constructor must be a call to either the primary constructor (line 12) or another auxiliary constructor (line 15). This means that, ultimately, the primary constructor is always called first, which guarantees that the object is properly initialized before the auxiliary constructor comes into play. You can't use **val** or **var** for the auxiliary constructor arguments; that would mean the field is generated by only *that* auxiliary constructor. By forcing the field-generating class arguments to only be in the primary constructor, Scala guarantees that all objects have the same structure.

The expressions within a constructor are treated as statements, so they are only executed for their side effects, which in this case is how they affect the state of the object being created. The result of the final expression in a constructor is not returned, but ignored. In addition, Scala will not allow you to short-circuit the creation of an object. You cannot, for example, put a **return** in the middle of a class body (try it and see the error message).

You'll probably solve most of your constructor needs using named and default arguments, but sometimes you need to overload a constructor.

Exercises

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

1. Create a class called **ClothesWasher** with a default constructor and two auxiliary constructors, one that specifies **modelName** (as a **String**) and one that specifies **capacity** (as a **Double**).

2. Create a class **ClothesWasher2** that looks just like your solution for Exercise 1, but use named and default arguments instead so that you can produce the same results with just a default constructor.

3. Show that the first line of an auxiliary constructor must be a call to the primary constructor.

4. Recall from Overloading that methods can be overloaded in Scala, and that this is different from the way that we overload constructors (by writing auxiliary constructors). Add two methods to your solution for Exercise 1 to show that methods can be overloaded. Satisfy the following tests:
```
val washer =
  new ClothesWasher3("LG 100", 3.6)
washer.wash(2, 1) is
"Wash used 2 bleach and 1 fabric softener"
washer.wash() is "Simple wash"
```

Class Exercises

Now you're ready to solve some more comprehensive exercises about defining and using classes.

Exercises

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

1. Make a class **Dimension** that has an integer field **height** and an integer field **width** that can be both retrieved and modified from outside the class. Satisfy the following tests:
```
val c = new Dimension(5,7)
c.height is 5
c.height = 10
c.height is 10
c.width = 19
c.width is 19
```

2. Make a class **Info** that has a **String** field **name** that can be retrieved from outside the class (but not modified) and a **String** field **description** that can be both modified and retrieved from outside of the class. Satisfy the following tests:
```
val info = new Info("stuff", "Something")
info.name is "stuff"
info.description is "Something"
info.description = "Something else"
info.description is "Something else"
```

3. Working from your solution to Exercise 2, modify the **Info** class to satisfy the following test:
```
info.name = "This is the new name"
info.name is "This is the new name"
```

4. Modify **SimpleTime** (from Named and Default Arguments) to add a method **subtract** that subtracts one **SimpleTime** object from

another. If the second time is greater than the first, just return 0's. Satisfy the following tests:

```
val t1 = new SimpleTime(10, 30)
val t2 = new SimpleTime(9, 30)
val st = t1.subtract(t2)
st.hours is 1
st.minutes is 0
val st2 = new SimpleTime(10, 30).
  subtract(new SimpleTime(9, 45))
st2.hours is 0
st2.minutes is 45
val st3 = new SimpleTime(9, 30).
  subtract(new SimpleTime(10, 0))
st3.hours is 0
st3.minutes is 0
```

5. Modify your **SimpleTime** solution to use default arguments for minutes (see Named and Default Arguments). Satisfy the following tests:

```
val anotherT1 =
  new SimpleTimeDefault(10, 30)
val anotherT2 = new SimpleTimeDefault(9)
val anotherST =
  anotherT1.subtract(anotherT2)
anotherST.hours is 1
anotherST.minutes is 30
val anotherST2 = new SimpleTimeDefault(10).
  subtract(new SimpleTimeDefault(9, 45))
anotherST2.hours is 0
anotherST2.minutes is 15
```

6. Modify your solution for Exercise 5 to use an auxiliary constructor. Again, satisfy the following tests:
```
val auxT1 = new SimpleTimeAux(10, 5)
val auxT2 = new SimpleTimeAux(6)
val auxST = auxT1.subtract(auxT2)
auxST.hours is 4
auxST.minutes is 5
val auxST2= new SimpleTimeAux(12).subtract(
  new SimpleTimeAux(9, 45))
auxST2.hours is 2
auxST2.minutes is 15
```

7. Defaulting both hours and minutes in the previous exercise is problematic. Can you see why? Can you figure out how to use named arguments to solve this problem? Did you have to change any code?

Case Classes

The **class** mechanism does a fair amount of work for you, but there is still a significant amount of repetitive code when creating some common types of classes. Scala tries to eliminate repetition whenever it can, and that's what the *case class* does. You define a case class like this:

```
case class TypeName(arg1:Type, arg2:Type, ...)
```

At first glance, a **case class** looks like an ordinary class with the **case** keyword in front of it. However, a case class automatically creates all the class arguments as if you've put the **val** keyword in front of each one in an ordinary class. If you want to be verbose, you can specify **val** before each field name and produce an identical result. If you need a class argument to be a **var** instead, put a **var** in front of that argument.

Whenever you primarily need to store data, case classes simplify your code and perform a lot of common work for you.

Here we define two new types, **Dog** and **Cat**, and instances of each:

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

case class Dog(name:String)
val dog1 = Dog("Henry")
val dog2 = Dog("Cleo")
val dogs = Vector(dog1, dog2)
dogs is Vector(Dog("Henry"), Dog("Cleo"))

case class Cat(name:String, age:Int)
val cats =
  Vector(Cat("Miffy", 3), Cat("Rags", 2))
```

```
13 cats is
14   "Vector(Cat(Miffy,3), Cat(Rags,2))"
```

With case classes, unlike regular classes, we don't have to use the **new** keyword when creating an object. You can see this wherever **Dog** and **Cat** objects are created.

Case classes also provide a way to print objects in a nice, readable format without having to define a special display method. You see both the name of the case class (**Dog** or **Cat**) and the field information for each object, as on line 14.

This is only a basic introduction to case classes. You'll see more of their value as the book progresses.

Exercises

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

1. Create a **case class** to represent a **Person** in an address book, complete with **String**s for the name and a **String** for contact information. Satisfy the following tests:
```
val p = Person("Jane", "Smile",
"jane@smile.com")
p.first is "Jane"
p.last is "Smile"
p.email is "jane@smile.com"
```

2. Create some **Person** objects. Put the **Person** objects in a **Vector**. Satisfy the following tests:
```
val people = Vector(
Person("Jane","Smile","jane@smile.com"),
Person("Ron","House","ron@house.com"),
Person("Sally","Dove","sally@dove.com"))
people(0) is
"Person(Jane,Smile,jane@smile.com)"
```

