ATOMIC SCALA

2ND EDITION

LEARN PROGRAMMING IN A LANGUAGE OF THE FUTURE

BRUCE ECKEL
DIANNE MARSH

ATOMIC SCALA

2ND EDITION

LEARN THE SUCCESSOR TO JAVA TODAY!

- For dedicated beginners (no programming background necessary) with summaries for experienced programmers.
- Explained carefully, Scala is simpler and more sensible than Java — and a lot more powerful.
- Small steps ("atoms") with exercises and solutions for an incremental learning process with many checkpoints.
- Foundations in programming and Scala that don't overwhelm you.
- Our principles: Baby steps and small wins. No forward references. No references to other languages. Show don't tell. Practice before theory.
- Establishes a functioning subset, a "baseline Scala" for your group or company.
- Benefit from the language without "drinking from a firehose."
- Immediately be more productive than with Java.
- After Atomic Scala, you're ready for more advanced books.
- Installation/getting started for Windows, Mac and Linux.
- Example code and exercise solutions freely downloadable at AtomicScala.com
- Live seminars, workshops and consulting available through AtomicScala.com

Cover design by will-harris.com - Illustrations by John Lucas

Bruce Eckel is the author of the multi-award-winning *Thinking in Java* and *Thinking in C++* and other books. He has inhabited the programming planet for over 30 years: writing, giving lectures and seminars, and consulting.

Dianne Marsh has researched and spoken on Scala since 2008. She has programmed professionally since 1987 in languages ranging from C to C#, Python to Java, and finds Scala powerful and fun.

$40

(ROCKET FUEL SOLD SEPARATELY)

WWW.ATOMICSCALA.COM

MINDVIEW LLC - CRESTED BUTTE, CO

ISBN 978-0-9818725-1-3

90000 >

9 780981 872513

Atomic Scala

2nd Edition

Bruce Eckel

Dianne Marsh

MindView LLC, Crested Butte, CO

Contents

How to Use This Book

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

Beginners: Start with the Introduction and move through each chapter (we call chapters *atoms* because they're so small) as you would any other book – including the *Summary* atoms, which solidify your knowledge.

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

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.

Changes in the Second Edition

These are predominantly (many) fixes to the exercises and solutions, corrections from bug reports, and any updates necessary for Scala version 2.11. Some examples are replaced or improved, and a large amount of prose is improved. If you bought the first edition eBook, you automatically get an update to the second edition. Unfortunately, the number of changes to the first edition print book are just too comprehensive to summarize in a document.

Introduction

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

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

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

Beginning programmers should think of it as a game: You'll get through by solving a few puzzles along the way. Experienced programmers can move rapidly through the book and find the place where they must 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 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 the meaning. Indeed, it's often easier to understand a single page of Scala that produces the same effect as many pages of code in another language, because you 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 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

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

Sample the Book

To introduce 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 find the complete electronic version of this book in numerous 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 help someone in any way they need. Perhaps in the future you'll be better off, and you can buy something.

Example Code & Exercise Solutions

These are available for download from **AtomicScala.com**.

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 an 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, contact him at **MindviewInc@gmail.com**.

Conferences

Bruce has organized the *Java Posse Roundup* (which has become the *Winter Tech Forum*: **www.WinterTechForum.com**), 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. It took time and effort to produce this book and accompanying support materials. If you enjoy this book and want to see more things like it, please support us:

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

About Us

Bruce Eckel is the author of the multi-award-winning *Thinking in Java* and *Thinking in C++*, and several other books on computer programming. Living in the computer industry for over 30 years, he 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 Winter Tech Forum* 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 considers these among his stable of life-projects, along with abstract painting. Bruce has a BS in applied physics, and an MS in computer engineering. He studies organizational dynamics, trying to find a new way to organize companies so working together becomes a joy; read about his struggles at **www.reinventing-business.com**, while his programming work is 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 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 fun using Scala. Dianne helped organize CodeMash (**www.codemash.org**), an all-volunteer developer conference bringing together programmers of various languages to learn from each other, and was 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 specifically thank Steve Harley, Alf Kristian Stoyle, Andrew Harmel-Law, Al Gorup, Joel Neely, and James Ward, all of whom were 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 grew weary of passive voice and punctuation errors. And special thanks go to her husband, Tom Sosnowski, for his tolerance and encouragement throughout this process.

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

Dedication

To Julianna and Benjamin Sosnowski. You are amazing.

Copyrights

Editors

To install Scala, you might need to make changes to your system configuration files. To do this you need a program called an *editor*. You 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. 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 handy editing features that you'll probably come to love. More details are on their site.

Although Sublime Text is commercial software, you can freely use it for as long as you like (you 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; these 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 – everything was text-based.

Although it can seem primitive in the age of graphical user interfaces, there are still a surprising number of valuable things to accomplish with a shell, and we use it regularly, 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 opens a shell in the destination directory. (If Powershell doesn't start, go to the Microsoft website and install it from there).

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. Think of a directory as 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 essential operations in a shell, ones we 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** displays all the files and subdirectory names in the current directory. 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 to 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 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 the graphical file browser (Windows Explorer, the Mac Finder, or Nautilus or equivalent on Linux) will 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)

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). In this book we use JDK8 (Java 1.8).

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

```
java version "1.8.0_11"
Java(TM) SE Runtime Environment (build 1.8.0_25-b18)
Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed
mode)
```

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

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 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" and press "Return" to open the System application. Look for "System Type," which 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 attempts to detect whether to 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*. The path tells the operating system where to find executable programs. For example, something like this goes at the end of the path:

```
;C:\Program Files\Java\jre8\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, press Windows+W, then type **env** in the edit box, and 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 2.11, the latest available at the time. In general, the code in this book should also work on versions more recent than 2.11.

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 might 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     api
```

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 places 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 **atomic-scala-examples-master.zip** and choose "Extract all …" then choose the default destination folder. Once everything is extracted, move into the destination folder and navigate down until you find the **examples** directory.

Move to the **C:** directory and create the **C:\AtomicScala** directory. Either copy or drag the **examples** directory into the **C:\AtomicScala** directory. Now the **AtomicScala** directory contains all the examples from the book.

Set Your CLASSPATH

To run the examples, you must first set your *CLASSPATH*, an *environment variable* 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\examples
```

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\examples** 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**, to test all the code in the book using Windows. Before you run the script for the first time, you must tell Powershell that it's OK. In addition to setting the Execution Policy as described in The Shell, you must unblock the script. Using the Windows Explorer, go to the **C:\AtomicScala\examples** directory. Right click on **testall.ps1**, choose "Properties" and then check "Unblock."

Running **./testall.ps1** tests all the code examples from the book. You 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 2.11 or greater.

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

Installation (Mac)

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. In this book we use JDK8 (Java 1.8).

Follow the instructions in The Shell to open a shell in the desired directory. 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 2.11, the latest available at the time. In general, the code in this book should also work on versions more recent than 2.11.

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, 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 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 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 can 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 **atomic-scala-examples-master.zip** into a convenient location on your computer.

Unpack the book's source code by double clicking on **atomic-scala-examples-master.zip**. Navigate down into the resulting unpacked folder until you find the **examples** directory. Create an **AtomicScala** directory in your home directory, and drag **examples** into the **AtomicScala** directory, using the directions above (for installing Scala).

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

Set Your CLASSPATH

The *CLASSPATH* is an *environment variable* 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="$HOME/AtomicScala/examples:${CLASSPATH}"
export CLASSPATH
```

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

```
cd ~/AtomicScala/examples
```

Now run:

```
scalac AtomicTest.scala
```

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

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

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

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

You 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 2.11 or greater.
3. Quit the REPL by typing **:quit**.

Installation (Linux)

In this book, we use Scala version 2.11, the latest available at the time. In general, the examples in this book should also work on versions more recent than 2.11.

Standard Package Installation

Important: The standard package installer might 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 also installs 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 might 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 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**, 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 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 is a **bin** directory. Edit your **.profile** (following the instructions earlier in this atom) and locate the last **PATH** directive, if there is one. Add or modify your **PATH** so 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 can 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, Mac OSX, Cygwin." The file has 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 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 can look like this:

```
export set
PATH=/home/`whoami`/jre1.7.0_09/bin:/home/`whoami`/scala-
2.11.4/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 **atomic-scala-examples-master.zip** to your home directory using the shell command:

```
cp atomic-scala-examples-master.zip ~
```

Unpack the book's source code by running **unzip atomic-scala-examples-master.zip**. Navigate down into the resulting unpacked folder until you find the **examples** directory.

Create an **AtomicScala** directory in your home directory, and move **examples** into the **AtomicScala** directory. The **~/AtomicScala** directory now contains all the examples from the book in the subdirectory **examples**.

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* 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/examples:$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/examples** subdirectory. Then run:

```
scalac AtomicTest.scala
```

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

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

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

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

You 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 2.11 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 2.11.4 (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 2.11 or greater.

The REPL gives you immediate interactive feedback, which is helpful for experimentation. 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.

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 (which we call *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 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 runs 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, enter:

```
scala myfile.scala
```

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

Scripting makes it easy to quickly create simple programs, so we use it throughout much of this book (thus, you run the examples via The Shell). Scripting solves basic problems, such as making utilities for your computer. More sophisticated programs require the *compiler*, which we 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 can 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 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:

```
1  // Values.scala
2
3  val whole = 11
4  val fractional = 1.4
5  val words = "A value"
6
7  println(whole, fractional, words)
8
9  /* Output:
10 (11,1.4,A value)
11 */
```

The first line of each example in this book contains the name of the source code file as you find it in the **AtomicScala** directory that you set up in your appropriate "Installation" atom. You 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 so it fits on an eBook reader page, so we sometimes add line breaks – to shorten the lines – where they would not otherwise be necessary.

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* or *immutable*). 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 your code easier to understand 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 **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 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 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 get a fractional number, but you probably don't think about it much. Scala categorizes these different ways of representing data into 'types' so it knows if you're using the right kind of data. Here, 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 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 explicitly specify the type, you tell Scala that **n** is an **Int** and **p** is a **Double**, rather than letting it infer the type.

Here are Scala's basic types:

```
1  // Types.scala
2
3  val whole:Int = 11
4  val fractional:Double = 1.4
5  // true or false:
6  val trueOrFalse:Boolean = true
7  val words:String = "A value"
8  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 see this on line 3. To hold fractional numbers, as on line 4, use a **Double**.

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

A **String** holds a sequence of characters. You assign a value using a double-quoted string as on line 7, or if you have many lines and/or special characters, you 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** is a **Double** and ensures that it follows all the rules for **Double**s.

Scala does a lot of type inference for you, as part of its strategy of doing work for the programmer. If you leave out the type declaration, Scala will usually 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 **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, use a *variable* instead of a value.

Like a value, a *variable* holds a particular type of information. But with a variable, you can change the stored data. You define a variable in exactly the same way you define a value, except 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 in the book, we conserve space by omitting the space.

Exercises

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

1. Create an **Int** value (**val**) and set it 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** and set it to **5**. Update it to **10** and store in a **val** named **constantv1**. Did this work? Can you think of how this is useful?

3. Using **v1** and **constantv1** from above, 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*. These have one simple difference.

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). As a memory aid:

> *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**. The same result comes from 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 **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.
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 if 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 simple conditional that uses the **>** (greater than) sign and shows Scala's **if** keyword:

```
1  // If.scala
2
3  if(1 > 0) {
4    println("It's true!")
5  }
6
7  /* Output:
8  It's true!
9  */
```

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

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

```
1  // If2.scala
2
3  val x:Boolean = { 1 > 0 }
4
5  if(x) {
6    println("It's true!")
7  }
8
9  /* Output:
10 It's true!
11 */
```

Because **x** is **Boolean**, the **if** can test it directly by saying **if(x)**.

You test for the opposite of the Boolean expression using the "not" operator '**!**':

```
1  // If3.scala
2
3  val y:Boolean = { 11 > 12 }
4
5  if(!y) {
6    println("It's false")
7  }
8
9  /* Output:
10 It's false
11 */
```

By putting the "not" operator in front, **if(!y)** reads "if not y."

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

```
1  // If4.scala
2
3  val z:Boolean = false
4
5  if(z) {
6    println("It's true!")
7  } else {
8    println("It's false")
9  }
10
11 /* Output:
12 It's false
13 */
```

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

The entire **if** is an expression, so it can produce a result:

```
// If5.scala

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

/* Output:
42
*/
```

You will learn more about conditionals in later atoms.

Exercises

Solutions are available at **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. Here's an example of evaluation order which mixes arithmetic operations:

```
45 + 5 * 6
```

The multiplication operation 5 * 6 is performed first, followed by the addition 30 + 45 to produce 75.

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

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

For a result of 300.

As another 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 higher is overweight.

```
1  // BMI.scala
2
3  val kg = 72.57 // 160 lbs
4  val heightM = 1.727 // 68 inches
5
6  val bmi = kg/(heightM * heightM)
7  if(bmi < 18.5) println("Underweight")
8  else if(bmi < 25) println("Normal weight")
9  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.

Here's another case where different evaluation order produces different results:

```
1  // EvaluationOrder.scala
2
3  val sunny = true
4  val hoursSleep = 6
5  val exercise = false
6  val temp = 55
7
8  val happy1 = sunny && temp > 50 ||
9    exercise && hoursSleep > 7
10 println(happy1) // true
11
12 val sameHappy1 = (sunny && temp > 50) ||
13   (exercise && hoursSleep > 7)
14 println(sameHappy1) // true
15
16 val notSame =
17   (sunny && temp > 50 || exercise) &&
18   hoursSleep > 7
19 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. Here, 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 (without parentheses, the "ands" are evaluated first, then the "or"). Lines 16-18 use parentheses to produce a different result; in that expression we're only happy if we get at least 7 hours of sleep.

When you're not sure what evaluation order Scala will choose, use parentheses to force your intention. This also makes it clear to anyone who reads 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*, meaning "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, 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 **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 can contain one line of code, or multiple lines of code surrounded with curly braces. Now we 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 compound expression:

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

Notice that **a** is defined inside the compound expression. The result of the last expression becomes the result of the compound expression; here, 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 that determines if a business is open or closed, based on the **hour**:

```
1  // CompoundExpressions1.scala
2
3  val hour = 6
4
5  val isOpen = {
6    val opens = 9
7    val closes = 20
8    println("Operating hours: " +
9      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. Line 10 checks whether **hour** is between opening time and closing time, so we combine the expressions with the Boolean **&&** (and).

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

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

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

The compound expression from **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**.

Expressions that produce results simplify code:

```
1  // CompoundExpressions3.scala
2  val activity = "swimming"
3  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      false
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 **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 if you want to do each activity based on both temperature and if a facility is open. Print the results of the comparisons to match the output described below. 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 to do an activity. For example, 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, go back and study the earlier 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 name that 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 a **val**, you can modify a **var**, so the following is valid:

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

However, the type can't be changed, so you 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*. These have one simple difference:

> *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 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, so 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

If you're not sure what order Scala will evaluate expressions, use parentheses to force your intention. This also makes it clear to anyone who reads 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 produces 3.4.

Compound Expressions

Compound expressions are contained within curly braces. A compound expression holds 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)
```

A value defined inside an expression, such as **bmi** on line 5, 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 the result of its last expression; here, the **String** "Normal weight."

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

Exercises

Solutions are available at **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 produce **Unit** instead of **String**.
10. Modify **CompoundBMI.scala** to produce 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 (sometimes 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
}
```

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 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:

```
1  // MultiplyByTwo.scala
2
3  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 captures the result into the value **r**. 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 (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 the third argument, a **String**, and returns a **Double** value, the result of the calculation on line 6.

Line 13 begins another method, only defined 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 if the two don't agree.

The **assert** on line 16 is a method defined by Scala. It takes a Boolean expression and a **String** message (which we build 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 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.

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

In other Scala code, you'll see many ways to write methods in addition to the form shown in this atom. Scala is 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 use this form and introduce the others after you're more comfortable with Scala.

Exercises

Solutions are available at **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? Use 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 hold data and perform actions. An object-oriented language is oriented towards creating and using 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 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*. 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 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, you get a list of options. We can find the possible operations on any class this way (the REPL will give lots of information – 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 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, look them up in the Scala documentation, which we introduce in the following atom.

A warning: although the REPL is a 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 don'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 perform operations on them. Instead of "performing an operation," we sometimes say *sending a message* or *calling a method*. To perform an operation on an object, give 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 call it by saying **r.reverse**, which 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 **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** by 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 shows you the available operations for a class, ScalaDoc provides much more detail. It's helpful to keep a window open with the REPL running for quick experiments when you have a question, and a second window containing the documentation.

The documentation can be installed on your machine (see below), or 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 see several items that contain the word "Range." Click on **Range**; the right-hand window will 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 helpful to get used to the Scala documentation so you become comfortable looking things up.

If the Scala installation process you used didn't give you the option to install the documentation locally, download it from **www.scala-lang.org** by selecting 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 Scala classes are actually Java classes, and they were dropped from the ScalaDoc as of 2.8. **String** is an example of a Java class we often use in this book, which 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, 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) different about a species of animal is captured in the classification of that animal's object. Scala provides the **class** keyword to create new types of objects:

```
1  // Animals.scala
2
3  // Create some classes:
4  class Giraffe
5  class Bear
6  class Hippo
7
8  // Create some objects:
9  val g1 = new Giraffe
10 val g2 = new Giraffe
11 val b = new Bear
12 val h = new Hippo
13
14 // Each 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*. When 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$$anon1** 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 it's explained in Wikipedia) is the address where the object is located in your computer's memory.

The classes defined here (**Giraffe**, **Bear**, and **Hippo**) are as simple as possible: 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 showing an object is being created:

```
1  // Hyena.scala
2
3  class Hyena {
4    println("This is in the class body")
5  }
6  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 **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

A method defined within a class belongs to that class. Here, the **bark** method belongs to the **Dog** class:

```
1  // Dog.scala
2  class Dog {
3    def bark():String = { "yip!" }
4  }
```

Methods are called (invoked) with the object name, followed by a '.' (dot/period), followed by the method name and argument list. 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:

```
1  // Hamster.scala
2  class Hamster {
3    def speak():String = { "squeak!" }
4    def exercise():String = {
```

```
5       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 **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

A fundamental principle in programming is the acronym DRY: *Don't Repeat Yourself*. Duplicating code is not just extra work. You also create multiple identical pieces of code that you must change whenever you make fixes or improvements. 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 a class name:

```
import packagename.classname
```

A package is an associated collection of code; each package is usually designed to solve a particular problem, and often contains multiple classes. For example, Scala's standard **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. To import more than one class, we use multiple **import** statements:

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

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

Here, we import more than one item within the same **import** statement:

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

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

Here, we combine 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 from a package, use the underscore:

```
1  // ImportEverything.scala
2  import util._
3
4  val r = new Random
5  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:

```
1  // FullyQualify.scala
2
3  val r = new util.Random
4  val p = util.Properties
```

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

```
1  // PythagoreanTheorem.scala
2  package pythagorean
3
4  class RightTriangle {
5    def hypotenuse(a:Double, b:Double):Double={
6      Math.sqrt(a*a + b*b)
7    }
8    def area(a:Double, b:Double):Double = {
9      a*b/2
10   }
11 }
```

On line 2, we name the package **pythagorean**, and then define the class **RightTriangle** 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 PythagoreanTheorem.scala
```

Packages cannot be scripts – they can only be compiled.

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

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

```
1  // ImportPythagorean.scala
2  import pythagorean.RightTriangle
3
4  val rt = new RightTriangle
5  println(rt.hypotenuse(3,4))
6  println(rt.area(3,4))
7  assert(rt.hypotenuse(3,4) == 5)
8  assert(rt.area(3,4) == 6)
```

Run the script as usual with:

```
scala ImportPythagorean.scala
```

You need '.' In your CLASSPATH for this to work. A bug in Scala 2.11 and below causes a delay between compiling and making the classes available for import. To get around this bug, use **nocompdaemon**:

```
scala -nocompdaemon ImportPythagorean.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.pythagorean** rather than just **pythagorean**. This helps us avoid name collisions with other libraries that might also use the name **pythagorean**.

Exercises

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

1. Rename the package **pythagorean** 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 **ImportPythagorean.scala**, above, and save as **Solution-1.scala**. Remember to update the package import to use your new class. Ensure that the tests run properly.

2. Add another class **EquilateralTriangle** to your solution for Exercise 1. Create a method **area** with an argument **side** (as a **Double**); look up the formula in Wikipedia. Display a test result and use **assert** to verify it.

3. Modify **ImportPythagorean.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

Robust code must be tested constantly – every time you make changes. This way, if you change one part of your code that unexpectedly affects other code, 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, changes accumulate and you don't know which one caused the problem – you 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 become accustomed to testing as a standard part of the programming process.

Using **println** to verify code correctness is a weak approach; you must pay attention to the output every time and consciously ensure that it's right. Using **assert** is better because it happens automatically. However, a failed **assert** produces 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:

- Shows the expected result of expressions right next to those expressions, for easier comprehension.
- Shows some output so you see that 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, so reach for that when you start producing real Scala code.

Here, our testing framework is 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 running a Scala script that uses **AtomicTest**, you must follow the instructions in your appropriate "Installation" atom to compile the **AtomicTest** object (or 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. The code is in Appendix A.

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 the **is** method:

```
expression is expected
```

You see this used on lines 8-10 in the previous example.

This system is 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 compared directly (without converting them first). In either case, *expression* is displayed on the console so you 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 12-16 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 see an example of failure 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 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 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 just display 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. 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. There's 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).

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

```
3
4   calculateBMI(160, 68) is "Normal weight"
5   calculateBMI(100, 68) is "Underweight"
6   calculateBMI(200, 68) is "Overweight"
7
8   def calculateBMI(lbs: Int,
9     height: Int):String = { "Normal weight" }
```

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

```
1   // TDDStillFails.scala
2   import com.atomicscala.AtomicTest._
3
4   calculateBMI(160, 68) is "Normal weight"
5   calculateBMI(100, 68) is "Underweight"
6   calculateBMI(200, 68) is "Overweight"
7
8   def calculateBMI(lbs:Int,
9     height:Int):String = {
10    val bmi = lbs / (height*height) * 703.07
11    if (bmi < 18.5) "Underweight"
12    else if (bmi < 25) "Normal weight"
13    else "Overweight"
14  }
```

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:

```
1   // TDDWorks.scala
2   import com.atomicscala.AtomicTest._
3
4   calculateBMI(160, 68) is "Normal weight"
5   calculateBMI(100, 68) is "Underweight"
6   calculateBMI(200, 68) is "Overweight"
7
8   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 **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 using 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 **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. 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 to perform operations using each value. You start with the keyword **for**, followed by a parenthesized expression that traverses the sequence. 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 that we append to a **var String** called **result** (using the '**+=**' operator) 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 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), use:

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

or

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

The first form makes the meaning more explicit.

Note the 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 modify **sum** each time through the loop.

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

Exercises

Solutions are available at **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 if 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 to add 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 stating the **Vector** name and handing it 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")
```

Here's something different: notice that all the **Vector** objects are created without using the **new** keyword. For convenience, Scala allows you to build a class that can be instantiated without **new**, and **Vector** is such a class. In fact, you *can't* create a **Vector** object using the **new** keyword – try it and see what error message you get, so you recognize when it happens with other classes. You'll eventually learn how to do this with your own classes, but for now it's enough to know that some library classes behave this way.

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 initialization order, and you select them individually by number. Like most programming languages, Scala starts 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, notice that **for** loops work well with **Vector**s: **for(i <- v1)** means "**i** gets each value in **v1**." This is more help from Scala: 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 it for you. Many other programming languages will force you to do 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 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 "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. Calling it "sort" implies that the original **Vector** is changed directly (a.k.a. *sorted in place*). Throughout Scala, you 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.

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 be "reversed."

Exercises

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

1. Use the REPL to create several **Vector**s, each populated by a different type of data. See 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 can 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**. Now 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 **replace** method 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.

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.

More Conditionals

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

```
1  // TrueOrFalse.scala
2  import com.atomicscala.AtomicTest._
3
4  def trueOrFalse(exp:Boolean):String = {
5    if(exp) {
6      return "It's true!" // Need 'return'
7    }
8    "It's false"
9  }
10
11 val b = 1
12 trueOrFalse(b < 3) is "It's true!"
13 trueOrFalse(b > 3) is "It's false"
```

The 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 passed to the method. Here, **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 often. If we give the **String** "It's true" without the **return**, nothing happens; the method continues 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 – 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. Test multiple combinations by combining **else** and **if**:

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

```
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 **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 **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 is 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 is your next atom after Summary 1.

Beginning programmers should read this atom and perform the exercises as review. If any information here isn't clear to you, 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 start by introducing packages and imports so 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:

```
1  // ALibrary.scala
2  package com.yoururl.libraryname
3  // Components to reuse ...
4  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 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 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.

Write an **import** statement to use a library:

```
1  // UseALibrary.scala
2  import com.yoururl.libraryname._
3  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 2.11 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 UseALibrary.scala
```

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

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

```
4   val pi = 3.14
5   val pie = "A round dessert"
6
7   pi is 3.14
8   pie is "A round dessert"
9   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, defined in the same way as a type for a **var** or **val**: a colon followed by the type name. A method's type is the type of the returned result.

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**:

```
1  // BasicMethods.scala
2  import com.atomicscala.AtomicTest._
3
4  def cube(x:Int):Int = { x * x * x }
5  cube(3) is 27
6
7  def bang(s:String):String = { s + "!" }
8  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*. 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, notice that Scala uses parentheses for indexing into sequences (indexing is zero-based) rather than square brackets as in many languages.

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, 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 when possible.

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 executed during the creation of each object

This example shows 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 line 7 and a **Vector** holding **SomeBody** objects on line 13. Fields with fixed contents are of limited use; things will get more interesting later.

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

Here's a class with methods:

```
1  // Temperature.scala
2  import com.atomicscala.AtomicTest._
3
4  class Temperature {
5    var current = 0.0
6    var scale = "f"
7    def setFahrenheit(now:Double):Unit = {
8      current = now
9      scale = "f"
10   }
11   def setCelsius(now:Double):Unit = {
12     current = now
13     scale = "c"
14   }
15   def getFahrenheit():Double = {
16     if(scale == "f")
17       current
18     else
19       current * 9.0/5.0 + 32.0
20   }
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
```

```
temp.getCelsius is 37.0
temp.setCelsius(100.0)
temp.getFahrenheit is 212.0
```

These methods are just like those we've defined *outside* of classes, except 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 prevents the reference **temp** from being reassigned to a new object; it does not restrict the behavior of the object itself.

Notice on lines 31, 32 and 34 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._

class Cell {
  var entry = ' '
  def set(e:Char):String = {
    if(entry==' ' && (e=='X' || e=='O')) {
      entry = e
      "successful move"
    } else
      "invalid move"
  }
}

```

```
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 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 to 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**:

```
1  // ForVector.scala
2  val v = Vector("Somewhere", "over",
3    "the", "rainbow")
4  for(word <- v) {
5    println(word)
6  }
```

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 **word** isn't declared 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:

```
1  // RangeShorthand.scala
2  import com.atomicscala.AtomicTest._
3
4  var result = ""
5  for(i <- 0 until 10) {
6    result += i + " "
7  }
8  result is "0 1 2 3 4 5 6 7 8 9 "
9
10 result = ""
11 for(i <- 0 to 10) {
12   result += i + " "
13 }
14 result is "0 1 2 3 4 5 6 7 8 9 10 "
15
16 result = ""
17 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 **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 if 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 makes comparisons and takes 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.

```
1  // MatchExpressions.scala
2  import com.atomicscala.AtomicTest._
3
4  def matchColor(color:String):String = {
5    color match {
6      case "red" => "RED"
7      case "blue" => "BLUE"
8      case "green" => "GREEN"
9      case _ => "UNKNOWN COLOR: " + color
10   }
11 }
12
13 matchColor("white") is
14   "UNKNOWN COLOR: white"
15 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. Lines 6-8 compare the value **color** to **"red"**, **"blue"** and **"green"**. The first successful match finishes the execution of the pattern match – here, a **String** is produced by the pattern match which becomes the return value of **matchColor**.

Line 9 is another special use of "_" (underscore). Here, it is a *wildcard*, and matches anything not 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 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.

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 **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. 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 typically want to initialize it by passing some information. 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.

Note that **a** is not accessible outside the class body, as shown by the error that 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 (See lines 12-14).

Note that **ca2** is a **val** (lines 7). Does the fact that you 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:

```
1  // MultipleClassArgs.scala
2  import com.atomicscala.AtomicTest._
3
4  class Sum3(a1:Int, a2:Int, a3:Int) {
5    def result():Int = { a1 + a2 + a3 }
6  }
7
```

```
8  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 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 **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? Should you 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 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 be 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:

```
1  // NamedAndDefaultArgs.scala
2
3  class Color2(red:Int = 100,
4    blue:Int = 100, green:Int = 100)
5  new Color2(20)
6  new Color2(20, 17)
7  new Color2(blue = 20)
8  new Color2(red = 11, green = 42)
```

Any argument you don't specify gets its default value, so you need only provide 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:

```
1  // Family.scala
2
3  class Family(mom:String, dad:String,
4    kids:String*)
5
6  new Family(mom="Mom", dad="Dad")
7  // Doesn't work:
8  // new Family(dad="Dad", mom="Mom")
9
10 new Family(mom="Mom", dad="Dad",
11   kids="Sammy", "Bobby")
12 // Doesn't work:
13 /* new Family(dad="Dad", mom="Mom",
14   kids="Sammy", "Bobby") */
```

Ordinarily, named arguments allow us to change the order of the parents, so we can 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 recommend that you avoid using named arguments with variable argument lists.

Exercises

Solutions are available at **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 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 test:
```
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 test:
```
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(taxRate = 0.06) is 530
```

Overloading

The term *overload* refers to the name of a method: You 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**. The method's *signature* consists of the name, argument list and return type. Scala distinguishes one method from another by comparing signatures. The only difference between the signatures on lines 5 and 6 is the argument list, and that's all Scala needs 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* to produce unique names (you can also create unique names using random characters but the typical pattern is to use 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 just say *what*, which raises the level of

abstraction and puts less mental load on the reader. If you want to know *how*, 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 slight 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 **AtomicScala.com**.

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

2. Create five overloaded methods that sum their arguments. 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 a class.

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

Initialization is a significant stumbling block. Your code might 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 the arguments 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 can 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 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 **AtomicScala.com**.

1. Modify **Coffee.scala** to 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 here 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 important act of initialization, constructor overloading has an additional constraint: all auxiliary constructors must first call the *primary constructor*. This 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:

```
1  // GardenGnome.scala
2  import com.atomicscala.AtomicTest._
3
4  class GardenGnome(val height:Double,
5    val weight:Double, val happy:Boolean) {
6    println("Inside primary constructor")
7    var painted = true
8    def magic(level:Int):String = {
9      "Poof! " + level
10   }
11   def this(height:Double) {
12     this(height, 100.0, true)
13   }
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 if 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 must overload a constructor.

Exercises

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

1. Create a class called **ClothesWasher** with a default constructor and two auxiliary constructors, one that specifies **model** (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 you 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 (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 **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 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 & 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 zero. 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 & 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 classes that primarily hold data. 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 **val**s. 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.

When your class is basically a data-holder, case classes simplify your code and perform common work.

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

```
1  // CaseClasses.scala
2  import com.atomicscala.AtomicTest._
3
4  case class Dog(name:String)
5  val dog1 = Dog("Henry")
6  val dog2 = Dog("Cleo")
7  val dogs = Vector(dog1, dog2)
8  dogs is Vector(Dog("Henry"), Dog("Cleo"))
9
10 case class Cat(name:String, age:Int)
11 val cats =
12   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 see this where the **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 **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

With *string interpolation*, you create strings containing formatted values. You 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"
```

Notice that any identifier preceded by a '**$**' is converted to string form.

You can evaluate and convert an expression by placing it inside '**${}**' (this is especially 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 quotes on the argument of the **Sky** constructor.

Exercises

Solutions are available at **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 consider it a good idea to let Scala infer types whenever possible. It tends to make the code cleaner and easier to read. Sometimes, however, Scala can't figure out what type to use (it complains), so we must help. For example, we must occasionally tell Scala the type contained in a **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
```

The initialization values tell Scala that the **Vector** on line 5 contains **Int**s and the **Vector** on line 6 contains **String**s.

To show you how it looks, we 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, **Vector[Int]**. The square brackets are new here; they denote a *type parameter*. Here, the container holds objects of the type parameter. 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 elements other than containers, but you usually see them with container-like objects, and in this book we normally use **Vector** as a 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 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 **AtomicScala.com**.

1. Modify **explicit** in **ParameterizedReturnTypes.scala** so it creates and returns a **Vector** of **Double**. Satisfy the following test:
```
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**. 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

It's possible to 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, **foreach** is a helpful method available for sequences like **Vector**. It 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*.

Functions you pass as arguments to other methods or functions are often quite small, and it's 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. This is called an *anonymous function* or a *function literal*.

An anonymous function is defined using the **=>** symbol, often called "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 it's 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 and if Scala can infer the type of that argument, we can leave off the parentheses and the argument type:

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

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

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

Not only does this produce fewer lines of code, the call becomes a 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**.

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**:

```
1   // DisplayDuckTestable.scala
2   import com.atomicscala.AtomicTest._
3
4   var s = ""
5   val duck = "Duck".toVector
6   duck.foreach(n => s = s + n + ":")
7   s is "D:u:c:k:"
```

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

```
1   // TwoArgAnonymous.scala
2   import com.atomicscala.AtomicTest._
3
4   val v = Vector(19, 1, 7, 3, 2, 14)
5   v.sorted is Vector(1, 2, 3, 7, 14, 19)
6   v.sortWith((i, j) => j < i) is
7   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 calls that function sometime later. Pay attention to the type of the class argument, declared using anonymous-function syntax – no arguments, a rocket, and **Unit** to indicate that nothing is returned:

```
1  // CallLater.scala
2
3  class Later(val f: () => Unit) {
4    def call():Unit = { f() }
5  }
6
7  val cl = new Later(() => println("now"))
8  cl.call()
```

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

```
1  // AssignAnonymous.scala
2
3  val later1 = () => println("now")
4  var later2 = () => println("now")
5
6  later1()
7  later2()
```

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 **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 4 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 (**temperature**, **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. 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 work with 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**:

```
1  // SimpleMap.scala
2  import com.atomicscala.AtomicTest._
3
4  val v = Vector(1, 2, 3, 4)
5  v.map(n => n + 1) is Vector(2, 3, 4, 5)
```

This uses the succinct form of the anonymous function, as we explored in the previous atom.

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

```
1  // Sum.scala
2  import com.atomicscala.AtomicTest._
3
4  val v = Vector(1, 10 , 100, 1000)
5  var sum = 0
6  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 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 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 **map**, **reduce** and **foreach** are examples): It solves problems in small steps, and the functions often do things that seem 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 this way 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 **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 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 the sequence **v** is placed in **n**. The type of **n** is inferred based on the type contained in **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 criterion (that is, it *filters* on that criterion) and puts those in the **result Vector**:

```
1  // Comprehension.scala
2  import com.atomicscala.AtomicTest._
3
4  def evenGT5(v:Vector[Int]):Vector[Int] = {
5    // 'var' so we can reassign 'result':
6    var result = Vector[Int]()
7    for {
8      n <- v
9      if n > 5
10     if n % 2 == 0
11   } result = result :+ n
12   result
13 }
14
```

```
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**, so 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 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 the **for** to include multiple statements or expressions. 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 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** (the modulus operator **%** produces the remainder, so 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** (here, 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.

There *is* a way to use **val** (instead of **var**) for **result**: by building **result** "in place" rather than creating it piece-by-piece. To achieve this, use 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 the comprehension traverses – **v** is a **Vector[Int]**, so **yield** creates a **Vector[Int]** (the first line of the comprehension determines the type of the result). Now, with a comprehension and **yield**, we 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:

```
1  // Yielding2.scala
2  import com.atomicscala.AtomicTest._
3
4  def yielding2(v:Vector[Int]):Vector[Int]={
5    for {
6      n <- v
7      if n < 10
8      isOdd = (n % 2 != 0)
9      if(isOdd)
10   } yield n
11 }
12
13 val v = Vector(1,2,3,5,6,7,8,10,13,14,17)
14 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. 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 just give the expression.