```
people(1) is
"Person(Ron,House,ron@house.com)"
people(2) is
"Person(Sally,Dove,sally@dove.com)"
```

3. First, create a **case class** that represents a **Dog**, using a **String** for name and a **String** for breed. Then, create a **Vector** of **Dog**s. Satisfy the following tests:

```
val dogs = Vector(
    /* Insert Vector initialization */
  )
dogs(0) is "Dog(Fido,Golden Lab)"
dogs(1) is "Dog(Ruff,Alaskan Malamute)"
dogs(2) is "Dog(Fifi,Miniature Poodle)"
```

4. As in Class Exercises, make a **case class Dimension** that has an integer field **height** and an integer field **width** that can be both retrieved and modified from outside of the class. Create and print an object of this class. How does this solution differ from your solution for Exercise 1 in Class Exercises? Satisfy the following tests:

```
val c = new Dimension(5,7)
c.height is 5
c.height = 10
c.height is 10
c.width = 19
c.width is 19
```

5. Modify your solution for Exercise 4, using one ordinary (**val**) argument for height and one **var** argument for width. Demonstrate that one is read-only and the other is modifiable.

6. Can you use default arguments with **case class**es? Repeat Exercise 5 from Class Exercises to find out. How does your solution differ, if at all? Satisfy the following tests:

```
val anotherT1 =
  new SimpleTimeDefault(10, 30)
val anotherT2 = new SimpleTimeDefault(9)
val anotherST =
  anotherT1.subtract(anotherT2)
anotherST.hours is 1
anotherST.minutes is 30
val anotherST2 =
  new SimpleTimeDefault(10).subtract(
  new SimpleTimeDefault(9, 45))
anotherST2.hours is 0
anotherST2.minutes is 15
```

String Interpolation

String interpolation allows you to create strings containing formatted values. To produce this effect, put an '**s**' in front of the string, and a '**$**' before the identifier you want Scala to interpolate:

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

def i(s:String, n:Int, d:Double):String = {
  s"first: $s, second: $n, third: $d"
}

i("hi", 11, 3.14) is
"first: hi, second: 11, third: 3.14"
```

As you can see, any identifier preceded by a '**$**' is converted to string form.

You can even evaluate and convert an expression by placing it inside '**${}**' (among other things, this is helpful for systems that generate web pages):

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

def f(n:Int):Int = { n * 11 }

s"f(7) is ${f(7)}!" is "f(7) is 77!"
```

The expressions can be complex, but it's more readable to keep them simple.

Interpolation also works with case classes:

```
1   // CaseClassInterpolation.scala
2   import com.atomicscala.AtomicTest._
3
4   case class Sky(color:String)
5
6   s"""${new Sky("Blue")}""" is "Sky(Blue)"
```

We use triple quotes around the string on line 6 to allow the quotes on the argument of the **Sky** constructor.

Exercises

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

1. The Garden Gnome example in Auxiliary Constructors has a **show** method to display information about a gnome. Rewrite **show** using **String** interpolation. Satisfy the following tests:
   ```
   val gnome =
     new GardenGnome(20.0, 110.0, false)
   gnome.show() is "20.0 110.0 false true"
   val bob = new GardenGnome("Bob")
   bob.show() is "15.0 100.0 true true"
   ```

2. Use **GardenGnome**'s **magic** method with String Interpolation. Add a method **show** that takes one parameter, **level**, and calls **magic(level)** in place of **height** and **width**. Satisfy the following tests:
   ```
   val gnome =
     new GardenGnome(20.0, 50.0, false)
   gnome.show(87) is "Poof! 87 false true"
   val bob = new GardenGnome("Bob")
   bob.show(25) is "Poof! 25 true true"
   ```

3. Rework your solution for Exercise 1 to display **height** and **weight** with labels. Satisfy the following tests:
   ```
   val gnome =
     new GardenGnome(20.0, 110.0, false)
   ```

```
gnome.show() is "height: 20.0 " +
"weight: 110.0 happy: false painted: true"
val bob = new GardenGnome("Bob")
bob.show() is
"height: 15.0 weight: 100.0 true true"
```

Parameterized Types

We've made heavy use of Scala's type inference, letting the language do the work for us. Sometimes, however, Scala can't figure out what type to use, so we must help it (Scala will always complain when it needs help).

Although there are any number of situations that require explicit type declarations, this book uses containers in particular. When we use a **Vector**, for example, we sometimes need to tell Scala the type contained in that **Vector**. Often, Scala can figure this out:

```
1   // ParameterizedTypes.scala
2   import com.atomicscala.AtomicTest._
3
4   // Type is inferred:
5   val v1 = Vector(1,2,3)
6   val v2 = Vector("one", "two", "three")
7   // Exactly the same, but explicitly typed:
8   val p1:Vector[Int] = Vector(1,2,3)
9   val p2:Vector[String] =
10    Vector("one", "two", "three")
11
12  v1 is p1
13  v2 is p2
```

Lines 5 and 6 are the kind of thing we've been using – a simple **val** on the left and a simple **Vector** on the right. Scala looks at the initialization values and detects that the **Vector** on line 5 contains **Int**s and the **Vector** on line 6 contains **String**s.

We consider it a good idea to let Scala infer types whenever possible. It tends to make the code cleaner and easier to read. However, as an example, we'll rewrite lines 5 and 6 using explicit typing. Line 8 is the rewrite of line 5. The part on the right side of the equals sign is the same, but on the left side we add the colon and the type declaration, which is **Vector[Int]**. The square brackets are new here; they are how we denote a *type parameter*. For a container the type parameter tells the kind of object the container holds. You typically pronounce **Vector[Int]** “vector of Int,” and the same for other types of container: “list of Int,” “set of Int” and so forth.

Type parameters are useful for things other than containers, but you’ll usually see them with container-like objects, and in this book we’ll normally use **Vector** as our container.

Return types can also have parameters:

```
1  // ParameterizedReturnTypes.scala
2  import com.atomicscala.AtomicTest._
3
4  // Return type is inferred:
5  def inferred(c1:Char, c2:Char, c3:Char)={
6    Vector(c1, c2, c3)
7  }
8
9  // Explicit return type:
10 def explicit(c1:Char, c2:Char, c3:Char):
11   Vector[Char] = {
12   Vector(c1, c2, c3)
13 }
14
15 inferred('a', 'b', 'c') is
16   "Vector(a, b, c)"
17 explicit('a', 'b', 'c') is
18   "Vector(a, b, c)"
```

On line 5 we allow Scala to infer the return type of the method, and on line 11 we specify the method return type. You can’t just say that it returns a **Vector**; Scala will complain, so you must give the type parameter as well. When you specify the return type of a method, Scala can check and enforce your intention.

Exercises

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

1. Modify **explicit** in **ParameterizedReturnTypes.scala** so it creates and returns a **Vector** of **Double**. Satisfy the following tests:
   ```
   explicitDouble(1.0, 2.0, 3.0) is
   Vector(1.0, 2.0, 3.0)
   ```

2. Building on the previous exercise, change **explicit** to take a **Vector** and create and return a **List**. Refer to the ScalaDoc for **List**, if necessary. Satisfy the following tests:
   ```
   explicitList(Vector(10.0, 20.0)) is
   List(10.0, 20.0)
   explicitList(Vector(1, 2, 3)) is
   List(1.0, 2.0, 3.0)
   ```

3. Building on the previous exercise, change **explicit** to return a **Set**. Satisfy the following tests:
   ```
   explicitSet(Vector(10.0, 20.0, 10.0)) is
   Set(10.0, 20.0)
   explicitSet(Vector(1, 2, 3, 2, 3, 4)) is
   Set(1.0, 2.0, 3.0, 4.0)
   ```

4. In Pattern Matching, we created a method for weather forecasts using “Sunny” (100), “Mostly Sunny” (80), “Partly Sunny” (50), “Mostly Cloudy” (20), and “Cloudy” (0). Using parameterized types, create a method **historicalData** that counts the number of sunny, partly sunny, etc. days. Satisfy the following tests:
   ```
   val weather = Vector(100, 80, 20, 100, 20)
   historicalData(weather) is
   "Sunny=2, Mostly Sunny=1, Mostly Cloudy=2"
   ```

Functions as Objects

Everything in Scala is an object. Although objects often contain only ordinary methods and data, you can also pass methods – in the form of objects – as arguments to other methods. To accomplish this, the methods are packaged using *function objects*, often called simply *functions*.

For example, one helpful method available for sequences such as **Vector** is **foreach**, which takes its argument – a function – and applies it to each element in the sequence. Here we take each element of a **Vector** and display it along with a leading '>':

```
1  // DisplayVector.scala
2
3  def show(n:Int):Unit = { println("> "+ n) }
4  val v = Vector(1, 2, 3, 4)
5  v.foreach(show)
```

A method is attached to a class or object, while a function is its own object (this is why we can pass it around so easily). **show** is a method that becomes part of the object that Scala automatically creates for scripts. When we pass **show** as if it were a function, as on line 5, Scala automatically converts it to a function object. This is called *lifting*.

It turns out that functions that you pass as arguments to other methods or functions are often quite small, and it's also common that you only use them once. It seems like extra effort for the programmer and distracting for the reader to be forced to create a named method, then pass it as an argument. So instead, you can define a function in place, without giving it a name. Such a function is called an *anonymous function* (also called a *function literal*).

An anonymous function is defined using the **=>** symbol, often called the "rocket." To the left of the rocket is the argument list, and to the right is a single expression – which can be compound – that produces the function result. We can transition **show** into an anonymous function and hand it directly to **foreach**. First, remove the **def** and the function name:

```
(n:Int) = { println("> " + n) }
```

Now change the **=** to a rocket to tell Scala that it is a function:

```
(n:Int) => { println("> " + n) }
```

This is a legitimate anonymous function (try it in the REPL), but we can simplify it further. If there's only a single expression, Scala allows us to remove the curly braces:

```
(n:Int) => println("> " + n)
```

If there's only a single argument in the argument list and if Scala can infer the type of the argument, we can leave off the parentheses and the argument type:

```
n => println("> " + n)
```

With these simplifications, **DisplayVector.scala** becomes:

```
// DisplayVectorWithAnonymous.scala

val v = Vector(1, 2, 3, 4)
v.foreach(n => println("> " + n))
```

Not only does this produce fewer lines of code, the call becomes a more succinct description of the operation. In addition, type inference allows us to apply the same anonymous function to sequences holding other types:

```
1  // DisplayDuck.scala
2
3  val duck = "Duck".toVector
4  duck.foreach(n => println("> " + n))
```

Line 3 takes the **String** “Duck” and turns it into a **Vector**, with one character in each location. When we pass our anonymous function, Scala infers the function type to the **Char**s in the **Vector**.

Now let’s produce a testable version by storing the results in a **String** rather than sending output to the console. The function passed to **foreach** appends the result to that **String**:

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

var s = ""
val duck = "Duck".toVector
duck.foreach(n => s = s + n + ":")
s is "D:u:c:k:"
```

Testing against the resulting **s** verifies the results.