As with any expression, the **yield** expression can be compound (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
8      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),
```

```
20    Vector(0, 1, 2, 3, 4),
21    Vector(0, 1, 2, 3, 4, 5, 6)
22  )
```

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 **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. Start with **yielding3** and rewrite the comprehension so it is as compact as possible (reduce **isOdd** and the **yield** clause). Now assign the comprehension to an explicitly-typed value called **result**, and return **result** at the end of the method. Continue to satisfy the existing tests in **Yielding3.scala**.

4. Confirm that you can't modify **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 for the expression to the right of the **=>**.

Line 7 restricts the match beyond 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 **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 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. Pattern Matching included an exercise to check 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 class 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 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 a **case** class being 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 arguments values from 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 declare any variable in the case expression and use it directly. The types (**Vector[Passenger]** and **String**, in this case) are inferred.

The name in the constructor doesn't have to match the case class arguments (line 17). When we define **Bus** on line 9, we specify 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 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 the matched value, 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 **AtomicScala.com**.

1. Building from **PatternMatchingCaseClasses.scala**, define a new class **Plane** containing 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 must 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

Many languages require the programmer to write a lot of code to do something simple. This is often called "boilerplate" or "jumping through hoops."

Scala can express concepts briefly – sometimes, arguably, *too* briefly. As you learn the language you will understand that this powerful brevity can produce the impression that Scala is "too complicated."

Until now we've used a consistent form for code, without introducing 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 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:

```
1  // Brevity1.scala
2  import com.atomicscala.AtomicTest._
3
4  def filterWithYield1(
5    v:Vector[Int]):Vector[Int] = {
6    val result = for {
7      n <- v
8      if n < 10
9      if n % 2 != 0
10   } yield n
11   result
12 }
```

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

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

```
1  // Brevity2.scala
2  import com.atomicscala.AtomicTest._
3
4  def filterWithYield2(
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
13 val v = Vector(1,2,3,5,6,7,8,10,13,14,17)
14 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:

```
1  // Brevity3.scala
2  import com.atomicscala.AtomicTest._
3
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 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:

```
1   // Brevity4.scala
2   import com.atomicscala.AtomicTest._
3
4   // Semicolons allow a single-line for:
5   def filterWithYield4(
6     v:Vector[Int]):Vector[Int] =
7     for{n <- v; if n < 10; if n % 2 != 0}
8       yield n
9
10  val v = Vector(1,2,3,5,6,7,8,10,13,14,17)
11  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 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, and only one expression in the function so we leave off curly braces.

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 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 can 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 can 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 '()'. (The extra parentheses are required to make it explicit to **AtomicTest**). 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

When using someone else's code, you might find the names they've chosen to be too long or 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 get the wrong result. Of course, you must test everything as you go.

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

Exercises

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

1. Refactor the following example. First, remove the intermediate result and satisfy the 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.

A Bit of Style

Most programming languages develop style guides as they mature. In some cases, these guides tell you how to format the code on the page for greater readability. Fortunately, code-formatting style for Scala was established from the inception of the language (and, in some cases, is enforced by the language syntax). Most editing tools that support Scala automatically format your code as you create it, so you don't need to think too hard about that.

The Scala Style Guide, at **docs.scala-lang.org/style**, includes useful information about generally accepted style in Scala. In this book we break some of those rules, in particular those that relate to adding spaces for readability, due to page width restrictions in the book. As you develop experience with Scala you will find useful tidbits in the Scala Style Guide.

One important guideline involves parentheses on methods that take no arguments.

```
1  // MethodParentheses.scala
2  import com.atomicscala.AtomicTest._
3
4  class Simple(val s:String) {
5    def getA() = s
6    def getB = s
7  }
8
9  val simple = new Simple("Hi")
10 simple.getA() is "Hi"
11 simple.getA is "Hi"
12 simple.getB is "Hi"
13 // simple.getB() is "Hi" // Rejected
```

Neither method **getA** nor **getB** takes arguments. Both methods are as succinct as possible: single expressions (requiring no curly braces) that return the value of **s**. Here, we leave off the return type, taking advantage of Scala's ability to infer that each produces a **String**.

A method without arguments can leave off the parentheses in the definition, as shown in **getB** on line 6. In the test code on lines 10 and 11, notice that even though **getA** is defined with parentheses, it can be called with or without them. However, because **getB** is defined without parentheses, it can *only* be called without parentheses.

Here's the style question: Since Scala is flexible about the way you call a method that doesn't have arguments, does it matter? Yes: Parentheses have stylistic meaning in the Scala community. If a method modifies the internal state of the object – if internal variables change when you call the method – then leave the parentheses *on* in the method definition. This signals the reader that this is a *mutating* method (it causes the object to change). Ideally, when you call the method you also include the parentheses to send the same message (although you've seen that Scala doesn't require it).

On the other hand, if calling the method produces a result without changing the state of the object, the convention is to leave the parentheses *off* the method definition, telling the reader that this method reads data without mutating the object. Since both methods return the stored value of **s**, **getB** is the preferred form.

Why is it preferable to leave the parentheses off methods that don't change an object? Programmers who call **getB** should not have to care whether **getB** is a field (**val**) or a method (**def**). The caller only cares that **getB** produces the desired value, not how it happens (this is the *Uniform Access Principle*).

Exercises

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

1. Create a class **Exclaim** with a class argument **var s:String**. Create methods **parens** and **noParens** that append an exclamation point to **s** and return it. Satisfy the following tests:
   ```
   val e = new Exclaim("yes")
   e.noParens is "yes!"
   e.parens() is "yes!"
   ```

2. Building on Exercise 1, change **noParens** to be a field (**val**) instead of a method. Satisfy the following tests:
   ```
   val e2 = new Exclaim2("yes")
   e2.noParens is "yes!"
   e2.parens() is "yes!"
   ```

3. Refactor your solution to Exercise 1, renaming the class **Exclaim3**. Remove the method that doesn't match the conventional style for parentheses in Scala.

4. Add the variable **count** to the class in the previous exercise. Increment **count** when someone calls the method that adds an exclamation point. Call that method twice and satisfy the following test:
   ```
   val e4 = new Exclaim4("counting")
   // Call exclamation method
   // Call exclamation method again
   e4.count is 2
   ```

Idiomatic Scala

Native speakers of spoken languages use *idioms*: expressions that other native speakers not only understand, but have come to expect. To keep things easy, we didn't burden you with stylistic recommendations earlier in this book. Now that you've seen the Brevity and A Bit of Style atoms, let's rework some earlier exercises to conform more closely to the expectations of the Scala community.

Exercises

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

Work the following exercises to reflect the Brevity and A Bit of Style guidelines, as well as other constructs that you have learned up to this point.

1. Refactor **If4.scala** and **If5.scala** from Conditional Expressions.
2. Refactor **For.scala** from For Loops.
3. Refactor **CompoundExpressions2.scala** from Compound Expressions.
4. Refactor **AddMultiply.scala** from Methods. Remove the return type of the method.
5. Refactor **CheckTruth.scala** from More Conditionals.
6. Refactor **Dog.scala**, **Cat.scala** and **Hamster.scala** from Methods Inside Classes.
7. Refactor **ClassArg.scala** and **VariableClassArgs.scala** from Class Arguments.

Defining Operators

Method names can contain almost any characters. For example, when creating a math package you can define the Greek letter sigma the same way mathematicians do: to sum a series. Or you might find it useful to create a meaning for the '+' operator. Scala treats sigma and '+' just like any other characters that can be used in a method name, like **f** or **g** or **plus**:

```
1  // Molecule.scala
2  class Molecule {
3    var attached:Molecule = _
4    def plus(other:Molecule) =
5      attached = other
6    def +(other:Molecule) =
7      attached = other
8  }
9
10 var m1 = new Molecule
11 var m2 = new Molecule
12 m1.plus(m2)
13 m1.+(m2)
14 // Infix calls:
15 m1 plus m2
16 m1 + m2
```

This class models something called a **Molecule** that attaches to another object of its own kind. The **attached** field on line 3 connects one **Molecule** to another, and must be initialized to keep Scala from complaining. Here, we invoke yet another meaning for Scala's special "blank" character, the underscore. When used in an initialization expression, it means "default initialization value."

Notice that the methods defined on lines 4-5 and 6-7 are identical except for the name of the method: in one case the name is **plus**, in the other it is **+**. Scala treats the two methods equally. On lines 12 and 13, you see the ordinary "dot notation" method calls, but both methods can also be called using *infix notation*, placing the method name between the objects as on lines 15 and 16. Line 16 happens to read like a familiar math expression, but it's no different than using **plus** on line 12. (Infix notation enables **AtomicTest** to support its "*object* **is** *expression*" syntax).

Some languages include *operator overloading*, which means a select group of characters is set aside for special parsing and behavior. Instead of that, Scala makes all characters equal and treats all methods the same – if they happen to look like operators, that's your perception. Scala therefore doesn't provide operator overloading, choosing instead the more elegant approach.

Because characters are treated equally in method names, you can easily create incomprehensible code (the import statement removes a warning):

```
1  // Swearing.scala
2  import language.postfixOps
3
4  class Swearing {
5    def #!>% = "Rowzafrazaca!"
6  }
7  val x = new Swearing
8  println(x.#!>%)
9  println(x #!>%)
```

Scala accepts this code, but what does it mean to the reader? Because code is read much more than it is written, you should make your programs as understandable as possible. Unfortunately, even some standard Scala libraries have violated this principle, and this helps

produce accusations that Scala is too complicated and obtuse. Because the language doesn't remove programming power in order to protect you from yourself, it is indeed possible to create complicated and obtuse code. You can also create elegant and transparent code.

Languages work just fine without overloading or the ability to invent your own operators. These aren't essential features, but are excellent examples of how a language is more than just a way to manipulate the underlying computer. That's the easy part. The hard part is crafting the language to provide better ways to express your abstractions, so humans have an easier time understanding the code without getting bogged down in needless detail. It's possible to define operators in ways that obscure meaning, so tread carefully.

"Everything is syntactic sugar. Toilet paper is syntactic sugar, and I still want it." – Barry Hawkins

Exercises

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

1. In Exercise 4 of Class Exercises, you created a class **SimpleTime** with a **subtract** method. Change the name of that method to use the minus sign (**-**). Satisfy the following tests:

```
val someT1 = new SimpleTime2(10, 30)
val someT2 = new SimpleTime2(9, 30)
val someST = someT1 - someT2
someST.hours is 1
someST.minutes is 0
val someST2 = new SimpleTime2(10, 30) -
  new SimpleTime2(9, 45)
someST2.hours is 0
someST2.minutes is 45
```

2. Create a class **FancyNumber1** that takes an **Int** as a class parameter and has one method, **power(n: Int)** that raises that

number to the **n**th power. Hint: you may choose to use **scala.math.pow**, and if you do, investigate **toInt** and **toDouble**. Satisfy the following tests:

```
val a1 = new FancyNumber1(2)
a1.power(3) is 8
val b1 = new FancyNumber1(10)
b1.power(2) is 100
```

3. Adding to your solution for the previous exercise, replace **power** with **^**. Satisfy the following tests:

```
val a2 = new FancyNumber2(2)
a2.^(3) is 8
val b2 = new FancyNumber2(10)
b2 ^ 2 is 100
```

4. Building on the previous exercise, add another method ****** which does the same thing as **^**. Do you see a benefit to leaving in the method **power** and calling that from both methods? Satisfy the following tests:

```
val a3 = new FancyNumber3(2.0)
a3.**(3) is 8
val b3 = new FancyNumber3(10.0)
b3 ** 2 is 100
```

Automatic String Conversion

case classes nicely format an object for display, including its arguments (line 6):

```
1  // Bicycle.scala
2  import com.atomicscala.AtomicTest._
3
4  case class Bicycle(riders:Int)
5  val forTwo = Bicycle(2)
6  forTwo is "Bicycle(2)" // Nice
```

A method called **toString** is automatically defined when you create a **case class**. Whenever you do anything with an object that expects a **String**, Scala silently produces a **String** representation for the object by calling **toString**.

If you create a regular, non-**case** class, you still get an automatic **toString**:

```
1  // Surrey.scala
2  class Surrey(val adornment:String)
3  val fancy = new Surrey("fringe on top")
4  println(fancy) // Ugly
```

This is the default **toString** and isn't too useful; when you run **Surrey.scala** you get output that looks something like **Main$$anon1Surrey@7b2884e0**. For better results, define your own **toString**:

```
1  // SurreyWithToString.scala
2  import com.atomicscala.AtomicTest._
3
4  class Surrey2(val adornment:String) {
5    override def toString =
6      s"Surrey with the $adornment"
7  }
8
9  val fancy2 = new Surrey2("fringe on top")
10 fancy2 is "Surrey with the fringe on top"
```

Line 5 introduces a new keyword: **override**. This is necessary – Scala insists on it – because **toString** is already defined (the definition that produces that ugly result). The **override** keyword tells Scala that yes, we do actually want to replace it with our own definition; this explicitness makes it clear to the reader of the code what is happening and helps prevent mistakes. Note that we use the brief syntax forms here: no parentheses (because this method doesn't change the object), return type inference, and a single-line method.

A good **toString** is useful when debugging a program; sometimes just looking inside an object is enough to see what's going wrong.

Exercises

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

1. Override **toString** in a **case class**. Modify **Bicycle** so its **toString** produces "Bicycle built for 2." Satisfy the following test:
   ```
   val forTwo = Bicycle(2)
   forTwo is "Bicycle built for 2"
   ```
2. Build on the previous exercise to show that the **toString** method can be more complex than a single-line method.
   A) Change the class name to **Cycle** and pass the number of wheels as a class argument when you create the object.

B) Use pattern matching to display “Unicycle” for a single wheeled cycle, “Bicycle” for 2 wheels, “Tricycle” for 3 wheels, “Quadricycle” for 4 wheels, and “Cycle with n wheels” for numbers greater than 4, replacing “n” with the argument. Satisfy the following tests:

```
val c1 = Cycle(1)
c1 is "Unicycle"
val c2 = Cycle(2)
c2 is "Bicycle"
val cn = Cycle(5)
cn is "Cycle with 5 wheels"
```

3. Add to the previous exercise. For a negative number of wheels, satisfy the following test:

```
Cycle(-2) is "That's not a cycle!"
```

Tuples

Suppose you must return more than one item from a method, perhaps a value and some information about that value. A perfectly legitimate approach is to create a special class to hold the return value:

```
1  // ReturnBlob.scala
2  import com.atomicscala.AtomicTest._
3
4  case class
5    ReturnBlob(data:Double, info:String)
6
7  def data(input:Double) =
8    if(input > 5.0)
9      ReturnBlob(input * 2, "High")
10   else
11     ReturnBlob(input * 2, "Low")
12
13 data(7.0) is ReturnBlob(14.0, "High")
14 data(4.0) is ReturnBlob(8.0, "Low")
```

Many programming language designers consider this an adequate solution, but this is a case where little things make a big difference: even though returning multiple values is a helpful technique, in languages that do not support *tuples*, you don't see it used that often.

A tuple allows you to collect multiple elements, effortlessly. It's like **ReturnBlob**, but Scala automates everything for you. You create a tuple by grouping elements together inside parentheses:

(element1, element2, element3, ...)

We can rewrite the above example using tuples:

```
1  // Tuples.scala
```

```
2  import com.atomicscala.AtomicTest._
3
4  def data2(input:Double):(Double, String) =
5    if(input > 5.0)
6      (input * 2, "High")
7    else
8      (input * 2, "Low")
9
10 data2(7.0) is (14.0, "High")
11 data2(4.0) is (8.0, "Low")
12
13 def data3(input:Double) =
14   if(input > 5.0)
15     (input * 2, "High", true)
16   else
17     (input * 2, "Low", false)
18
19 data3(7.0) is (14.0, "High", true)
20 data3(4.0) is (8.0, "Low", false)
```

Line 4 specifies the return type; A tuple return type simply means surrounding your types with parentheses. Line 13 uses type inference for the return tuple. Note that lines 6, 8, 15 and 16 return tuples by surrounding the values with parentheses. Lines 15 and 16 show that returning additional elements is easy. Tuples make grouping elements so trivial that it becomes an effortless choice; indeed, before programmers know about tuples they tend to only think in terms of returning a single thing, and afterwards returning multiple elements becomes a natural approach – the existence of the feature changes the way you program.

If you have a tuple and want to capture the values, use tuple unpacking as on line 6:

```
1   // TupleUnpacking.scala
2   import com.atomicscala.AtomicTest._
3
4   def f = (1,3.14,"Mouse",false,"Altitude")
5
6   val (n, d, a, b, h) = f
7
8   (a, b, n, d, h) is
9     ("Mouse", false, 1, 3.14, "Altitude")
10
11  // Tuple indexing:
12  val all = f
13  f._1 is 1
14  f._2 is 3.14
15  f._3 is "Mouse"
16  f._4 is false
17  f._5 is "Altitude"
```

On line 6, a single **val** followed by a tuple of identifiers unpacks the tuple returned by **f**. On line 8 we even write the test with a tuple on the left of the **is** (moving some elements to make it interesting).

If instead you capture the entire tuple into a single **val** or **var** as on line 12, you can select each element by indexing with "**._n**" (lines 13-17; note that we start counting from one and not zero).

There's a similar form to unpack case classes. On line 6, the case class almost behaves like a tuple with a class name attached:

```
1   // CaseUnpack.scala
2   import com.atomicscala.AtomicTest._
3
4   case class Employee(name:String, ID:Int)
5   val empA = Employee("Bob", 1130)
6   val Employee(nm, id) = empA
7   nm is "Bob"
```

```
8   id is 1130
```

You can even combine initializations using tuples (whether this makes your code more readable depends on context):

```
scala> var (d, n, s) = (1.1, 12, "Hi")
d: Double = 1.1
n: Int = 12
s: String = Hi
```

Without tuples, methods that group elements together become awkward. With tuples, collecting elements in groups becomes effortless enough that it tends to produce better code.

Exercises

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

1. Unpack the values from the tuples below into named variables for **temp**, **sky**, and **view**. Satisfy the following tests:

```
val tuple1 = (65, "Sunny", "Stars")
val (/* fill this in */) = tuple1
temp1 is 65
sky1 is "Sunny"
view1 is "Stars"

val tuple2 =
  (78, "Cloudy", "Satellites")
val (/* fill this in */) = tuple2
temp2 is 78
ski2 is "Cloudy"
view2 is "Satellites"

val tuple3 = (51, "Blue", "Night")
val (/* fill this in */) = tuple3
temp3 is 51
```

```
ski3 is "Blue"
view3 is "Night"
```

2. Create a tuple to hold the values 50 and 45. Unpack the values using numeric indices. Satisfy the following tests:
```
val info = // fill this in
info./* what goes here? */ is 50
info./* what goes here? */ is 45
```

3. Create a method **weather** that takes arguments for **temperature** and **humidity**. Your method will return "Hot" if the temp is above 80 degrees and "Cold" if the temperature is below 50 degrees. Otherwise, return "Temperate." Your method will also return "Humid" if humidity is above 40%, unless the temperature is below 50. In that case, it should return "Damp." Otherwise, return "Pleasant." Write tests for the above conditions, and also satisfy the following tests:
```
weather(81, 45) is ("Hot", "Humid")
weather(50, 45) is ("Temperate", "Humid")
```

4. Using your solution for the previous exercise, unpack the values into **heat** and **moisture**. Satisfy the following tests:
```
val (/* fill this in */) = weather(81, 45)
heat1 is "Hot"
moisture1 is "Humid"
val (/* fill this in */) = weather(27, 55)
heat2 is "Cold"
moisture2 is "Damp"
```

Companion Objects

Methods act on particular instances of a class:

```
1  // ObjectsAndMethods.scala
2  import com.atomicscala.AtomicTest._
3
4  class X(val n:Int) {
5    def f = n * 10
6  }
7
8  val x1 = new X(1)
9  val x2 = new X(2)
10
11 x1.f is 10
12 x2.f is 20
```

When you call **f**, you must call it with an object. During the call, **f** can access the members of that object, without qualification (on line 5, we just say **n**, without specifying the object).

Scala keeps track of the object of interest by quietly passing around a reference to that object. That reference is available as the keyword **this**. You can access **this** explicitly, but most of the time it isn't necessary. This example is exactly the same as the previous one, except **this** is added to line 5:

```
1  // ThisKeyword.scala
2  import com.atomicscala.AtomicTest._
3
4  class X(val n:Int) {
5    def f = this.n * 10
6  }
7
8  val x1 = new X(1)
```

```
val x2 = new X(2)

x1.f is 10
x2.f is 20
```

Note how the **n** that's part of **x1** is distinguished from the **n** that's part of **x2**. Scala does this for you, under the covers.

Some methods aren't "about" a particular object, so it doesn't make sense to tie them to an object. You could argue that, in this case, you should just make an ordinary method (some languages work this way), but it's more expressive if you also say, "this method or field is about the class, but not about a particular object."

Scala's **object** keyword defines something that looks roughly like a class, except you can't create instances of an **object** – there's only one. An **object** is a way to collect methods and fields that logically belong together but don't need multiple instances. Thus, you never create any instances – there's only one instance and it's "just there."

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

object X {
  val n = 2
  def f = n * 10
  def g = this.n * 20
}

X.n is 2
X.f is 20
X.g is 40
```

You can't say **new X**. If you try, Scala complains that there is no "type X." That's because the **object** declaration sets up the structure and creates the object at the same time.

When using **object**, the naming convention is slightly different. Typically, when we create an instance of a class using the **new** keyword, we lower-case the first letter of the instance name. When you define an **object**, however, Scala defines the class *and* creates a single instance of that class. Thus, we capitalize the first letter of the **object** name because it represents both a class and an instance.

Notice on line 7 that the **this** keyword still works, but it just refers to that one object instance rather than multiple possible instances.

The **object** keyword allows you to create a *companion object* for a class. The only difference between an ordinary **object** and a companion **object** is the latter has the same name as the name of a regular class. This creates an association between the companion **object** and its class:

```
1  // CompanionObject.scala
2  class X
3  object X
```

This makes **object X** the companion object of **class X**.

If you create a field inside a companion object, it produces a single piece of data for that field no matter how many instances of the associated class you make:

```
1  // ObjectField.scala
2  import com.atomicscala.AtomicTest._
3
4  class X {
5    def increment() =  { X.n += 1; X.n }
6  }
7
8  object X {
9    var n:Int = 0 // Only one of these
```

```
10  }
11
12  var a = new X
13  var b = new X
14  a.increment() is 1
15  b.increment() is 2
16  a.increment() is 3
```

Lines 14-16 show that **n** has only a single piece of storage (no matter how many instances are created) and that **a** and **b** are both accessing that same memory. To access elements of the companion object from methods of the class, you must give the name of the companion object, as on line 5.

When a method is *only* accessing fields in the companion object, it makes sense to move that method into the companion object:

```
1   // ObjectMethods.scala
2   import com.atomicscala.AtomicTest._
3
4   class X
5
6   object X {
7     var n:Int = 0
8     def increment() =  { n += 1; n }
9     def count() = increment()
10  }
11
12  X.increment() is 1
13  X.increment() is 2
14  X.count() is 3
```

On line 8 we no longer need to qualify access for **n** because the method is now in the same scope as **n**. Line 9 shows that companion object methods can call other companion object methods without qualification.

Here's a helpful use for companion objects: Count each instance and include the count when displaying the object. This gives each object a unique identifier:

```
1  // ObjCounter.scala
2  import com.atomicscala.AtomicTest._
3
4  class Count() {
5    val id = Count.id()
6    override def toString = s"Count$id"
7  }
8
9  object Count {
10   var n = -1
11   def id() = { n += 1; n }
12 }
13
14 Vector(new Count, new Count, new Count,
15   new Count, new Count) is
16 "Vector(Count0, Count1, " +
17 "Count2, Count3, Count4)"
```

By initializing **n** to -1, the first call to **id** produces zero. This code also works with **case** classes – try adding the **case** keyword on line 4.

Companion objects enable some pleasant syntax sugar. You've already seen one of the more common of these: when you create a **case** class, you don't have to use **new** to create an instance of that class:

```
scala> case class Car(make:String)
defined class Car

scala> Car("Toyota")
res1: Car = Car(Toyota)
```

This happens because creating a **case** class automatically creates a companion object containing a special method **apply**, called a *factory method* because it creates other objects. When you give the companion object's name followed by parentheses (with arguments as appropriate), Scala calls **apply**. Here, we write a factory method for a non-**case**-class:

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

class Car(val make:String) {
  override def toString = s"Car($make)"
}

object Car {
  def apply(make:String) = new Car(make)
}

val myCar = Car("Toyota")
myCar is "Car(Toyota)"
```

Our **toString** produces nice output, just as the **case class** does.

Companion objects have several other bits of syntax sugar you can learn about elsewhere. Companion objects are not strictly necessary (imagine using ordinary methods and **var**s/**val**s), but they provide improvements in organization and syntax that makes code easier to understand.

When viewing the ScalaDoc, you switch between class view and companion-object view by clicking on the graphic "O" or "C" that's in the upper-left region of the documentation pages (you can see whether this switching is possible for a particular class if it looks like the "O" or "C" has a little peeled-back corner).

Exercises

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

1. Create a class **WalkActivity** that takes no class arguments. Create a companion object with a single method **start** that has a single argument for a name and prints "started!" Demonstrate how to call this method. Did you have to instantiate the **WalkActivity** object?

2. Building on your solution for the previous exercise, add a field to the companion object to log activities (Hint: Use a **var String**). Calling **start("Sally")** should append "[Sally] Activity started." Also, add a **stop** method that similarly appends "[Sally] Activity stopped."

3. Add a field for *Metabolic Equivalent of Task* (MET) initialized to 2.3. Add the supplied method **calories**. Where did you put the field? Where did you put the method? If you didn't put them in the companion object, do so now. Did you have to make any changes to do so? Satisfy the following tests:

```
def calories(lbs:Int, mins:Int,
  mph:Double=3):Long = math.round(
    (MET * 3.5 * lbs * 0.45)/200.0 * mins
  )
val sally = new WalkActivity3
sally.calories(150, 30) is 82
```

4. Vary the *Metabolic Equivalent of Task* based on speed of walking. Add the following **MET** method. Validate the method with tests. Did you put it in the class or the companion object? Update your **calories** method to call **MET(mph).** Satisfy the following tests:

```
def MET(mph: Double) = mph match {
  case x if(x < 1.7) => 2.3
  case x if(x < 2.5) => 2.9
  case x if(x < 3) => 3.3
  case x if(x >= 3) => 3.3
```

```
  case _ => 2.3
}
WalkActivity4.MET(1.0) is 2.3
WalkActivity4.MET(2.7) is 3.3
val suzie = new WalkActivity4
suzie.calories(150, 30) is 117
val john = new WalkActivity4
john.calories(150, 30, 1.5) is 82
```

Inheritance

Objects store data in fields and perform actions via operations (typically called methods). Each object occupies a unique place in storage so one object's fields can have different values from every other object.

An object also belongs to a category called a class, which determines the form or template for its objects: the fields and the methods. Thus, all objects look like the class that created them (via its constructor).

Creating and debugging a class can require a lot of work. What if you want to make a class like an existing class, but with some variations? It seems like a pity to build a new class from scratch, so object-oriented languages provide a mechanism for reuse called *inheritance*.

With inheritance (following the concept of biological inheritance), you say, "I want to make a new class from an existing class, but with some additions and modifications." You inherit a new class based on an existing class using the **extends** keyword on lines 9-11:

```
1  // GreatApe.scala
2  import com.atomicscala.AtomicTest._
3
4  class GreatApe {
5    val weight = 100.0
6    val age = 12
7  }
8
9  class Bonobo extends GreatApe
10 class Chimpanzee extends GreatApe
11 class BonoboB extends Bonobo
12
13 def display(ape:GreatApe) =
14   s"weight: ${ape.weight} age: ${ape.age}"
```

```
15 display(new GreatApe) is
16 "weight: 100.0 age: 12"
17 display(new Bonobo) is
18 "weight: 100.0 age: 12"
19 display(new Chimpanzee) is
20 "weight: 100.0 age: 12"
21 display(new BonoboB) is
22 "weight: 100.0 age: 12"
```

The terms *base class* and *derived class* (or *parent class* and *child class*, or *superclass* and *subclass*) are often used to describe the inheritance relationship. Here, **GreatApe** is the base class, and it looks a bit strange for reasons you'll understand in the next atom: it has two fields with fixed values. The derived classes **Bonobo**, **Chimpanzee** and **BonoboB** are new types that are identical to their parent class.

The **display** method on line 13 takes a **GreatApe** as an argument, and naturally you call it with a **GreatApe** as on line 15. But see on lines 17 and 19 that you can also call **display** with a **Bonobo** or a **Chimpanzee**! Even though the latter two are distinct types, Scala happily accepts them as if they were the *same type* as **GreatApe**. This works at any level of inheritance, as you see on line 21 (**BonoboB** is two inheritance levels away from **GreatApe**).

This works because inheritance guarantees that anything that inherits from **GreatApe** *is* a **GreatApe**. All code that acts upon objects of these derived classes knows that **GreatApe** is at their core, so any methods and fields in **GreatApe** will also be available in its children.

Inheritance enables you to write a single piece of code (the **display** method) that works not just with one type, but with that type and every class that inherits that type. Thus, inheritance creates opportunities for code simplification and reuse.

This example is a bit *too* simple because all the classes are exactly identical. It only gets interesting when child classes can differentiate themselves from their parents. First, however, we must learn about object initialization during inheritance.

Exercises

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

1. Add a method **vocalize** to **GreatApe**. Satisfy the following tests:
```
val ape1 = new GreatApe
ape1.vocalize is "Grrr!"
val ape2 = new Bonobo
ape2.vocalize is "Grrr!"
val ape3 = new Chimpanzee
ape3.vocalize is "Grrr!"
```

2. Building on the previous exercise, create a method **says** that takes a **GreatApe** argument and calls **vocalize**. Satisfy the following tests:
```
says(new GreatApe) is "says Grrr!"
says(new Bonobo) is "says Grrr!"
says(new Chimpanzee) is "says Grrr!"
says(new BonoboB) is "says Grrr!"
```

3. Create a class **Cycle** that has a field for **wheels** set to 2, and a method **ride** that returns "Riding." Create a derived class **Bicycle** that inherits from **Cycle**. Satisfy the following tests:
```
val c = new Cycle
c.ride is "Riding"
val b = new Bicycle
b.ride is "Riding"
b.wheels is 2
```

Base Class Initialization

Scala guarantees correct object creation by ensuring that all constructors are called: not just the constructors for the derived-class parts of the object, but also the constructor for the base class. In our Inheritance example, the base class didn't have constructor arguments. If a base class does have constructor arguments, then any class that inherits from that base must provide those arguments during construction.

Let's rewrite **GreatApe** in a more sensible fashion, using constructor arguments:

```
1  // GreatApe2.scala
2  import com.atomicscala.AtomicTest._
3
4  class GreatApe(
5    val weight:Double, val age:Int)
6
7  class Bonobo(weight:Double, age:Int)
8    extends GreatApe(weight, age)
9  class Chimpanzee(weight:Double, age:Int)
10   extends GreatApe(weight, age)
11 class BonoboB(weight:Double, age:Int)
12   extends Bonobo(weight, age)
13
14 def display(ape:GreatApe) =
15   s"weight: ${ape.weight} age: ${ape.age}"
16
17 display(new GreatApe(100, 12)) is
18 "weight: 100.0 age: 12"
19 display(new Bonobo(100, 12)) is
20 "weight: 100.0 age: 12"
21 display(new Chimpanzee(100, 12)) is
22 "weight: 100.0 age: 12"
```

```
23 display(new BonoboB(100, 12)) is
24 "weight: 100.0 age: 12"
```

When we inherit from **GreatApe**, Scala forces us to pass the constructor arguments to the **GreatApe** base class (otherwise you get an error message). You typically produce those arguments by creating an argument list for the derived class, as on lines 7, 9, and 11. Then you use those arguments when calling the base-class constructor.

After Scala creates memory for your object, it calls the base-class constructor first, then the constructor for the next-derived class, and so on until it reaches the most-derived constructor. This way, all the constructor calls can rely on the validity of all the sub-objects created before them. Indeed, those are the only things it knows about; a **Bonobo** knows that it inherits from **GreatApe** and the **Bonobo** constructor can call methods in the **GreatApe** class, but a **GreatApe** cannot know whether it's a **Bonobo** or a **Chimpanzee** or call methods specific to those subclasses.

When you inherit, the derived-class constructor must call the primary base-class constructor; if there are auxiliary (overloaded) constructors in the base class you may optionally call one of those instead. The derived-class constructor must pass the appropriate arguments to the base-class constructor:

```
1  // AuxiliaryInitialization.scala
2  import com.atomicscala.AtomicTest._
3
4  class House(val address:String,
5    val state:String, val zip:String) {
6    def this(state:String, zip:String) =
7      this("address?", state, zip)
8    def this(zip:String) =
9      this("address?", "state?", zip)
10 }
```

```
11
12 class Home(address:String, state:String,
13   zip:String, val name:String)
14   extends House(address, state, zip) {
15     override def toString =
16       s"$name: $address, $state $zip"
17 }
18
19 class VacationHouse(
20   state:String, zip:String,
21   val startMonth:Int, val endMonth:Int)
22   extends House(state, zip)
23
24 class TreeHouse(
25   val name:String, zip:String)
26   extends House(zip)
27
28 val h = new Home("888 N. Main St.", "KS",
29   "66632", "Metropolis")
30 h.address is "888 N. Main St."
31 h.state is "KS"
32 h.name is "Metropolis"
33 h is
34 "Metropolis: 888 N. Main St., KS 66632"
35
36 val v =
37   new VacationHouse("KS", "66632", 6, 8)
38 v.state is "KS"
39 v.startMonth is 6
40 v.endMonth is 8
41
42 val tree = new TreeHouse("Oak", "48104")
43 tree.name is "Oak"
44 tree.zip is "48104"
```

When **Home** inherits from **House** it passes the appropriate arguments to the primary **House** constructor. Notice that it also adds its own **val**

argument – you aren't limited to the number, type or order of the arguments in the base class. Your only responsibility is to provide the correct base-class arguments.

In a derived class, you call any of the overloaded base-class constructors via the derived-class primary constructor by providing the necessary constructor arguments in the base-class constructor call. You see this in the definitions of **Home**, **VacationHouse** and **TreeHouse**. Each uses a different base-class constructor.

You can't call base-class constructors inside of overloaded derived-class constructors. As before, the primary constructor is the "gateway" for all the overloaded constructors.

Inheritance from case classes is limited, as seen in Exercise 8.

Exercises

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

1. In **GreatApe2.scala**, add another **val** field in **GreatApe**. Now add a new subclass **BonoboC** that inherits from **BonoboB**. Write a test for **BonoboC**.

2. Demonstrate that the **Bonobo** constructor can call methods in the **GreatApe** class by adding a method to **GreatApe** and calling it from the **Bonobo** constructor.