If you need more than one argument, you must use parentheses for the argument list but you can still take advantage of type inference:

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

val v = Vector(19, 1, 7, 3, 2, 14)
v.sorted is Vector(1, 2, 3, 7, 14, 19)
v.sortWith((i, j) => j < i) is
Vector(19, 14, 7, 3, 2, 1)
```

The default **sorted** method produces the expected ascending order. The **sortWith** method takes a two-argument function and produces a Boolean result indicating whether the first argument is less than the second one; reversing the comparison produces the sorted output in descending order.

A function with zero arguments can also be anonymous. In the following example, we define a class that takes a zero-argument function as an argument, then call that function via the class sometime later. Thus, you can plug in any zero-argument function that you want to call later (you can imagine adding a timer function so you give it a function and say how much later to call it). Pay attention to the type of the class argument, which is declared using anonymous-function syntax – no arguments, a rocket, and **Unit** to indicate that nothing is returned:

```
// CallLater.scala

class Later(val f: () => Unit) {
  def call():Unit = { f() }
}

val cl = new Later(() => println("now"))
cl.call()
```

You can even assign an anonymous function to a **var** or **val**:

```
// AssignAnonymous.scala

val later1 = () => println("now")
var later2 = () => println("now")

later1()
later2()
```

Assigning an anonymous function to a variable name is convenient for testing. You can use an anonymous function anywhere you use a regular function, but if the anonymous function starts getting too complex it's usually better to define a named function, for clarity, even if you're only going to use it once.

Exercises

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

1. Modify **DisplayVectorWithAnonymous.scala** to store results in a **String**, as in **DisplayDuckTestable.scala**. Satisfy the following test:
   ```
   str is "1234"
   ```

2. Working from your solution to the exercise above, add a comma between each number. Satisfy the following test:
   ```
   str is "1,2,3,4,"
   ```

3. Create an anonymous function that calculates age in "dog years" (by multiplying years by 7). Assign it to a **val** and then call your function. Satisfy the following test:
   ```
   val dogYears = // Your function here
   dogYears(10) is 70
   ```

4. Create a **Vector** and append the result of **dogYears** to a **String** for each value in the **Vector**. Satisfy the following test:
   ```
   var s = ""
   val v = Vector(1, 5, 7, 8)
   v.foreach(/* Fill this in */)
   s is "7 35 49 56 "
   ```

5. Repeat Exercise 2 without using the **dogYears** method:
   ```
   var s = ""
   val v = Vector(1, 5, 7, 8)
   v.foreach(/* Fill this in */)
   s is "7 35 49 56 "
   ```

6. Create an anonymous function with three arguments (**temperate**, **low**, and **high**). The anonymous function will return **true** if the temperature is between **high** and **low**, and **false** otherwise. Assign the anonymous function to a **def** and then call your function. Do you think that having three arguments is too many? Satisfy the following tests:
   ```
   between(70, 80, 90) is false
   between(70, 60, 90) is true
   ```

7. Create an anonymous function to square a list of numbers. Call the function for every element in a **Vector**, using **foreach**. Satisfy the following test:
   ```
   var s = ""
   val numbers = Vector(1, 2, 5, 3, 7)
   numbers.foreach(/* Fill this in */)
   s is "1 4 25 9 49 "
   ```

8. Create an anonymous function and assign it to the name **pluralize**. It should construct the (simple) plural form of a word by just adding an “s.” Satisfy the following tests:
   ```
   pluralize("cat") is "cats"
   pluralize("dog") is "dogs"
   pluralize("silly") is "sillys"
   ```

9. Use **pluralize** from the previous exercise. Use **foreach** on a **Vector** of **String**s and print the plural form of each word. Satisfy the following test:
   ```
   var s = ""
   val words = Vector("word", "cat", "animal")
   words.foreach(/* Fill this in */)
   s is "words cats animals "
   ```

map & reduce

In the previous atom you learned about anonymous functions, using **foreach** as an example. Although **foreach** can be useful, it is limited because it can only be used for its side effects: **foreach** doesn't return anything. That's why we used **println** to check solutions.

Methods that return values are often more useful; two good examples are **map** and **reduce**, both of which are available for sequences like **Vector**.

map takes its argument – a function that takes a single argument and produces a result – and applies it to each element in the sequence. This is similar to what we saw with **foreach**, but **map** captures the return value from each call and stores it in a new sequence, which **map** produces as its return value. Here's an example that adds one to each element of a **Vector**:

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

val v = Vector(1, 2, 3, 4)
v.map(n => n + 1) is Vector(2, 3, 4, 5)
```

This uses the more succinct form of the anonymous function that we explored in the previous atom.

Here's one way to add up the values in a sequence:

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

val v = Vector(1, 10 , 100, 1000)
var sum = 0
```

```
v.foreach(x => sum += x)
sum is 1111
```

It's awkward to adapt **foreach** to this purpose; for one thing it requires a **var** to accumulate the sum (there's almost always a way to use **val**s instead of **var**s, and it becomes an intriguing puzzle to try to do so).

reduce uses its argument – in the following example, an anonymous function – to combine all the elements of a sequence. This produces a cleaner way to sum a sequence (notice there are no **var**s):

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

val v = Vector(1, 10, 100, 1000)
v.reduce((sum, n) => sum + n) is 1111
```

reduce first adds 1 and 10 to get 11. That becomes the **sum**, which is added to the 100 to get 111, which becomes the new **sum**. This is added to 1000 to get 1111, which becomes the **sum**. **reduce** then stops because there is nothing else to add, returning the final **sum** of 1111. Of course, Scala doesn't really know it's doing a "sum" – the choice of variable name was ours. We could also have defined the anonymous function with **(x, y)**, but we use a meaningful name to make it easier to understand at a glance.

reduce can perform all sorts of operations on sequences. It's not limited to **Int**s, or to addition:

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

(1 to 100).reduce((sum, n) => sum + n) is
  5050
val v2 = Vector("D", "u", "c", "k")
v2.reduce((sum, n) => sum + n) is "Duck"
```

Line 4 sums the values from 1 to 100 (the famous mathematician Carl Friederich Gauss was said to have done this in his head as an elementary student). Line 7 uses that same anonymous function – along with different type inference – to combine a **Vector** of letters.

Notice that **map** and **reduce** take care of the iteration code that you would normally write by hand. Although managing the iteration yourself might not seem like much effort, it's one more error-prone detail, one more place to make a mistake (and since they're so "obvious," such mistakes are particularly hard to find).

This is one of the hallmarks of *functional programming* (of which **foreach**, **map** and **reduce** are examples): It solves problems in small steps, and the functions will often do things that are seemingly trivial – obviously, it's not that hard to write your own code rather than using **map**, **reduce** and **foreach**. However, once you have a collection of these small, debugged solutions, you can easily combine them without having to debug at each level, and thus create more robust code, more quickly. Scala sequences, for example, come with a fair number of functional programming operations in the same vein as **map**, **reduce** and **foreach**.

Exercises

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

1. Modify **SimpleMap.scala** so the anonymous function multiplies each value by 11 and adds 10. Satisfy the following tests:
```
val v = Vector(1, 2, 3, 4)
v.map(/* Fill this in */) is
  Vector(21, 32, 43, 54)
```

2. Can you replace **map** with **foreach** in the above solution? What happens? Test the result.

3. Rewrite the solution for the previous exercise using **for**. Was this more or less complex than using **map**? Which approach has the greater potential for errors?

4. Rewrite **SimpleMap.scala** using a **for** loop instead of **map**, and observe the additional complexity this introduces.

5. Rewrite **Reduce.scala** using **for** loops.

6. Use **reduce** to implement a method **sumIt** that takes a variable argument list and sums those arguments. Satisfy the following tests:
   ```
   sumIt(1, 2, 3) is 6
   sumIt(45, 45, 45, 60) is 195
   ```

Comprehensions

Now you're ready to learn about a powerful combination of **for** and **if** called the ***for** comprehension*, or just *comprehension*.

In Scala, comprehensions combine *generators*, *filters*, and *definitions*. The **for** loop that you saw in For Loops was a comprehension with a single generator that looks like this:

```
for(n <- v)
```

Each time through the loop, the next element of **v** is placed into **n**. The type of **n** is inferred based on the type contained in the sequence **v**.

Comprehensions can be more complex. In the following example, the method **evenGT5** ("even, greater than 5") takes and returns **Vector**s containing **Int**s. It selects **Int**s from the input **Vector** that satisfy a particular criteria (that is, it *filters* on that criteria) and puts those in the **result Vector**:

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

def evenGT5(v:Vector[Int]):Vector[Int] = {
  // 'var' so we can reassign 'result':
  var result = Vector[Int]()
  for {
    n <- v
    if n > 5
    if n % 2 == 0
  } result = result :+ n
  result
}

```

```
15  val v = Vector(1,2,3,5,6,7,8,10,13,14,17)
16  evenGT5(v) is Vector(6, 8, 10, 14)
```

Notice that **result** is not the usual **val**, but rather a **var**, which means we can modify **result**. Ordinarily, we always try to use a **val** because **val**s can't be changed and that makes our code more self-contained and easily changed. Sometimes, however, you just can't seem to achieve your goal without changing an object. Here, we are building up the **result Vector** by assembling it, so **result** must be changeable. By initializing it with **Vector[Int]()** (you learned about the type parameter **[Int]** in Parameterized Types) we establish the type parameter as **Int** and create an empty **Vector**.

On lines 7 and 11, we use curly braces { } instead of parentheses ( ). This allows us to include multiple statements or expressions in a **for**. While you *can* use parentheses, that requires a discussion about when semicolons are necessary, and we think this is easier. It's also the way most Scala programmers write their comprehensions.

The comprehension begins typically: **n** gets all the values from **v**. But instead of stopping there, we see two **if** expressions. Each of these filters the value of **n** that will make it through the comprehension. First, each **n** that we're looking for must be greater than 5. But an **n** of interest must also satisfy **n % 2 == 0** (since the modulus operator **%** produces the remainder, the expression looks for even numbers).

Next, we want to append all those numbers to **result**. Because **result** is a **var**, we can assign to it. But a **Vector** can't be modified, so how do we "add to" our **Vector**? **Vector** has an operator '**:+**' which creates a new **Vector** by taking an existing one (but *not* changing it) and combining it with the element to the right of the operator. So **result = result :+ n** produces a new **Vector** by appending **n** to the old one, and then assigns this new **Vector** to **result** (in this case the old **Vector** is thrown away and Scala automatically cleans it up). When the **for** loop ends, there's a new **Vector** filled with the desired values.

It turns out there *is* a way to use **val** (instead of **var**) for **result**, and that's to build **result** "in place" rather than creating it piece-by-piece. This effect is produced by Scala's **yield** keyword. When you say **yield n**, it "yields up" the value **n** to become part of **result**. Here's an example:

```
1   // Yielding.scala
2   import com.atomicscala.AtomicTest._
3
4   def yielding(v:Vector[Int]):Vector[Int]={
5     val result = for {
6       n <- v
7       if n < 10
8       if n % 2 != 0
9     } yield n
10    result
11  }
12
13  val v = Vector(1,2,3,5,6,7,8,10,13,14,17)
14  yielding(v) is Vector(1, 3, 5, 7)
```

yield always fills a container. But we haven't declared the type of **result** on line 5, so how does Scala know what kind of container to create? It infers the type from the container that the comprehension is traversing – **v** is a **Vector[Int]**, so **yield** creates a **Vector[Int]** (so the first line of the comprehension determines the type of the result). Now, with a comprehension and **yield**, we can create the entire **Vector** before assigning it to **result**, so **result** can be a **val** instead of a **var**.

The Boolean '**!=**' operator means "not equal" (it produces **true** if the left-hand operand is not equal to the right-hand operand).

You can also *define* values within a comprehension. Here we assign to **isOdd** and then use it to filter the results:

```
// Yielding2.scala
import com.atomicscala.AtomicTest._

def yielding2(v:Vector[Int]):Vector[Int]={
  for {
    n <- v
    if n < 10
    isOdd = (n % 2 != 0)
    if(isOdd)
  } yield n
}

val v = Vector(1,2,3,5,6,7,8,10,13,14,17)
yielding2(v) is Vector(1, 3, 5, 7)
```

Notice that you don't declare **n** or **isOdd** as a **val** or a **var**. Both **n** and **isOdd** change each time through the loop, but you can't manually modify them; instead, you rely on Scala to do it. You can effectively think of them as temporary variables that are set each time through the loop.

This solution also doesn't store and return an intermediate **result** as we did previously. The result of the comprehension is the **Vector** we want to return. Since that expression is the last thing in the method, we can just give the expression.

As with any expression, the **yield** expression can be complex (lines 10-13):

```
1   // Yielding3.scala
2   import com.atomicscala.AtomicTest._
3
4   def yielding3(v:Vector[Int]):Vector[Int]={
5     for {
6       n <- v
7       if n < 10
```

```
      isOdd = (n % 2 != 0)
      if(isOdd)
    } yield {
      val u = n * 10
      u + 2
    }
  }

val v = Vector(1,2,3,5,6,7,8,10,13,14,17)
yielding3(v) is Vector(12, 32, 52, 72)
```

Note that only inside the comprehension can you get away without declaring **val** or **var** for new identifiers.