3. Define a class **Home** derived from **House** with an additional **Boolean** field **heart**. Satisfy the following tests:
```
val h = new Home
h.toString is "Where the heart is"
h.heart is true
```

4. Modify **VacationHouse** by including a class to represent months rented (pattern matching can help here). Satisfy the following:

```
val v = new VacationHouse("MI","49431",6,8)
v is "Rented house in MI for months of " +
  "June through August."
```

5. Create a class **Trip** including origin, destination, start and end dates. Create a subclass **AirplaneTrip**, including the name of an in-flight movie. Create a second subclass **CarTrip**, including a list of cities you will drive through. Satisfy the following tests:

```
val t = new Trip("Detroit","Houston",
  "5/1/2012","6/1/2012")
val a = new AirplaneTrip("Detroit",
  "London","9/1/1939",
  "10/31/1939", "Superman")
val cities = Vector("Boston",
  "Albany","Buffalo","Cleveland",
  "Columbus","Indianapolis",
  "St. Louis", "Kansas City",
  "Denver","Grand Junction",
  "Salt Lake City","Las Vegas",
  "Bakersfield","San Francisco")
val c = new CarTrip(cities,
  "6/1/2012","7/1/2012")
c.origination is "Boston"
c.destination is "San Francisco"
c.startDate is "6/1/2012"
t is "From Detroit to Houston:"
  + " 5/1/2012 to 6/1/2012"
a is "On a flight from Detroit to"
  + " London, we watched Superman"
c is "From Boston to San Francisco:"
  + " 6/1/2012 to 7/1/2012"
```

6. Does inheritance simplify the implementation of Exercise 5?

7. Can you think of other ways to design the classes in Exercise 5?

8. Show what happens if you try to inherit from a **case class**.

Overriding Methods

So far, the classes we've inherited haven't really done anything to distinguish themselves. Inheritance gets interesting when you start *overriding* methods, which means redefining a method from a base class to do something different in a derived class. Let's look at another version of the **GreatApe** example, this time without worrying about constructor calls:

```
1  // GreatApe3.scala
2  import com.atomicscala.AtomicTest._
3
4  class GreatApe {
5    def call = "Hoo!"
6    var energy = 3
7    def eat() = { energy += 10; energy }
8    def climb(x:Int) = energy -= x
9  }
10
11 class Bonobo extends GreatApe {
12   override def call = "Eep!"
13   // Modify the base-class var:
14   energy = 5
15   // Call the base-class version:
16   override def eat() = super.eat() * 2
17   // Add a new method:
18   def run() = "Bonobo runs"
19 }
20
21 class Chimpanzee extends GreatApe {
22   override def call = "Yawp!"
23   override def eat() = super.eat() * 3
24   def jump = "Chimp jumps"
25   val kind = "Common" // New field
26 }
```

```
27
28 def talk(ape:GreatApe) = {
29   // ape.run()  // Not an ape method
30   // ape.jump // Nor this
31   ape.climb(4)
32   ape.call + ape.eat()
33 }
34
35 talk(new GreatApe) is "Hoo!9"
36 talk(new Bonobo) is "Eep!22"
37 talk(new Chimpanzee) is "Yawp!27"
```

Now we're looking at what the apes do and how it relates to their energy. Any **GreatApe** has a **call**, they store **energy** when they **eat** and expend energy when they **climb**. Note that **call** doesn't change the internal state of the object so it doesn't use parentheses, while **eat** does change the internal state and so it uses parentheses, following the convention described in A Bit of Style.

Notice that **call** is defined the same way in **Bonobo** and **Chimpanzee** as it is in **GreatApe**: It takes no arguments and returns a **String** (as determined through type inference). This combination of name, arguments and return type is the *method signature*.

Both **Bonobo** and **Chimpanzee** have different **call**s than **GreatApe**, so we want to change their definitions of **call**. If you create an identical method signature in a derived class as in a base class, you substitute the behavior defined in the base class with your new behavior. This is called *overriding*.

When Scala sees an identical method signature in the derived class as in the base class, it decides that you've made a mistake, called an *accidental override*. It assumes you've unintentionally chosen the same name, arguments and return type *unless* you use the **override** keyword (which you first saw in Automatic String Conversion) to say "yes, I

mean to do this." The **override** keyword also helps when reading the code so you don't have to compare signatures to notice the overrides.

If you accidentally write a method that has the same name as a method in the base class, you get an error message saying that you forgot the **override** keyword (try it!).

It's even more interesting to take a **Bonobo** or a **Chimpanzee** and treat it as an ordinary **GreatApe**. In the **talk** method on line 28, the method **call** produces the correct behavior in each case. **talk** somehow knows the exact type of the object and produces the appropriate variation of **call**. This is Polymorphism.

Inside **talk**, you can only call **GreatApe** methods. Even though **Bonobo** defines **run** and **Chimpanzee** defines **jump**, neither method is part of **GreatApe**, and the argument to **talk** is *only* a **GreatApe**, not anything more specific.

Often when you override a method, you want to call the base-class version of that method (for one thing, to reuse the code), as with **eat** on lines 16 and 23. This produces a conundrum: If you simply call **eat**, you call the same method you're currently inside (this is *recursion*). To specify that you want to call the base-class version of **eat**, use the **super** keyword, short for "superclass."

Exercises

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

1. On line 7 in **GreatApe3.scala**, the method **eat** is defined with parentheses. Do you recall why?

2. Rework your solution for Exercise 2 in Base Class Initialization by defining **myWords** in the base class and overriding it in the derived class. Satisfy the following tests:
   ```
   val roaringApe =
   ```

```
  new GreatApe2(112, 9, "Male")
roaringApe.myWords is Vector("Roar")
val chattyBonobo =
  new Bonobo2(150, 14, "Female")
chattyBonobo.myWords is
Vector("Roar","Hello")
```

3. Rework your solution for the **Trip**, **AirplaneTrip**, and **CarTrip** exercises in Base Class Initialization, using **super** on the **toString** method from the base class rather than duplicating the code. Start with the same setup as before, but satisfy these tests:

```
t is "From Detroit to Houston:" +
  " 5/1/2012 to 6/1/2012"
a is
  "From Detroit to London:" +
  " 9/1/1939 to 10/31/1939" +
  ", we watched Superman"
c.origination is "Boston"
c.destination is "San Francisco"
c.startDate is "6/1/2012"
c is "From Boston to San Francisco:" +
  " 6/1/2012 to 7/1/2012, we visited" +
  " Vector(Albany, Buffalo, " +
  "Cleveland, Columbus, Indianapolis," +
  " St. Louis, Kansas City, Denver, " +
  "Grand Junction, Salt Lake City, " +
  "Las Vegas, Bakersfield)"
```

Enumerations

An *enumeration* is a collection of names. Scala's **Enumeration** class is a convenient way to manage these names. To create an enumeration, you inherit, typically into an **object**:

```
1  // Level.scala
2  import com.atomicscala.AtomicTest._
3
4  object Level extends Enumeration {
5    type Level = Value
6    val Overflow, High, Medium,
7        Low, Empty = Value
8  }
9
10 Level.Medium is "Medium"
11 import Level._
12 Medium is "Medium"
13
14 { for(n <- Range(0, Level.maxId))
15     yield (n, Level(n)) } is
16   Vector((0, Overflow), (1, High),
17     (2, Medium), (3, Low), (4, Empty))
18
19 { for(lev <- Level.values)
20     yield lev }.toIndexedSeq is
21   Vector(Overflow, High,
22     Medium, Low, Empty)
23
24 def checkLevel(level:Level)= level match {
25   case Overflow => ">>> Overflow!"
26   case Empty => "Alert: Empty"
27   case other =>  s"Level $level OK"
28 }
29
30 checkLevel(Low) is "Level Low OK"
```

```
31 checkLevel(Empty) is "Alert: Empty"
32 checkLevel(Overflow) is ">>> Overflow!"
```

The **Enumeration** names represent various levels (lines 6 and 7). At first glance it can seem like we're creating a set of **val**s and only **Empty** is assigned to **Value** (part of **Enumeration**), but Scala allows you to abbreviate your definitions – this actually means a new **Value** is created for *each* of the **val**s that you see.

Line 5 is a bit surprising at first. It seems like the definition of **Level** on line 4 is enough to introduce a new type, but if you comment out line 5 you see this isn't the case. That's because creating an **object** doesn't create a new type the way creating a **class** does. If we want to treat it as a type, we use the **type** keyword (introduced in Brevity) to alias **Level** to **Value**.

On line 10 you see that if you only create the enumeration, you must qualify each reference to the enumeration names. To eliminate this extra noise, use the **import** statement on line 11, which in this case doesn't import a package from outside this file, but instead imports all the names from the enumeration into the current *name space* (a way to keep names from colliding with each other). On line 12 you see that we no longer need to qualify access to the enumeration names.

There's an **id** field in **Value** which is incremented each time a new **Value** is created. The **for** loop on lines 14 & 15 creates a combination of each **id** and the display name for that **id**, and **yields** the two as a tuple (see Tuples). Notice how the **id**s are numbered from 0 to **maxId**. Line 15 shows how to use an **id** value to look up the corresponding enumeration element (just use parentheses).

Line 19 shows that you can also iterate through the enumeration names, using the **values** field. We call **toIndexedSeq** to produce a **Vector** because that's a familiar collection.

The **checkLevel** method starting on line 24 shows how **Level** has become a new type, but with convenient names used within the method. Again, without the **import** statement on line 11, Scala will not recognize **Level**.

Enumerations can make your code more readable, which is always desirable.

Exercises

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

1. Create an enumeration for **MonthName**, using January, February, etc. Satisfy the following test:
   ```
   MonthName.February is "February"
   MonthName.February.id is 1
   ```

2. In the previous exercise, an **id** of 1 isn't really what we expected for February. We want that to be 2, since February is the second month. Try explicitly setting January to **Value(1)** and leaving the others alone. What does that tell you about what **Value** does? Satisfy the following tests:
   ```
   MonthName2.February is "February"
   MonthName2.February.id is 2
   MonthName2.December.id is 12
   MonthName2.July.id is 7
   ```

3. Building from the previous exercise, demonstrate how to use **import** so you don't have to qualify the name space. Create a method **monthNumber** that returns the appropriate value. Satisfy the following tests:
   ```
   July is "July"
   monthNumber(July) is 7
   ```

4. Create a method **season** that takes a **MonthName** type (from Exercise 1) and returns "Winter" if the month is December, January, or February, "Spring" if March, April, or May, "Summer" if June, July, or August, and "Autumn" if September, October, or November. Satisfy the following tests:
```
season(January) is "Winter"
season(April) is "Spring"
season(August) is "Summer"
season(November) is "Autumn"
```

5. Modify **TicTacToe.scala** from Summary 2 to use enumerations.

6. Modify the **Level** enumeration code from **Level.scala**. Create a new **val** and add another set of values for "Draining, Pooling, and Dry" to the **Level** enumeration. Update the code on lines 14-28 as necessary. Satisfy the following tests:
```
Level.Draining is Draining
Level.Draining.id is 5
checkLevel(Low) is "Level Low OK"
checkLevel(Empty) is "Alert"
checkLevel(Draining) is "Level Draining OK"
checkLevel(Pooling) is "Warning!"
checkLevel(Dry) is "Alert"
```

Abstract Classes

An *abstract class* is like an ordinary class except one or more methods or fields is incomplete. Scala insists that you use the **abstract** keyword for a class containing methods without definitions or fields without initialization. Try removing the **abstract** keyword from either of the following classes and see what message you get:

```
1  // AbstractKeyword.scala
2  abstract class WithValVar {
3    val x:Int
4    var y:Int
5  }
6
7  abstract class WithMethod {
8    def f():Int
9    def g(n:Double)
10 }
```

Lines 3 and 4 *declare* **x** and **y** but provide no initialization values (a *declaration* describes something without providing a *definition* to create storage for a value or code for a method). Without an initializer, Scala considers **var**s and **val**s to be **abstract**, and requires the **abstract** keyword on the class. Without an initializer, Scala has nothing from which to infer type, so it also requires type information for an **abstract var** or **val**.

Lines 8 and 9 declare **f** and **g** but provide no method definitions, again forcing the class to be **abstract**. If you don't give a return type for the method as in line 9, Scala assumes it returns **Unit**.

Abstract methods and fields must somehow exist (be made *concrete*) in the class that you ultimately create using the abstract class as a foundation.

Declaring methods without defining them allows you to describe structure without specifying form. The most common use for this is the *template method pattern*. A template method captures common behavior in the base class and relegates the details that vary to derived classes.

Suppose we are creating a children's program describing animals and what they say, producing statements of the form, "The <animal> goes <sound>." We can easily create a new method in each specific animal class to do this, but that duplicates effort and if we want to change the phrase we'd have to repeat all the changes (and we might miss some).

```
1  // AbstractClasses.scala
2  import com.atomicscala.AtomicTest._
3
4  abstract class Animal {
5    def templateMethod =
6      s"The $animal goes $sound"
7    // Abstract methods (no method body):
8    def animal:String
9    def sound:String
10 }
11
12 // Error -- abstract class
13 // cannot be instantiated:
14 // val a = new Animal
15
16 class Duck extends Animal {
17   def animal = "Duck"
18   // "override" is optional here:
19   override def sound = "Quack"
20 }
21
```

```
22 class Cow extends Animal {
23   def animal = "Cow"
24   def sound = "Moo"
25 }
26
27 (new Duck).templateMethod is
28 "The Duck goes Quack"
29 (new Cow).templateMethod is
30 "The Cow goes Moo"
```

The **templateMethod** in class **Animal** captures common code in a single place. Notice it's completely legal for **templateMethod** to call the **animal** and **sound** methods, even if those methods *aren't defined yet*. This is safe because Scala will not allow you to make an instance of an abstract class, as you see on line 14 (try removing the '**//**' and see what happens).

We define **Duck** and **Cow** by extending **Animal** and *only specifying the behavior that varies*. The common behavior is captured in the base class, in **templateMethod**. Notice that **Duck** and **Cow** are not **abstract** because all their methods now have definitions – we call such classes *concrete*.

When you provide a definition for an abstract method from a base class, the keyword **override** is optional. Technically, you're *not* overriding because there's no definition to override. When something is optional in Scala, we generally leave it out to reduce visual noise.

Since **Duck** and **Cow** are concrete, they can be instantiated, as you see on lines 27 and 29. Because we are only creating the objects in order to call **templateMethod** on them, we use a shortcut: We don't assign the objects to an identifier. Instead, we surround the **new** expression with parentheses and call **templateMethod** on the resulting object.

Abstract classes can have arguments, just like ordinary classes:

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

abstract class Adder(x:Int) {
  def add(y:Int):Int
}
```

Since **Adder** is **abstract**, it cannot be instantiated, but any class that inherits from **Adder** can now perform base-class initialization by calling the **Adder** constructor (as you'll see in the exercises).

Exercises

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

1. Modify **Animal** and its subclasses to also indicate what each animal eats. Satisfy the following tests:
   ```
   val duck = new Duck
   duck.food is "plants"
   val cow = new Cow
   cow.food is "grass"
   ```

2. Add new classes for **Chicken** and **Pig**. Satisfy the following tests:
   ```
   val chicken = new Chicken
   chicken.food is "insects"
   val pig = new Pig
   pig.food is "anything"
   ```

3. Inherit from the **Adder** class to make it operational. Satisfy the following tests:

```
class NumericAdder(val x:Int)
extends Adder(x) {
  def add(y:Int):Int = // Complete this
}
val num = new NumericAdder(5)
num.add(10) is 15
```

4. Can **case** classes inherit from **abstract** classes?

5. Inherit a class from **Animal** and try making an **animal** method that takes an argument.

Traits

Inheritance creates a new class by building on an existing class, so you don't need to rewrite everything from scratch. Traits are an alternate approach to creating classes: they let you acquire abilities piecemeal rather than inheriting them as a clump. Traits are small, logical concepts; basic pieces of functionality that allow you to easily "mix in" ideas to create a class. For this reason they are often called *mixin types*.

Ideally, a trait represents a single concept. For example, traits allow you to separate the concepts "color," "texture" and "resilience," rather than putting them all in one place just because you're creating a base class.

A trait definition looks like a class, but uses the **trait** keyword instead of **class**. To combine traits into a class, you always begin with the **extends** keyword, then add additional traits using the **with** keyword:

```
1  // Materials.scala
2
3  trait Color
4  trait Texture
5  trait Hardness
6
7  class Fabric
8
9  class Cloth extends Fabric with Color
10   with Texture with Hardness
11
12 class Paint extends Color with Texture
13   with Hardness
```

Lines 9-10 create **Cloth** from the **Fabric** class using the **extends** keyword, and adds additional traits using **with**. A class can only

inherit from a single concrete or **abstract** base class, but it can combine as many traits as you want. If there is no concrete or **abstract** base class, as on lines 12-13, you still start with the **extends** keyword for the first trait, followed by the **with** keyword for the remaining traits.

Like an abstract class, fields and methods in traits can have definitions or they can be left abstract:

```
1  // TraitBodies.scala
2
3  trait AllAbstract {
4    def f(n:Int):Int
5    val d:Double
6  }
7
8  trait PartialAbstract {
9    def f(n:Int):Int
10   val d:Double
11   def g(s:String) = s"($s)"
12   val j = 42
13 }
14
15 trait Concrete {
16   def f(n:Int) = n * 11
17   val d = 1.61803
18 }
19
20 /* None of these are legal -- traits
21 cannot be instantiated:
22 new AllAbstract
23 new PartialAbstract
24 new Concrete
25 */
26
27 // Scala requires 'abstract' keyword:
28 abstract class Klass1 extends AllAbstract
```

```
  with PartialAbstract

/* Can't do this -- d and f are undefined:
new Klass1
*/

// Class can provide definitions:
class Klass2 extends AllAbstract {
  def f(n:Int) = n * 12
  val d = 3.14159
}

new Klass2

// Concrete's definitions satisfy d & f:
class Klass3 extends AllAbstract
  with Concrete

new Klass3

class Klass4 extends PartialAbstract
  with Concrete

new Klass4

class Klass5 extends AllAbstract
  with PartialAbstract with Concrete

new Klass5

trait FromAbstract extends Klass1
trait fromConcrete extends Klass2

trait Construction {
  println("Constructor body")
}

class Constructable extends Construction
```

```
67  new Constructable
68
69  // Create unnamed class on-the-fly:
70  val x = new AllAbstract with
71    PartialAbstract with Concrete
```

A free-standing trait cannot be instantiated; for one thing, it does not have a full-fledged constructor. When combining traits to generate a new class, all the fields and methods must have definitions or Scala will insist that you include the **abstract** keyword, as on lines 28-29 (put another way, an **abstract** class can inherit from a trait). Definitions can be provided by the class, as with **Klass2**, or through other traits as **Concrete** does in **Klass3**, **Klass4** and **Klass5** (the methods in an abstract class will also work).

Traits can inherit from abstract or concrete classes (lines 59-60). Lines 62-67 show that, even though traits cannot have constructor arguments, they can have constructor bodies.

Lines 70-71 show an interesting trick: creating an instance of a class that you assemble at the site of creation. The resulting object has no type name. This technique creates a single instance of an object without creating a new named class just for that one usage.

Traits can inherit from other traits:

```
1   // TraitInheritance.scala
2
3   trait Base {
4     def f = "f"
5   }
6
7   trait Derived1 extends Base {
8     def g = "17"
9   }
10
```

```
trait Derived2 extends Derived1 {
  def h = "1.11"
}

class Derived3 extends Derived2

val d = new Derived3

d.f
d.g
d.h
```

When combining traits, it's possible to mix two methods with the same *signature* (the name combined with the type). If method or field signatures collide, resolve the collisions by hand, as seen in **object C** (Here, an **object** serves as a shorthand for creating a class and then an instance):

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

trait A {
  def f = 1.1
  def g = "A.g"
  val n = 7
}

trait B {
  def f = 7.7
  def g = "B.g"
  val n = 17
}

object C extends A with B {
  override def f = 9.9
  override val n = 27
  override def g = super[A].g + super[B].g
```

```
20 }
21
22 C.f is 9.9
23 C.g is "A.gB.g"
24 C.n is 27
```

The methods **f** and **g** and the field **n** have identical signatures across traits **A** and **B** so Scala doesn't know what to do and gives an error message (try individually commenting lines 17-19). Methods and fields can be overridden with new definitions (lines 17-18), but methods can also access the base versions of themselves using the **super** keyword as demonstrated on line 19 (this behavior is not available for fields but might be in the future). Collisions where the identifier is the same but the type is different are not allowed in Scala, so you cannot resolve them.

Trait fields and methods can be used in calculations, even though they're not yet defined:

```
1  // Framework.scala
2  import com.atomicscala.AtomicTest._
3
4  trait Framework {
5    val part1:Int
6    def part2:Double
7    // Even without definitions:
8    def templateMethod = part1 + part2
9  }
10
11 def operation(impl:Framework) =
12   impl.templateMethod
13
14 class Implementation extends Framework {
15   val part1 = 42
16   val part2 = 2.71828
17 }
```

```
18
19 operation(new Implementation) is 44.71828
```

On line 8, **templateMethod** uses **part1** and **part2** even though they have no definitions at that point. Traits guarantee that all abstract fields and methods must be implemented before any objects can be created – and you can't call a method unless you've got an object.

Defining an operation in a base type that relies on pieces to be defined by a derived type is commonly called the *Template Method* pattern and is the foundation for many *frameworks*. The framework designer writes the template methods, and you inherit from those types and customize it to your needs by filling in the missing pieces (as on lines 14-17).

Some object-oriented languages support *multiple inheritance* to combine multiple classes. Traits are usually considered a superior solution. If you have a choice between classes and traits, prefer traits.

Exercises

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

1. Create a **trait BatteryPower** to report remaining charge. If the charge is greater than 40%, report "green." If the charge is between 20-39%, report "yellow." If the charge is less than 20%, report "red." Instantiate the trait and satisfy the following tests:

```
class Battery extends
  EnergySource with BatteryPower
val battery = new Battery
battery.monitor(80) is "green"
battery.monitor(30) is "yellow"
battery.monitor(10) is "red"
```

2. Create a new class **Toy**. Use **Toy** and **BatteryPower** to create a new class **BatteryPoweredToy**. Satisfy the following tests:
```
val toy = new BatteryPoweredToy
toy.monitor(50) is "green"
```

3. Instantiate an object without creating an intermediate class, using **Toy** and **BatteryPower** directly. Satisfy the following test:
```
val toy2 = new // Fill this in
toy2.monitor(50) is "green"
```

Uniform Access & Setters

Here's an example of the flexibility (and good design) of Scala:

```
1  // UniformAccess.scala
2  import com.atomicscala.AtomicTest._
3
4  trait Base {
5    def f1:Int
6    def f2:Int
7    val d1:Int
8    val d2:Int
9    var d3:Int
10   var n = 1
11 }
12
13 class Derived extends Base {
14   def f1 = 1
15   val f2 = 1 // Was def, now val
16   val d1 = 1
17   // Can't do this; must be a val:
18   // def d2 = 1
19   val d2 = 1
20   def d3 = n
21   def d3_=(newVal:Int) = n = newVal
22 }
23
24 val d = new Derived
25 d.d3 is 1 // Calls getter (line 20)
26 d.d3 = 42 // Calls setter (line 21)
27 d.d3 is 42
```

The **abstract** methods declared on lines 5 and 6 have the identical implementations on lines 14 and 15, with a single exception: line 14, like its base version, is a **def**, but line 15 implements the **def** on line 6 using a **val**! Is this a problem? Well, whenever we call the method **f2**, it's treated as a method that produces an **Int**. And when we reference the **val f2**, it also produces an **Int**. In Scala, methods without arguments that produce a result can be treated identically to **val**s that produce a result of that same type. This is an example of the *Uniform Access Principle*: From the client programmer's perspective, you can't tell how something is implemented (here, you can't tell the difference between storage and computation).

Going the other way doesn't work: if you have a **val** in the base type, you can't implement it using a **def**. Scala says "d2 must be a stable, immutable value." That's because a **val** represents a promise that things can't change, while a **def** means you execute code in the process of producing your result.

But what if the field is a **var**, as on line 9? Then there's no promise that it always has to be the same, and it should work if you implement it with a **def**. However, you can't just implement **d3** using line 20 by itself, because that only *produces* ("gets") the result, and a **var** must also be settable. Scala will say, "... an abstract var requires a *setter* in addition to the getter." Line 21 shows the form of a setter; the identifier followed by "**_=**" and a single argument. Now you both read the variable via line 20's method, and change the variable via line 21.

Exercises

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

1. Show that the uniform access principle demonstrated in **UniformAccess.scala** works when **Base** is an abstract class.

2. Does the uniform access principle demonstrated in **UniformAccess.scala** work when **Base** is a concrete class? Can you

think of a different way to use a setter here? Hint: Look at Base Class Initialization.

3. Create a class with a **var** named **internal** and a getter and setter for **internal** named **x**, and demonstrate that it works.

Reaching into Java

Sometimes you must import a Java package to use Java classes. Java's **Date** class, for example, is not automatically available. It's in **java.util.Date** and you import it the same way you import Scala packages. Using the REPL:

```
scala> val d = new Date
<console>:7: error: not found: type Date
       val d = new Date
                   ^

scala> import java.util.Date
import java.util.Date

scala> val d = new Date
d: java.util.Date = Sat Aug 09 14:27:18 MDT 2014
```

The entire Java standard library is available using imports like this.

You can also download third-party Java libraries and use them in Scala. This is powerful because it builds on all the effort invested in those Java libraries. For example, you can fit points to a line using a *linear regression least-squares fit* (see Wikipedia) provided by the popular Apache Commons Math library, written in Java.

Download the Apache Commons Math library from `archive.apache.org/dist/commons/math/binaries`. For this edition of the book, the latest version of the library was `commons-math3-3.3-bin.zip`. If there's a more recent one you can download that, but be aware you might need to make some adjustments in the instructions and code below (for example, once Apache moved to the "math3" library, we had to change the import from `org.apache.commons.math` to `org.apache.commons.math3`).

Extract the contents of the zip file to the directory on your disk where you installed AtomicScala in your appropriate "Installation" atom. If you chose our defaults, this is **C:\AtomicScala** on Windows, or **~/AtomicScala** on Mac or Linux.

For now, just add this library to your **CLASSPATH** when you run the Scala script, by specifying the path to the library with the **-classpath** flag. For the default directory the shell command is (all on a single line):

```
scala -classpath $CLASSPATH:$HOME/AtomicScala/commons-
math3-3.3/commons-math3-3.3.jar LinearRegression.scala
```

Type the line above from your **AtomicScala/examples** directory to make sure your **CLASSPATH** is set properly.

To use the library without using the **–classpath** argument, add **AtomicScala/commons-math3-3.3/commons-math3-3.3.jar** to the **CLASSPATH** in your profile using the same process as in your appropriate "Installation" atom.

Import the **SimpleRegression** package the same way you import any other library:

```
1  // LinearRegression.scala
2  import com.atomicscala.AtomicTest._
3  import org.apache.commons.math3._
4  import stat.regression.SimpleRegression
5
6  val r = new SimpleRegression
7  r.addData(1, 1)
8  r.addData(2, 1.1)
9  r.addData(3, 0.9)
10 r.addData(4, 1.2)
```

```
11
12 r.getN is 4
13 r.predict(6) is 1.19
```

The code in lines 6-10 creates an object of type **SimpleRegression** and adds **x** and **y** coordinates. On line 12 we ensure there are 4 data points. On line 13, we ask for a prediction of the value for x=6. We're using this class as if it were a Scala class, when in fact it's implemented in Java. The Java library ecosystem is a huge benefit to Scala.

Exercises

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

1. Import the class **SimpleDateFormat**, used for specifying what the input date string looks like, from **java.text.SimpleDateFormat**. Use Java's **SimpleDateFormat** to create a pattern, named **datePattern**, that you parse as 2-digit Month/2-digit Day/2-digit Year (Hint: MM/dd/yy). Satisfy the following test:
   ```
   val mayDay = datePattern.parse("05/01/12")
   mayDay.getDate is 1
   mayDay.getMonth is 4
   ```

2. In your solution for Exercise 1, why do you specify "MM" in the **SimpleDateFormat** pattern instead of "mm?" What does the parser expect if you specify "mm?" Try it.

3. In your solution for Exercise 1, why is May represented as a 4 instead of a 5? Is this what you expect? Is this consistent with the day?

4. The Apache Commons Math library (imported in this atom), contains a class called **Frequency** in **org.apache.commons.math.stat.Frequency**. Use its **addValue** method to add some strings to **Frequency**. Satisfy the following test:

```
val f = new Frequency
// add values for cat, dog, cat, bird,
//   cat, cat, kitten, mouse here
f.getCount("cat") is 4
```

5. Using the Apache Commons Math library that you imported above, calculate the mean and standard deviation and percentile of the following data set: 10, 20, 30, 80, 90, and 100. Satisfy the following tests:

```
val s = new SummaryStatistics
// add values here
s.getMean is 55
s.getStandardDeviation is
39.370039370059054
```

Applications

To keep things as simple as possible, we've used scripts in this book. A more common way to build a program is to compile everything, including code we've put in scripts. To do this, create an **object** that extends **App**. The constructor code for that **object** executes when you run the program. Here's what it looks like:

```
1  // Compiled.scala
2
3  object WhenAmI extends App {
4    hi
5    println(new java.util.Date())
6    def hi = println("Hello! It's:")
7  }
```

The **object** does not need to be in a file by itself. As usual, constructor statements execute in order. Here, the **hi** method executes followed by a call to the **Date** class in the Java standard library (as shown in Reaching into Java). To compile this application, use **scalac** in the shell:

```
scalac Compiled.scala
```

It doesn't matter what you call the file; the name of the resulting program depends on the name of the **object**. A directory listing shows **WhenAmI.class** which is the compiled **object**. To run the program in the shell, you use **scala** and the **object** name (but not the **.class** extension):

```
scala WhenAmI
```

Now instead of running the program as a script, Scala finds the compiled object and executes that.

What if you want to pass arguments on the command line? **App** comes with an object **args** that contains the command-line arguments as **String**s. Here's an application that echoes its arguments:

```
1  // CompiledWithArgs.scala
2
3  object EchoArgs extends App {
4    for(arg <- args)
5      println(arg)
6  }
```

You compile it as before:

```
scalac CompiledWithMain.scala
```

If you run the program like this:

```
scala EchoArgs bar baz bingo
```

You see the output:

```
bar
baz
bingo
```

There's another form that follows a pattern used in older programming languages: you define a method called **main**, and the method arguments contain the command-line arguments. Note that here, you *do not* inherit from **App**:

```
// CompiledWithMain.scala

object EchoArgs2 {
  def main(args:Array[String]) =
    for(arg <- args)
      println(arg)
}
```

For our purposes, an **Array** is the same as a **Vector**, and all the arguments come in as **String**s. There's no particular reason to use a **main** other than that it can make the code familiar to programmers from other languages (Java, in particular).

Exercises

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

1. Use the code from **Compiled.scala**. Compile it using **scalac**, as described above. Run it with the shell command **scala WhenAmI**.

2. In Exercise 1 from Traits, you implemented a class called **Battery**. Rework that as an application (hint: use a companion object). Run the same tests inside the application object.

3. Adding to your solution for the previous exercise, pass in an argument to represent the charge(s). Compile the application, then run it with the following shell command to verify the results:
   scala Battery2 80 30 10
   Hint: recall that you can convert from a **String** to an **Int** using **toInt**.

A Little Reflection

Reflection means taking an object and holding it up to a mirror, so it discovers things about itself. For example, we often want to find out an object's class name. Here's a **trait** that automatically adds a **toString** to any class so it displays the class name:

```
1  // Name.scala
2  package com.atomicscala
3  import reflect.runtime.currentMirror
4
5  object Name {
6    def className(o:Any) =
7      currentMirror.reflect(o).symbol.
8      toString.replace('$', ' ').
9      split(' ').last
10 }
11
12 trait Name {
13   override def toString =
14     Name.className(this)
15 }
```

The **className** method takes an **Any** object and produces that object's class name. To do this, we **reflect** the object in the **currentMirror**; this gives us access to the object's **symbol**, which we convert to a string.

This string isn't as plain as we'd like. Sometimes there are spaces in the string, and sometimes Scala inserts '**$**' signs in the name. If we replace the '**$**' signs with spaces using **replace**, we can then use Scala's **split** method to break the string across spaces. The result is a sequence of strings, and by calling **last** we get the end element which is the actual name of the class.

Now any class we combine with the **Name** trait will automatically include a **toString** that knows its own name. By passing the **this** keyword to **className**, we pass the current object.

Now have a reusable tool to combine with any class and automatically add a **toString** method:

```
// Solid.scala
import com.atomicscala.AtomicTest._
import com.atomicscala.Name

class Solid extends Name
val s = new Solid
s is "Solid"

class Solid2(val size:Int) extends Name {
  override def toString =
    s"${super.toString}($size)"
}
val s2 = new Solid2(47)
s2 is "Solid2(47)"
```

Solid combines **Name** in the simplest way possible, but **Solid2** overrides **toString** again, calling **Name**'s version using the **super** keyword to get the class name, then adding the **size** argument to produce more informative output – just like a **case** class.

Scala's *reflection API* is much more powerful and complex than we've shown here.

Exercises

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

1. Call **println** on an instance of a **case** class. Now combine the **case** class with **Name** and notice the difference. Remember to compile **Name.scala** and import it.

2. Can you use reflection on a class that isn't a **case** class? Repeat Exercise 1 using a non-**case** class.

3. Comment out the code in **Name.scala** that replaces the **$** with spaces and splits the **String**, so you see what Scala's reflection returns before our modifications. Repeat Exercise 2 using this new class.

4. In **TraitBodies.scala** in the Traits atom, we assert that the code on lines 70-71 creates a class without a type name. Determine if this is an exact statement.

Polymorphism

Polymorphism is an ancient Greek term that means "many forms." In programming, polymorphism means we perform the same operation on different types.

Base classes and traits are useful for more than just assembling classes. If we create a new class using class **A** along with traits **B** and **C**, we can choose to treat that new class as if it were only an **A** or only a **B** or only a **C**. For example, if both animals and vehicles can move, and you mix in a **Mobile** trait for both of them, you can write a method that takes a **Mobile** argument, and that method will automatically work for both animals and vehicles.

Suppose we want to create a fantasy game. Each game element will draw itself on the screen based on its location in the game world, and when two elements are in proximity, they interact. Here's a rough draft that uses polymorphism to give you a general idea of how you can design such a game, although we leave out the vast majority of implementation details:

```
1  // Polymorphism.scala
2  import com.atomicscala.AtomicTest._
3  import com.atomicscala.Name
4
5  class Element extends Name {
6    def interact(other:Element) =
7      s"$this interact $other"
8  }
9
10 class Inert extends Element
11 class Wall extends Inert
12
13 trait Material {
14   def resilience:String
```

```
}
trait Wood extends Material {
  def resilience = "Breakable"
}
trait Rock extends Material {
  def resilience = "Hard"
}
class RockWall extends Wall with Rock
class WoodWall extends Wall with Wood

trait Skill
trait Fighting extends Skill {
  def fight = "Fight!"
}
trait Digging extends Skill {
  def dig = "Dig!"
}
trait Magic extends Skill {
  def castSpell = "Spell!"
}
trait Flight extends Skill {
  def fly = "Fly!"
}

class Character(var player:String="None")
  extends Element
class Fairy extends Character with Magic
class Viking extends Character
  with Fighting
class Dwarf extends Character with Digging
  with Fighting
class Wizard extends Character with Magic
class Dragon extends Character with Magic
  with Flight

val d = new Dragon
d.player = "Puff"
```

```
52 d.interact(new Wall) is
53 "Dragon interact Wall"
54
55 def battle(fighter:Fighting) =
56   s"$fighter, ${fighter.fight}"
57 battle(new Viking) is "Viking, Fight!"
58 battle(new Dwarf) is "Dwarf, Fight!"
59 battle(new Fairy with Fighting) is
60 "anon, Fight!"
61
62 def fly(flyer:Element with Flight,
63   opponent:Element) =
64     s"$flyer, ${flyer.fly}, " +
65     s"${opponent.interact(flyer)}"
66
67 fly(d, new Fairy) is
68 "Dragon, Fly!, Fairy interact Dragon"
```

The **interact** method on line 6 is how one game element interacts with another when the two are in proximity. **interact** means something different depending on the exact types of elements participating in the interaction, and this is an interesting and challenging design problem in itself ... which we ignore by simply printing out the names of the two interacting elements.

Now we create different types of elements and traits, mixing them to achieve different effects. Note that, just like classes, traits can inherit from each other.

The **Skill** trait on line 25 is only used for its name, to classify the traits that follow it. However, if you later decide that all **Skill**s need common fields or methods, add those to **Skill** and they automatically appear in everything that incorporates it.

At this point, we're ready to define some characters for players to actually manipulate. The constructor argument **player**, on line 39, has

a default argument (see Named & Default Arguments), specified by the '=' after the argument type and the string **"None"**.

After assembling several different **Character** types, we create a **Dragon** on line 50, and then on line 51 we change **player** from its default value of **"None"** to **"Puff"**. We can do this because **player** is a **var** rather than our usual **val**, and a **var** can be changed. Ordinarily we'd stick with **val** and use base-class constructor calls (You will make that change in the exercises).