You can only have one **yield** expression connected with a comprehension, and you cannot place **yield**s in the body of the comprehension. You can, however, nest comprehensions:

```
// Yielding4.scala
import com.atomicscala.AtomicTest._

def yielding4(v:Vector[Int]) = {
  for {
    n <- v
    if n < 10
    isOdd = (n % 2 != 0)
    if(isOdd)
  } yield {
    for(u <- Range(0, n))
      yield u
  }
}

val v = Vector(1,2,3,5,6,7,8,10,13,14,17)
yielding4(v) is Vector(
  Vector(0),
  Vector(0, 1, 2),
  Vector(0, 1, 2, 3, 4),
  Vector(0, 1, 2, 3, 4, 5, 6)
)
```

Here, we let type inference determine the return type of **yielding4**. Each **yield** produces a **Vector**, so the end result is a **Vector** of **Vector**s.

Exercises

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

1. Change **yielding** to a more descriptive name.

2. Modify **yielding2** to accept a **List** instead of a **Vector**. Return a **List**. Satisfy the following test:
   ```
   val theList =
     List(1,2,3,5,6,7,8,10,13,14,17)
   yielding2(theList) is List(1,3,5,7)
   ```

3. Modify **yielding3** so the result is defined with an explicit type declaration. Satisfy the following test:
   ```
   val theList2 =
     List(1,2,3,5,6,7,8,10,13,14,17)
   yielding3(theList2) is Vector(1,3,5,7)
   ```

4. Confirm that you can't change **n** or **isOdd** in **yielding3**. Declare them as **var**s. What happened? Did you find a way to do this? Did it make sense to you?

5. Create a **case class** named **Activity** that contains a **String** for the date (like "01-30") and a **String** for the activity you did that day (like "Bike," "Run," "Ski"). Store your activities in a **Vector**. Create a method **getDates** that returns a **Vector** of **String** corresponding to the days that you did the specified activity. Satisfy the following tests:
   ```
   val activities = Vector(
   ```

```
Activity("01-01", "Run"),
Activity("01-03", "Ski"),
Activity("01-04", "Run"),
Activity("01-10", "Ski"),
Activity("01-03", "Run"))
getDates("Ski", activities) is
  Vector("01-03", "01-10")
getDates("Run", activities) is
  Vector("01-01", "01-04", "01-03")
getDates("Bike", activities) is Vector()
```

6. Building on the previous exercise, create a method **getActivities** that flips things around by returning a **Vector** of **String**s corresponding to the names of the activities that you did on the specified day. Satisfy the following tests:

```
getActivities("01-01", activities) is
  Vector("Run")
getActivities("01-02", activities) is
  Vector()
getActivities("01-03", activities) is
  Vector("Ski", "Run")
getActivities("01-04", activities) is
  Vector("Run")
getActivities("01-10", activities) is
  Vector("Ski")
```

Pattern Matching with Types

You've seen pattern matching with values. You can also match against the *type* of a value. Here's a method that doesn't care about the type of its argument:

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

def acceptAnything(x:Any):String = {
  x match {
    case s:String => "A String: " + s
    case i:Int if(i < 20) =>
      s"An Int Less than 20: $i"
    case i:Int => s"Some Other Int: $i"
    case p:Person => s"A person ${p.name}"
    case _ => "I don't know what that is!"
  }
}
acceptAnything(5) is
  "An Int Less than 20: 5"
acceptAnything(25) is "Some Other Int: 25"
acceptAnything("Some text") is
"A String: Some text"

case class Person(name:String)
val bob = Person("Bob")
acceptAnything(bob) is "A person Bob"
acceptAnything(Vector(1, 2, 5)) is
"I don't know what that is!"
```

The argument type for **acceptAnything** is something you haven't seen before: **Any**. As the name implies, **Any** allows any type of argument. If you want to pass a variety of types to a method and they have nothing in common, **Any** solves the problem.

The **match** expression looks for **String**, **Int**, or our own type **Person** and returns an appropriate message for each. Notice how the value declaration (**s:String**, **i:Int**, and **p:Person**) provides the resulting value in the expression to the right of the **=>**.

Line 7 restricts the match to more than just the type by using an **if** test on the value.

Remember from Pattern Matching that the underscore acts as a wildcard, matching anything without capturing the matched object into a value.

Exercises

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

1. Create a method **plus1** that pluralizes a **String**, adds 1 to an **Int**, and adds "+ guest" to a **Person**. Satisfy the following tests:
   ```
   plus1("car") is "cars"
   plus1(67) is 68
   plus1(Person("Joanna")) is
     "Person(Joanna) + guest"
   ```

2. Create a method **convertToSize** that converts a **String** to its length, uses **Int** and **Double** directly, and converts a **Person** to 1. Return 0 if you don't have a matching type. What was the return type of your method? Satisfy the following tests:
   ```
   convertToSize(45) is 45
   convertToSize("car") is 3
   convertToSize("truck") is 5
   convertToSize(Person("Joanna")) is 1
   ```

```
convertToSize(45.6F) is 45.6F
convertToSize(Vector(1, 2, 3)) is 0
```

3. Modify **convertToSize** from the previous exercise so that it returns an **Int**. Use the **scala.math.round** method to round the **Double** first. Did you need to declare the return type? Do you see an advantage to doing so? Satisfy the following tests:

```
convertToSize2(45) is 45
convertToSize2("car") is 3
convertToSize2("truck") is 5
convertToSize2(Person("Joanna")) is 1
convertToSize2(45.6F) is 46
convertToSize2(Vector(1, 2, 3)) is 0
```

4. Create a new method **quantify** to return "small" if the argument is less than 100, "medium" if the argument is between 100 and 1000, and "large" if the argument is greater than 1000. Support both **Double**s and **Int**s. Satisfy the following tests:

```
quantify(100) is "medium"
quantify(20.56) is "small"
quantify(100000) is "large"
quantify(-15999) is "small"
```

5. In Pattern Matching, you did an exercise that checked the forecast, based on sunniness. We tested using discrete values. Revisit that exercise with ranges of values. Create a method **forecast** that represents the percentage of cloudiness, and use it to produce a "weather forecast" string such as "Sunny" (100), "Mostly Sunny" (80), "Partly Sunny" (50), "Mostly Cloudy" (20), and "Cloudy" (0). Satisfy the following tests:

```
forecast(100) is "Sunny"
forecast(81) is "Sunny"
forecast(80) is "Mostly Sunny"
forecast(51) is "Mostly Sunny"
forecast(50) is "Partly Sunny"
forecast(21) is "Partly Sunny"
forecast(20) is "Mostly Cloudy"
forecast(1) is "Mostly Cloudy"
forecast(0) is "Cloudy"
forecast(-1) is "Unknown"
```

Pattern Matching with Case Classes

Although you've seen that **case** classes are generally useful, they were originally designed for use with pattern matching, and are well suited for that task. When working with **case** classes, a **match** expression can even extract the argument fields.

Here's a description of a trip taken by travelers using various modes of transportation:

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

case class Passenger(
  first:String, last:String)
case class Train(
  travelers:Vector[Passenger],
  line:String)
case class Bus(
  passengers:Vector[Passenger],
  capacity:Int)

def travel(transport:Any):String = {
  transport match {
    case Train(travelers, line) =>
      s"Train line $line $travelers"
    case Bus(travelers, seats) =>
      s"Bus size $seats $travelers"
    case Passenger => "Walking along"
    case what => s"$what is in limbo!"
  }
}

```

```
24 val travelers = Vector(
25   Passenger("Harvey", "Rabbit"),
26   Passenger("Dorothy", "Gale"))
27
28 val trip = Vector(
29   Train(travelers, "Reading"),
30   Bus(travelers, 100))
31
32 travel(trip(0)) is "Train line Reading " +
33   "Vector(Passenger(Harvey,Rabbit), " +
34   "Passenger(Dorothy,Gale))"
35 travel(trip(1)) is "Bus size 100 " +
36   "Vector(Passenger(Harvey,Rabbit), " +
37   "Passenger(Dorothy,Gale))"
```

Line 4 is a **Passenger** class containing the name of the passenger, and lines 6-11 show different modes of transportation along with varying details about each mode. However, all transportation types have something in common: they carry passengers. The simplest "passenger list" we can make is a **Vector[Passenger]**. Notice how easy it is to include this in the **case** class: just put it in the class argument list.

The **travel** method contains a single **match** expression. The argument type for **travel** is **Any**, like the previous atom. We need **Any** in this situation because we want to apply **travel** to all the **case** classes we've defined above, and they have nothing in common.

Line 15 shows how a **case** class can be matched – including the argument(s) used to create the matched object. On line 15, the arguments are named **travelers** and **line**, just like in the class definition, but you can use any names. When a match happens, the identifiers **travelers** and **line** are created and get the values used as the arguments when the **Train** object was created, so they can be used in the expression on the right side of the "rocket" (**=>**) symbol. This is powerful; we can declare any variable in the case expression and use it directly. The types (**Vector[Passenger]** and **String**, in this case) are inferred.

You don't need to match the name in the constructor with the case class arguments. You can see this on line 17. When we defined **Bus** on line 9, we specified the fields as **passengers** and **capacity**. Our pattern match uses **travelers** and **seats**, and the match expression extraction fills those in appropriately, based on the ordering used in the constructor.

You are not forced to unpack the **case** class arguments. Line 19 simply matches the type, without the arguments. But if you choose to unpack it into a value, you can treat it like any other object and access its properties.

Line 20 matches an identifier (**what**) that has no type. This means it matches anything else that the **case** expressions above it missed. The identifier is used as part of the resulting string on the right of the "rocket." If you don't need to use the matched value, you can use the special character '_' as the wildcard identifier.

On line 24 we create a **Vector[Passenger]** and on line 28 we create a **Vector** of the different types of transportation. Each type of transportation carries our travelers and also has details about the transportation.

The point of this example is to show the power of Scala – how easy it is to build a model that represents your system. As you learn, you'll discover that Scala contains numerous ways to keep representations simple, even as your systems get more complex.

Exercises

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

1. Building from **PatternMatchingCaseClasses.scala**, define a new class **Plane** that contains a **Vector** of **Passenger**s and a name for the plane, so you can create a trip. Satisfy the following test:
```
val trip2 = Vector(
  Train(travelers, "Reading"),
  Plane(travelers, "B757"),
  Bus(travelers, 100))
travel(trip2(1)) is "Plane B757 " +
  "Vector(Passenger(Harvey,Rabbit), " +
  "Passenger(Dorothy,Gale))"
```

2. Building on your solution for Exercise 1, change the **case** for **Passenger** so it extracts the object. Satisfy the following test:
```
travel2(Passenger("Sally", "Marie")) is
  "Sally is walking"
```

3. Building on your solution for Exercise 2, determine if you need to make any changes to pass in a **Kitten**. Satisfy the following test:
```
case class Kitten(name:String)
travel2(Kitten("Kitty")) is
  "Kitten(Kitty) is in limbo!"
```

Brevity

Scala is succinct, in contrast with many other languages that require the programmer to write a lot of code (often called “boilerplate” or “jumping through hoops”) to do something simple. Scala can express concepts briefly – sometimes, arguably, *too* briefly. As you learn the language you will understand that, more than anything, this powerful brevity can be responsible for the mistaken idea that Scala is “too complicated.” (We hope you find Scala clear and straightforward during this book).

Until now we’ve used a consistent form for code, without introducing these syntactic shorteners. However, we don’t want you to be too surprised when you see other people’s Scala code, so we are going to show a few of the most useful coding short-forms. This way you’ll start getting comfortable with their existence.

Eliminate Intermediate Results

The last expression in a compound expression becomes the result of that expression. Here’s an example where values are captured into a **val result**, then **result** is returned from the method:

```
// Brevity1.scala
import com.atomicscala.AtomicTest._

def filterWithYield1(
  v:Vector[Int]):Vector[Int] = {
  val result = for {
    n <- v
    if n < 10
    if n % 2 != 0
  } yield n
  result
}

val v = Vector(1,2,3,5,6,7,8,10,13,14,17)
filterWithYield1(v) is Vector(1,3,5,7)
```

Instead of putting it into an intermediate value, the comprehension itself can produce the result:

```
// Brevity2.scala
import com.atomicscala.AtomicTest._

def filterWithYield2(
  v:Vector[Int]):Vector[Int] = {
  for {
    n <- v
    if n < 10
    if n % 2 != 0
  } yield n
}

val v = Vector(1,2,3,5,6,7,8,10,13,14,17)
filterWithYield2(v) is Vector(1,3,5,7)
```

It becomes easier when you remember that everything is an expression, and the final expression becomes the result of the outer expression.