Line 52 is our first example of polymorphism. **Dragon** inherits the **interact** method from **Element**, but **interact** takes an **Element** argument – we pass it a **Wall**. However, **Wall** ultimately inherits from **Element**, so we say that "**Wall** is an **Element**" and Scala agrees. The **interact** method takes an **Element** *or anything derived from* ***Element***. That's polymorphism, and it's powerful because any method you write is more general. It applies to more types than just the type you write it for – it also applies to anything that inherits from that type. This is transparent and safe because Scala guarantees that a derived class "is a" base class by ensuring that the derived class has (at least) all the methods of the base class.

Line 55 shows a second example, this time using a trait polymorphically. The argument **fighter** happens to be a trait, and this means any object incorporating that trait can be safely passed into the **fight** method. The **Fighting** trait has only one method: **fight**, and that's *all* it can access in **fighter** because nothing else is defined in the **Fighting** trait.

Both **Viking** and **Dwarf** include the **Fighting** trait, so they can both be passed to **battle**, which again demonstrates polymorphism. Without polymorphism, you must write specific methods; for example **battle_viking** for **Viking** objects, and **battle_dwarf** for **Dwarf** objects. With polymorphism, you write one method that not only works with **Viking**s and **Dwarf**s, but also with anything else that implements

Fighting, including types you haven't thought of at the time you write **battle**. Polymorphism is a tool that allows you to write less code and make it more reusable.

As an example of "types you haven't thought of yet," consider line 59, in particular the argument passed to **battle**:

```
new Fairy with Fighting
```

The type of the object is created for us, as we write the **new** expression! In the expression, we combine the existing **Fairy** class with the **Fighting** trait, which creates a new class, and we immediately make an instance of that class. We didn't give the class a name, so Scala generates one for us: **$anon$1** ("anon" is short for "anonymous") and when **Element**'s **id** encounters this, it produces the '**1**'.

This technique of putting together types just as you need them also works for arguments, as you see on line 62. The first argument, **flyer**, mixes **Element with Flight**. Because we included **id** in the **Skill** mix, **fly** will work anyway for **flyer.id** and **flyer.fly**, but for **opponent.interact(flyer)** to work, **flyer** must actually be an **Element**. By saying **Element with Flight**, Scala will ensure that any argument you pass as a **flyer** includes both **Element** and **Flight**, so **fly** can properly call everything it must.

A question often arises: "How did you know to do it this way?" This is the design challenge. Once you decide what you want to build, there are different ways to assemble it. So far, you've seen how to create a base class and add new methods during inheritance, or mix in functionality using traits. These are design decisions you make using a combination of experience and observing how your system is used. You decide what makes sense, based on the requirements of your system. That's the design process.

The pragmatic approach is not to assume that you get it right the first time. Instead, write something, get it working, and then see how it looks. As you learn, "refactor" your code until the design feels right (don't settle for the first thing that works).

Exercises

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

1. Write code that verifies the animals/vehicles description in the second paragraph of this atom.

2. Add a **draw** method to **Element** in **Polymorphism.scala**. Satisfy the following tests:
```
val e = new Element
e.draw is "Drawing the element"
val in = new Inert
in.draw is "Inert drawing!"
val wall = new Wall
wall.draw is "Inert drawing!"
```

3. Building on the previous exercise, add a new **draw** method to **Wall** (that is, don't use the **Inert draw** method). Satisfy the following test:
```
val wall = new Wall
wall.draw is "Don't draw on the wall!"
```

4. In the definition of **Character** on line 39 of **Polymorphism.scala**, we use a **var** for the player, and then change the player on line 51. Use a **val** to accomplish the same thing. Satisfy the following test:
```
class Character(val player:String="None")
  extends Element
// Change the next line
class Dragon extends Character
val d = new Dragon("Puff")
d.player is "Puff"
```

5. Create a class **Seed** with subclasses **Tomato**, **Corn** and **Zucchini**. Override **toString** in each subclass to indicate the type of plant. Create a class **Garden** which takes as its constructor argument any number of **Seed**s. Store the **Seed**s in a **Vector** inside **Garden**. Override **Garden**'s **toString** to produce the string representation of this **Vector**, formatted with the **mkString** method. Satisfy the following test:

```
val garden = new Garden(
new Tomato, new Corn, new Zucchini)
garden is "Tomato, Corn, Zucchini"
```

6. Create a **trait Shape** with a method **draw** that returns a **String**. Create concrete **Ellipse** and **Rectangle** subclasses of **Shape**. Create a **Circle** subclass of **Ellipse** and a **Square** subclass of **Rectangle**. Create a **Drawing** class with a constructor that takes any number of **Shape** objects, and stores it internally in a **Vector**. Create **draw** methods for all these classes and an additional **toString** for **Drawing**. Satisfy the following tests:

```
val drawing = new Drawing(
  new Rectangle, new Square,
  new Ellipse, new Circle)
drawing.draw is "Vector(Rectangle," +
  " Square, Ellipse, Circle)"
drawing is "Rectangle, Square," +
  " Ellipse, Circle"
```

Composition

Suppose you are modeling a house. You might start like this:

```
1  // House1.scala
2
3  trait Building
4  trait Kitchen
5  trait House extends Building with Kitchen
```

That reads nicely: "A house is a building with a kitchen." But what if your house includes an additional space for another person to live and cook? Now you have two kitchens, and you can't inherit the same trait twice (even if you put a type parameter on that trait).

Inheritance describes an *is-a* relationship, and it's often helpful to read the description aloud: "A house is a building." That sounds right, doesn't it? When the is-a relationship makes sense, inheritance usually makes sense.

A trait represents a capability, so we say it is a *has-ability* relationship. So we might read, "A house has … kitchen … ability." It almost makes sense, but it's rather forced.

The most fundamental relationship is not inheritance, nor traits, but *composition*. Composition is often overlooked because it seems so simple: you just put something *inside*. Composition is a *has-a* relationship, and it solves our problem because we can say "The house has two kitchens:"

```
1  // House2.scala
2
3  trait Building
4  trait Kitchen
```

```
5
6   trait House extends Building {
7     val kitchen1:Kitchen
8     val kitchen2:Kitchen
9   }
```

If you want to allow any number of kitchens, compose with a collection:

```
1   // House3.scala
2
3   trait Building
4   trait Kitchen
5
6   trait House extends Building {
7     val kitchens:Vector[Kitchen]
8   }
```

We spend time and effort understanding inheritance and mixins because they are more complex, but this often gives the impression that they are somehow more important. On the contrary:

Prefer composition to inheritance

Composition produces simpler designs and implementations. This doesn't mean you should try to avoid inheritance or mixins. Not at all. It's just that we tend to get bound up in those more complicated relationships. The maxim "prefer composition to inheritance" is a reminder to step back, look at your design, and wonder whether to simplify things using composition. The ultimate goal is to properly apply your tools and produce a good design.

A kitchen has the ability to store food and utensils, cook food, and clean utensils. Abilities translate into traits:

```
// House4.scala

trait Building
trait Food
trait Utensil
trait Store[T]
trait Cook[T]
trait Clean[T]
trait Kitchen extends Store[Food]
  with Cook[Food] with Clean[Utensil]
  // Oops. Can't do this:
  // with Store[Utensil]
  // with Clean[Food]

trait House extends Building {
  val kitchens:Vector[Kitchen]
}
```

Even though we'd like to store utensils and clean food, this approach doesn't allow it because you can't inherit a trait twice. Once you have an ability, adding that ability a second time doesn't mean anything.

Once again, we "prefer composition to inheritance." What if it's not the kitchen that has these abilities, but the items themselves? This is not to say that a vegetable automatically washes itself, but rather that a vegetable has the ability to be washed. The kitchen does have storage and a sink, but those are general-purpose and not dedicated to either food or utensils (You can also wash shoes, babies and small dogs in the sink). The model becomes:

```
// House5.scala

trait Building
trait Room
trait Storage
trait Sink
```

```
7  trait Store[T]
8  trait Cook[T]
9  trait Clean[T]
10 trait Food extends Store[Food]
11   with Clean[Food] with Cook[Food]
12 trait Utensil extends Store[Utensil]
13   with Clean[Utensil] with Cook[Utensil]
14
15 trait Kitchen extends Room {
16   val storage:Storage
17   val sinks:Vector[Sink]
18   val food:Food
19   val utensils:Vector[Utensil]
20 }
21
22 trait House extends Building {
23   val kitchens:Vector[Kitchen]
24 }
```

We've added another is-a relationship here: a **Kitchen** is a **Room**. A kitchen can contain several rooms, but the "roomness" of a kitchen is fundamental.

To be honest, our first impulse was to say, "A kitchen has the ability to store things," thus storage is an ability and we should inherit **Kitchen** from **Storage**. Further thought and *another* application of "prefer composition to inheritance" showed that "a kitchen *has* storage" not only makes more sense, it's more flexible. Note that line 17 allows the kitchen to have more than one sink – a good test is whether composition makes it easy to have multiple items (and different kinds). When inheriting a trait, you can't represent more than one.

Exercises

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

1. Create a **trait Mobility** with a **String** method **mobility** that returns a description of the type of mobility. Create similar traits for **Vision** and **Manipulator**. Inherit a **class Robot** that takes **mobility**, **vision**, and **manipulator** arguments, and overrides **toString**. Satisfy the following tests:

```
val walker = new Robot("Legs",
  "Visible Spectrum", "Magnet")
walker is
  "Legs, Visible Spectrum, Magnet"
val crawler = new Robot("Treads",
  "Infrared", "Claw")
crawler is "Treads, Infrared, Claw"
val arial = new Robot("Propeller",
  "UV", "None")
arial is "Propeller, UV, None"
```

2. Start with your solution to Exercise 1. Turn the traits into case classes. Make those classes the arguments to **class Robot**. Satisfy the following tests:

```
val walker = new Robot(
  Mobility("Legs"),
  Vision("Visible Spectrum"),
  Manipulator("Magnet"))
walker is "Mobility(Legs), " +
  "Vision(Visible Spectrum)," +
  " Manipulator(Magnet)"
val crawler = new Robot(
  Mobility("Treads"),
  Vision("Infrared"),
  Manipulator("Claw"))
crawler is "Mobility(Treads)," +
  " Vision(Infrared), " +
  "Manipulator(Claw)"
```

```
val arial = new Robot(
  Mobility("Propeller"),
  Vision("UV"),
  Manipulator("None"))
arial is "Mobility(Propeller)," +
  " Vision(UV), Manipulator(None)"
```

3. Start with your solution for Exercise 2. Change the arguments for **Robot** to allow more than one ability. Use **mkString** in your overridden **toString**. Satisfy the following tests:

```
val bot = new Robot(
  Vector(
    Mobility("Propeller"),
    Mobility("Legs")),
  Vector(
    Vision("UV"),
    Vision("Visible Spectrum")),
  Vector(
    Manipulator("Magnet"),
    Manipulator("Claw"))
)

bot is "Mobility(Propeller)," +
" Mobility(Legs) | Vision(UV)," +
" Vision(Visible Spectrum) | " +
"Manipulator(Magnet), " +
"Manipulator(Claw)"
```

4. Modify your solution for Exercise 3 so the case classes inherit from a **trait Ability**, and change **Robot** to take a single **Vector[Ability]**. Satisfy the following tests:

```
val bot = new Robot(
  Vector(Mobility("Propeller"),
    Mobility("Legs"),
    Vision("UV"),
    Vision("Visible Spectrum"),
```

```
    Manipulator("Magnet"),
    Manipulator("Claw"))
)

bot is "Mobility(Propeller), " +
"Mobility(Legs), Vision(UV), " +
"Vision(Visible Spectrum), " +
"Manipulator(Magnet), " +
"Manipulator(Claw)"
```

5. Modify your solution for Exercise 3 to implement a "builder" approach. **Robot** has no constructor arguments, but instead has methods to **addMobility**, **addVision** and **addManipulator**. Satisfy the following tests:

```
val bot = new Robot
bot.addMobility(
  Mobility("Propeller"))
bot.addMobility(
  Mobility("Legs"))
bot.addVision(
  Vision("UV"))
bot.addVision(Vision(
  "Visible Spectrum"))
bot.addManipulator(
  Manipulator("Magnet"))
bot.addManipulator(
  Manipulator("Claw"))

bot is "Mobility(Propeller)," +
" Mobility(Legs) | Vision(UV)," +
" Vision(Visible Spectrum) | " +
"Manipulator(Magnet)," +
" Manipulator(Claw)"
```

6. Modify your solution for Exercise 5 to turn the "add" methods into overloaded '+' operators. For chaining, you must return **this** from

each operator. Compare all the solutions for this atom. Satisfy the following tests:

```
val bot = new Robot +
  Mobility("Propeller") +
  Mobility("Legs") +
  Vision("UV") +
  Vision("Visible Spectrum") +
  Manipulator("Magnet") +
  Manipulator("Claw")

bot is "Mobility(Propeller)," +
" Mobility(Legs) | Vision(UV)," +
" Vision(Visible Spectrum) |" +
" Manipulator(Magnet)," +
" Manipulator(Claw)"
```

⚛ Using Traits

Scala enables you to partition your model into appropriate pieces, whereas some languages force you into awkward abstractions. Traits (and the mixins they enable) might be the most powerful of these tools. Traits not only allow elegant and meaningful syntax, they prevent code duplication (and the associated bloat and maintenance headaches). So:

- Prefer traits to more concrete types (more abstract == more flexible)
- Divide models into independent pieces
- Delay concreteness

The primary difference between traits and abstract classes is that traits cannot have constructor arguments (although they can contain constructor expressions within the trait body). This makes sense because a trait is more of a "capability" than it is a physical thing – a trait is designed for reuse rather than instantiation. In this example, the **Aerobic** trait calculates whether someone is exercising in the aerobic zone, while **Activity** describes what they are doing:

```
1  // AerobicExercise.scala
2  import com.atomicscala.AtomicTest._
3
4  trait Aerobic {
5    val age:Int
6    def minAerobic = .5 * (220 - age)
7    def isAerobic(heartRate:Int) =
8      heartRate >= minAerobic
9  }
10
```

```
11 trait Activity {
12   val action:String
13   def go:String
14 }
15
16 class Person(val age:Int)
17
18 class Exerciser(age:Int,
19   val action:String = "Running",
20   val go:String = "Run!") extends
21   Person(age) with Activity with Aerobic
22
23 val bob = new Exerciser(44)
24 bob.isAerobic(180) is true
25 bob.isAerobic(80) is false
26 bob.minAerobic is 88.0
```

Traits combine their functionality with another object, as **Exerciser** combines **Aerobic** and **Activity** with a **Person**. Note how the **age** field in **Person** satisfies the abstract **age** field in **Aerobic**. The definitions of **action** and **go** on lines 19-20 must be **val**s (with the same names as the fields in **Activity**) to make them fields in the resulting object, and to thus satisfy the requirements from **Activity**.

Exercises

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

1. Create a **trait WIFI** that reports a status and has an address. Create a **class Camera**, and then another **class WIFICamera** that uses both the **Camera class** and **WIFI trait**. Satisfy the following tests:
   ```
   val webcam = new WIFICamera
   webcam.showImage is "Showing video"
   webcam.address is "192.168.0.200"
   webcam.reportStatus is "working"
   ```

2. Create a **trait Connections** that tells how many connected users there are and limits the number of connections to five. Satisfy the following tests:

```
val c = new Object with Connections
c.maxConnections is 5
c.connect(true) is true
c.connected is 1
for(i <- 0 to 3)
  c.connect(true) is true
c.connect(true) is false
c.connect(false) is true
c.connected is 4
for(i <- 0 to 3)
  c.connect(false) is true
c.connected is 0
c.connect(false) is false
```

3. Using the **Connections trait** from Exercise 2, create a **WIFICamera class** that limits connections to five. Did you have to create any additional classes or methods? Satisfy the following tests:

```
val c2 = new WIFICamera with Connections
c2.maxConnections is 5
c2.connect(true) is true
c2.connected is 1
c2.connect(false) is true
c2.connected is 0
c2.connect(false) is false
```

4. Create a new trait **ArtPeriod**, showing the art era associated with the creation year. Implement it for the following dates, ignoring potential lack of historical accuracy, and satisfy the following tests:

```
// From wikipedia.org/wiki/Art_periods
// Pre-Renaissance: before 1300
// Renaissance: 1300-1599
// Baroque: 1600-1699
// Late Baroque: 1700-1789
// Romanticism: 1790-1880
// Modern: 1881-1970
// Contemporary: after 1971
val art = new ArtPeriod
art.period(1400) is "Renaissance"
art.period(1650) is "Baroque"
art.period(1279) is "Pre-Renaissance"
```

5. Create a class **Painting** by adding in the trait **ArtPeriod**, passing the year into the **Painting** constructor. Satisfy the following test:

```
val painting =
  new Painting("The Starry Night", 1889)
painting.period is "Modern"
```

Tagging Traits & Case Objects

A *tagging trait* groups classes or objects together. The following example is an alternative to the approach in Enumerations. It has benefits and drawbacks compared to that technique – for example, there's no automatic way to iterate through all the types as there is with enumerations (solved by defining **values** on line 9):

```
1  // TaggingTrait.scala
2  import com.atomicscala.AtomicTest._
3
4  sealed trait Color
5  case object Red extends Color
6  case object Green extends Color
7  case object Blue extends Color
8  object Color {
9    val values = Vector(Red, Green, Blue)
10 }
11
12 def display(c:Color) = c match {
13   case Red => s"It's $c"
14   case Green => s"It's $c"
15   case Blue => s"It's $c"
16 }
17
18 Color.values.map(display) is
19 "Vector(It's Red, It's Green, It's Blue)"
```

The hallmark of a tagging trait (**Color**, in this case) is that it only exists to collect types under a common name, thus it typically has no fields or methods. The **sealed** keyword on line 4 tells Scala "there are no subtypes of **Color** other than the ones you see here" (all subtypes of a

sealed class must live in the same source file). Scala warns you that a "match may not be exhaustive" if you don't cover all the cases – try commenting out one of lines 13-15 to see this.

A **case object** is like a **case class** except it produces an **object** instead of a **class**. You get pattern-matching benefits (lines 13-15) and nice output when you convert a **case object** to a **String** (line 19).