Omit Unnecessary Curly Braces

If a method consists of a single expression, the curly braces around the method are unnecessary. **filterWithYield2** is effectively only one expression, so it doesn't need the surrounding curly braces:

```
// Brevity3.scala
import com.atomicscala.AtomicTest._

```

```
 4  def filterWithYield3(
 5    v:Vector[Int]):Vector[Int] =
 6    for {
 7      n <- v
 8      if n < 10
 9      if n % 2 != 0
10    } yield n
11
12  val v = Vector(1,2,3,5,6,7,8,10,13,14,17)
13  filterWithYield3(v) is Vector(1,3,5,7)
```

Note that this became possible because we had eliminated the intermediate result, producing a single expression.

At first the presence or absence of curly braces can be a little unsettling between one method and the next, but we found that we rapidly became comfortable with it *and* tend to want to eliminate braces whenever possible.

Should You Use Semicolons?

Note that lines 7-9 in the previous example are distinct expressions within the curly braces of the comprehension. In that configuration, the line breaks determine the end of each expression. You can put them all on the same line using semicolons:

```
// Brevity4.scala
import com.atomicscala.AtomicTest._

// Semicolons allow a single-line for:
def filterWithYield4(
  v:Vector[Int]):Vector[Int] =
  for{n <- v; if n < 10; if n % 2 != 0}
    yield n

val v = Vector(1,2,3,5,6,7,8,10,13,14,17)
```

```
filterWithYield4(v) is Vector(1,3,5,7)
```

You can even put the entire method onto a single line (try it). Is that more readable? Or just briefer? We prefer each expression within a comprehension to be on its own line, as in the more straightforward **Brevity3.scala**.

“Do not use semicolons,” said the author Kurt Vonnegut. “They are transvestite hermaphrodites representing absolutely nothing. All they do is show you’ve been to college.” (We are aware that we probably use too many semicolons in this book’s prose).

Remove Unnecessary Arguments

In Functions as Objects, we introduced the **foreach** method to apply an anonymous function to each element of a sequence. Here we pass anonymous functions that call **print** for each letter of a **String** (**foreach** treats the **String** as a sequence and pulls each letter out):

```
1  // Brevity5.scala
2  "OttoBoughtAnAuto".foreach(c => print(c))
3  println
4  "OttoBoughtAnAuto".foreach(print(_))
5  println
6  "OttoBoughtAnAuto".foreach(print)
```

The anonymous function in line 2 already applies some brevity: There’s only one argument in the argument list so we leave off the parentheses.

We can go further by using Scala’s special underscore character on line 4. So far, we’ve only seen the underscore used as a wildcard, but when it’s part of a method call, the underscore means “fill in the blank,” and Scala passes each character without needing a named argument. Since there’s only one argument to **print** and because Scala sees that **print** will accept a **Char**, Scala allows you to take brevity to the extreme and simply pass the method name as the argument to **foreach** without any argument list at all, as on line 6. In general, Scala will do the extra work to construct the proper method call whenever it can, so if you think something might work it's worth experimenting.

The form of line 6 might seem a bit advanced, but it is a commonly used idiom (and arguably more readable than the longer form). If you're coming from a different language it might require some mental shifting, but you'll probably come to appreciate the succinctness.

Use Type Inference for Return Types

Up to this point, we've written out the return type for methods, as on line 4 of the following example. For brevity's sake, we use Scala's type inference and leave off the return type, as seen on line 10:

```
1   // Brevity6.scala
2   import com.atomicscala.AtomicTest._
3
4   def explicitReturnType():Vector[Int] =
5     Vector(11, 22, 99, 34)
6
7   explicitReturnType() is
8     Vector(11, 22, 99, 34)
9
10  def inferredReturnType() =
11    Vector(11, 22, 99, 34)
12
13  inferredReturnType() is
14    Vector(11, 22, 99, 34)
15
16  def unitReturnType() {
17    Vector(11, 22, 99, 34)
18  }
```

```
19
20  unitReturnType() is ()
```

For type inference to work, the '=' sign is still necessary between the method argument list and the method body. If you leave off the '=' as on line 16, Scala will decide that you mean the method returns nothing, which you can also express as **Unit** or '**()**'. Some Scala developers prefer to define the return type for methods, because it makes their intent clear. It also enables the compiler to help detect errors in usage.

Aliasing Names with type

Sometimes when using someone else's code, you might find the names they've chosen to be too long or simply awkward. Scala allows you to alias an existing name to a new name using the **type** keyword:

```
1  // Alias.scala
2  import com.atomicscala.AtomicTest._
3
4  case class LongUnrulyNameFromSomeone()
5  type Short = LongUnrulyNameFromSomeone
6  new Short is LongUnrulyNameFromSomeone()
```

Line 6 shows that **Short** is just another name for **LongUnrulyNameFromSomeone**.

Finding a Balance

Scala will figure out what you mean whenever it can. The safest way to approach Scala brevity is to start by being completely explicit, and then slowly pare down your code. When you go too far, either Scala will produce an error message or you'll get the wrong result. Of course, you need to test everything as you go.

These brevity techniques result in more compact code but might also make it harder to read. You need to make choices depending on who will be reading your code.

Exercises

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

1. Refactor the following example. First, remove the intermediate result. Satisfy the following tests:
   ```
   def assignResult(arg:Boolean):Int = {
     val result = if(arg) 42 else 47
     result
   }
   assignResult(true) is 42
   assignResult(false) is 47
   ```

2. Continue the previous exercise by removing unnecessary curly braces. Satisfy the following tests:
   ```
   assignResult2(true) is 42
   assignResult2(false) is 47
   ```

3. Continue the previous exercise by removing the return type of the method. Note that you had to keep the equals sign. Do you see a downside if you don't declare the return type? Satisfy the following tests:
   ```
   assignResult3(true) is 42
   assignResult3(false) is 47
   ```

4. Refactor **Coffee.scala** from Constructors using the techniques in this atom.

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

Index