Note that the argument to **display** is the tagging trait **Color**. We can refer directly to any of the instances of the **case object**s (lines 13-15).

The **values** field allows iteration through all the **Color**s; you see it used on line 18 (since **display** only takes a single argument, we use the abbreviated form – introduced in Brevity – of the **map** argument). The problem with this approach (solved by Enumerations) is that someone might edit this file and add a new type of **Color** but forget to update **values**.

Exercises

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

1. Add "Purple" to **TaggingTrait.scala**. Don't add to the **match** expression. What happens?
2. Implement **Color** as an **Enumeration** called **EnumColor** for comparison. Satisfy the following tests:
   ```
   EnumColor.Red is "Red"
   EnumColor.Blue is "Blue"
   EnumColor.Green is "Green"
   ```
3. Add another **Red** to **EnumColor**. What happens?
4. Add another **Red** to the tagging trait **Color**. What happens?

Type Parameter Constraints

Let's revisit Enumerations. What if you'd like your enumeration to be a subtype of a trait? If this were a normal class you'd just add the trait to the list of base types during inheritance, but with enumerations you must create a new **Value** type by inheriting from **Val**. This example shows parameterized types with traits, and introduces type constraints, which allow you to impose conditions on type parameters:

```
1  // Resilience.scala
2  import com.atomicscala.AtomicTest._
3
4  trait Resilience
5
6  object Bounciness extends Enumeration {
7    case class _Val() extends Val
8      with Resilience
9    type Bounciness = _Val
10   val level1, level2, level3 = _Val()
11 }
12 import Bounciness._
13
14 object Flexibility extends Enumeration {
15   case class _Val() extends Val
16     with Resilience
17   type Flexibility = _Val
18   val type1, type2, type3 = _Val()
19 }
20 import Flexibility._
21
```

```
22 trait Spring[R <: Resilience] {
23   val res:R
24 }
25
26 case class BouncingBall(res:Bounciness)
27   extends Spring[Bounciness]
28
29 BouncingBall(level2) is
30 "BouncingBall(level2)"
31
32 case class FlexingWall(res:Flexibility)
33   extends Spring[Flexibility]
34
35 FlexingWall(type3) is "FlexingWall(type3)"
```

Here, the tagging trait is **Resilience**, and in order that the enumeration instances of **Bounciness** and **Flexibility** be subtypes of **Resilience**, each **Enumeration** creates a nested subtype of **Val**. Note that both the enumeration instances and the **type** alias must be of the new subtype **_Val**.

Lines 22-24 show a trait with a type parameter **R**. Now, if you say **trait Spring[R]{}**, Scala will accept it but there's nothing much to do with **R** except hold it (*contain* it); thus container types can be quite flexible because they don't particularly care what they hold. However, if you actually want to do something with **R** – say, call a method – then you must somehow determine that **R** is capable, that it has that method available. You must *constrain* **R** using *bounds*.

You're telling Scala, "I want to do something particular to **R**, so **R** must follow these rules." One of the most basic rules is inheritance, expressed by the **<:** symbol that you see on line 22. Here, it says, "**R** must be of type **Resilience** or something inherited from **Resilience**." Equivalently, we say that **Resilience** is the *upper bound* for **R**.

Here, we're only using **R** to declare the field **res** on line 23 so it's of type **R**. But because we know **R** is of type **Resilience**, we can also access any other fields or call methods that happen to be part of **Resilience**. Without the constraint, we can't make any assumptions about **R**.

Our ultimate results here are almost trivial, and just demonstrations: **BouncingBall** and **FlexingWall** each extend **Spring** with their own subtype of **Resilience**. Their **res** field is satisfied by the **res** argument of the case class, but that field is a different subtype in each case.

Here's a slightly more interesting example that uses type constraints to call a method:

```
1  // Constraint.scala
2  import com.atomicscala.AtomicTest._
3
4  class WithF {
5    def f(n:Int) = n * 11
6  }
7
8  class CallF[T <: WithF](t:T) {
9    def g(n:Int) = t.f(n)
10 }
11
12 new CallF(new WithF).g(2) is 22
13
14 new CallF(new WithF {
15   override def f(n:Int) = n * 7
16 }).g(2) is 14
```

The only reason it's possible to call **f** inside **CallF** is that the type is constrained to be **WithF** or a subclass of **WithF**. On line 12 we pass an instance of **WithF**, but on lines 14-16 you see a new trick: It's possible to make an unnamed subclass of **WithF** inline, by following the **new**

WithF with curly braces containing the body of the inherited class. Line 15 overrides **f** to give it a new meaning.

Type parameter constraints can be much more complex than what you see here; it's an algebra of its own that we won't delve into in this book. But you'll see constraints in code so it's good to start getting comfortable with the idea.

After seeing type inference, you might wonder why Scala can't also infer type constraints rather than forcing the programmer to write them out. This should eventually be possible in programming languages (and such languages may already exist), but it's not something we've seen yet in mainstream languages because it's a hard problem to solve (however, at one point, type inference seemed too difficult). One could also make the argument that we must see the type constraints written out in order to understand how to use something; perhaps when type constraint inference becomes available we might change our minds about that.

Exercises

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

1. Modify **House5.scala** from Composition by adding **Enumeration**s for different types of food and utensils. Use type constraints for **Clean** and **Store** as shown in **Resilience.scala**.
2. Modify **Constraint.scala** so **CallF** is a method rather than a class.
3. Create a three-level inheritance hierarchy **Base**, **Derived** and **Most**. Create three methods **f1**, **f2** and **f3** that each take a single object argument, constrained to a different class in the hierarchy. Try passing all different objects to all different methods.

Building Systems with Traits

Because traits are so independent and low-impact, you can break your problem down into pieces that are as numerous and small as necessary. Here's a model of the various ingredients that make up different kinds of ice-cream treats. We use the **Enumeration** subtype technique along with the type constraints introduced in the previous atom:

```
1  // SodaFountain.scala
2  package sodafountain
3
4  object Quantity extends Enumeration {
5    type Quantity = Value
6    val None, Small, Regular,
7      Extra, Super = Value
8  }
9  import Quantity._
10
11 object Holder extends Enumeration {
12   type Holder = Value
13   val Bowl, Cup, Cone, WaffleCone = Value
14 }
15 import Holder._
16
17 trait Flavor
18
19 object Syrup extends Enumeration {
20   case class _Val() extends Val
21     with Flavor
22   type Syrup = _Val
23   val Chocolate, HotFudge,
24     Butterscotch, Caramel = _Val()
```

```
}
import Syrup._

object IceCream extends Enumeration {
  case class _Val() extends Val
    with Flavor
  type IceCream = _Val
  val Chocolate, Vanilla, Strawberry,
    Coffee, MochaFudge, RumRaisin,
    ButterPecan = _Val()
}
import IceCream._

object Sprinkle extends Enumeration {
  case class _Val() extends Val
    with Flavor
  type Sprinkle = _Val
  val None, Chocolate, Rainbow = _Val()
}
import Sprinkle._

trait Amount {
  val quant:Quantity
}

trait Taste[F <: Flavor] extends Amount {
  val flavor:F
}

case class
Scoop(quant:Quantity, flavor:IceCream)
extends Taste[IceCream]

trait Topping

```

```
60 case class
61 Sprinkles(quant:Quantity, flavor:Sprinkle)
62 extends Taste[Sprinkle] with Topping
63
64 case class
65 Sauce(quant:Quantity, flavor:Syrup)
66 extends Taste[Syrup] with Topping
67
68 case class WhippedCream(quant:Quantity)
69 extends Amount with Topping
70
71 case class Nuts(quant:Quantity)
72 extends Amount with Topping
73
74 class Cherry extends Topping
```

Lines 46-52 create the concept of a **Taste** as something that has an **Amount** and a **Flavor**. At first you might wonder why we don't just go straight to creating **Taste** without creating **Amount** as a separate trait, and indeed this makes sense until you see **WhippedCream** and **Nuts**, which each have an **Amount** but no need to vary flavors.

Note that **Flavor** and **Topping** are both tagging traits.

As you analyze this code, keep in mind that we are trying to:

- Create a set of types that fit together in a sensible way.
- Configure it so the compiler catches any misuse of types.
- Eliminate duplicate code.

The last point is worth further consideration. A basic maxim in writing is "shorter sentences are better." Almost anything you do that shortens a sentence will also make it better (simpler, clearer, more active voice, etc.). The parallel maxim in programming is "eliminate code duplication." There's even an acronym: DRY, for *don't repeat*

yourself. And consider Methods, perhaps the most basic tool in programming: they capture common code.

The biggest problem with code duplication is *forking*: you end up with more than one piece of code that does the same thing. Then, when you need to change that functionality – as you always will – you must remember to change it everywhere. And you inevitably forget. You spend time chasing down the bug, and instead of rewriting it to eliminate the code duplication, you just "fix" the duplication. Because you, or your management, is "in a hurry." Or because you are the person who wrote the duplicate code in the first place and you don't think it's so bad.

Code duplication is the most elementary programming offense. If you discover yourself doing it, take the time and effort to root it out. As you focus on it, over time you'll start doing it less and less (and become a better programmer in the process). If you discover someone else doing it, gently point it out to them. If they don't seem to care, see if you can talk them into caring. If you can't do that, then someone needs a different job (either you or them). Otherwise, your life devolves into frustration.

Compile the above code using the shell command:

```
scalac SodaFountain.scala
```

Now we make some ice cream confections:

```
1  // MaltShoppe.scala
2  import com.atomicscala.AtomicTest._
3  import sodafountain._
4  import Quantity._
5  import Holder._
6  import Syrup._
7  import IceCream._
```

```
import Sprinkle._

case class
Scoops(holder:Holder, scoops:Scoop*)

val iceCreamCone = Scoops(
  WaffleCone,
  Scoop(Extra, MochaFudge),
  Scoop(Extra, ButterPecan),
  Scoop(Extra, IceCream.Chocolate))

iceCreamCone is "Scoops(WaffleCone," +
"WrappedArray(Scoop(Extra,MochaFudge), " +
"Scoop(Extra,ButterPecan), " +
"Scoop(Extra,Chocolate)))"

case class MadeToOrder(
  holder:Holder,
  scoops:Seq[Scoop],
  toppings:Seq[Topping])

val iceCreamDish = MadeToOrder(
  Bowl,
  Seq(
    Scoop(Regular, Strawberry),
    Scoop(Regular, ButterPecan)),
  Seq[Topping]())

iceCreamDish is "MadeToOrder(Bowl," +
"List(Scoop(Regular,Strawberry), " +
"Scoop(Regular,ButterPecan)),List())"

case class Sundae(
  sauce:Sauce,
  sprinkles:Sprinkles,
  whipped:WhippedCream,
  nuts:Nuts,
```

```
    scoops:Scoop*) {
    val holder:Holder = Bowl
}

val hotFudgeSundae = Sundae(
    Sauce(Regular, HotFudge),
    Sprinkles(Regular, Sprinkle.Chocolate),
    WhippedCream(Regular), Nuts(Regular),
    Scoop(Regular, Coffee),
    Scoop(Regular, RumRaisin))

hotFudgeSundae is "Sundae(" +
    "Sauce(Regular,HotFudge)," +
    "Sprinkles(Regular,Chocolate)," +
    "WhippedCream(Regular),Nuts(Regular)," +
    "WrappedArray(Scoop(Regular,Coffee), " +
    "Scoop(Regular,RumRaisin)))"
```

Scoops is a basic implementation that allows you to create a cone or dish of ice cream. **MadeToOrder** adds possibilities but is still generic in that it allows any **Seq[Topping]**, while **Sundae** is quite specific in how it describes what a sundae means.

You'll see in the exercises that there are numerous ways to assemble traits and classes into a system. Your final design depends on what works best for you – and you'll find that you don't usually discover "best" (or even "good enough") on your first try, so it's important to keep your structures flexible to easily try new approaches. This flexibility is the deeper part of good design.

Exercises

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

1. Rewrite **Coffee.scala** from Constructors using traits. Satisfy the following tests:

```
Coffee(Single, Caf, Here, Skim, Choc) is
  "Coffee(Single,Caf,Here,Skim,Choc)"
Coffee(Double, Caf,
  Here, NoMilk, NoFlavor) is
  "Coffee(Double,Caf,Here,NoMilk,NoFlavor)"
Coffee(Double,HalfCaf,ToGo,Skim,Choc) is
  "Coffee(Double,HalfCaf,ToGo,Skim,Choc)"
```

2. Assume a latte is a coffee with milk. Create a new class **Latte**. Simplify the **Milk** trait to remove **NoMilk**. **Coffee** should no longer take **Milk** as a class argument. Did you implement **Coffee** as a trait? Why or why not? Satisfy the following tests:

```
val latte = new Latte(Single, Caf,
  Here, Skim)
latte is "Latte(Single,Caf,Here,Skim)"
val usual = new Coffee(Double, Caf, Here)
usual is "Coffee(Double,Caf,Here)"
```

3. A mocha is a variant of a latte, with chocolate. Satisfy the following tests:

```
val mocha = new Mocha(Double,Caf,ToGo,Skim)
mocha is "Mocha(Double,Caf,ToGo,Skim,Choc)"
```

4. Import **sodafountain**, and add a **Container** with **Pint**, **Quart** and **HalfGallon**. Create a **TakeHome** class with arguments of type **Container** and **Flavor**. Satisfy the following tests:

```
TakeHome(Pint, Chocolate) is
  "TakeHome(Pint,Chocolate)"
TakeHome(Quart, Strawberry) is
  "TakeHome(Quart,Strawberry)"
TakeHome(HalfGallon, Vanilla) is
  "TakeHome(HalfGallon,Vanilla)"
```

Sequences

In this book we've used **Vector**s to hold objects. A **Vector** is a *collection* – as the name indicates, it collects objects. More specifically, a **Vector** is a *sequence*, and we now look at another basic sequence, the **List**. You use these without importing anything, as if they are native types in the language.

We've only used the most basic functionality of **Vector**, placing objects in **Vector**s and stepping through them with **for** loops. However, **Vector** has many powerful built-in operations. Here are several of the simpler ones, with explanations embedded in the code as comments. **Vector** and **List** inherit from **Seq** ("sequence") and thus have operations in common, so the method **testSeq** works on both. Note that **testSeq** is written to a specific sequence of values. Also, Scala allows us to call **testSeq** prior to the point where it is defined:

```
1  // SeqOperations.scala
2  import com.atomicscala.AtomicTest._
3
4  testSeq(Vector(1, 7, 22, 11, 17))
5  testSeq(List(1, 7, 22, 11, 17))
6
7  def testSeq(s:Seq[Int]) = {
8    // Is there anything inside?
9    s.isEmpty is false
10   // How many elements inside?
11   s.length is 5
12
13   // Appending to the end:
14   s :+ 99 is Seq(1, 7, 22, 11, 17, 99)
15   // Inserting at the beginning:
16   47 +: s is Seq(47, 1, 7, 22, 11, 17)
17
18   // Get the first element:
```

```
s.head is 1
// Get the rest after the first:
s.tail is Seq(7, 22, 11, 17)
// Get the last element:
s.last is 17
// Get all elements after the 3rd:
s.drop(3) is Seq(11, 17)
// Get all elements except last 3:
s.dropRight(3) is Seq(1, 7)
// Get first 3 elements:
s.take(3) is Seq(1, 7, 22)
// Get final 3 elements:
s.takeRight(3) is Seq(22, 11, 17)
// Section from indices 2 up to 5:
s.slice(2,5) is Seq(22, 11, 17)

// Get value at location 3:
s(3) is 11
// See if it contains a value:
s.contains(22) is true
s.indexOf(22) is 2
// Replace value at location 3:
s.updated(3, 16) is
  Seq(1, 7, 22, 16, 17)
// Remove location 3:
s.patch(3, Nil, 1) is
  Seq(1, 7, 22, 17)

// Append two sequences:
val seq2 = s ++ Seq(99, 88)
seq2 is Seq(1, 7, 22, 11, 17, 99, 88)
// Find the unique values and sort them:
s.distinct.sorted is
  Seq(1, 7, 11, 17, 22)
// Reverse the order:
s.reverse is
```

```
    Seq(17, 11, 22, 7, 1)
  // Find the common elements:
  s.intersect(seq2) is Seq(1,7,22,11,17)
  // Smallest and largest values:
  s.min is 1
  s.max is 22
  // Does it begin or end
  // with these sequences?
  s.startsWith(Seq(1,7)) is true
  s.endsWith(Seq(11,17)) is true
  // Total all the values:
  s.sum is 58
  // Multiply together all the values:
  s.product is 28798
  // "Set" forces unique values:
  s.toSet is Set(1, 17, 22, 7, 11)
}
```

Within **testSeq** we create **Seq** objects whenever we need a sequence to work with either a **Vector** or a **List**, and Scala adapts. This is another form of polymorphism.

The difference between **List** and **Vector** is subtle, and can be confusing. **List** and **Vector** have all their operations in common, but some operations are more efficient for a **List** and others are more efficient for a **Vector**. In general, choose **Vector**, and when you get to the point where you are tuning your program for speed, there are special tools (*profilers*) to tell you where your bottlenecks are (answer: *never where you think*).

Exercises

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

1. Create a **case class** that represents a **Person** in an address book, complete with name and email address. Satisfy the following tests:

```
val p = Person("John", "Smith",
  "john@smith.com")
p.fullName is "John Smith"
p.first is "John"
p.email is "john@smith.com"
```

2. Create three **Person** objects and put them in a **Vector** named **people**. Satisfy the following test: `people.size is 3`

3. Sort the **Vector** of **Person** objects by last name to produce a sorted **Vector**. Hint: Use **sortBy**(_.*fieldname*), where *fieldname* is the field that you want to sort by. Satisfy the following tests:

```
val people = Vector(
Person("Zach","Smith","zach@smith.com"),
Person("Mary", "Add", "mary@add.com"),
Person("Sally", "Taylor",
       "sally@taylor.com"))
val sorted = // call sort here
sorted is "Vector(" +
+ "Person(Mary,Add,mary@add.com)," +
+ "Person(Zach,Smith,zach@smith.com)," +
+ "Person(Sally,Taylor,sally@taylor.com))"
```

4. Move the email address to a **Contact** trait, and mix that in to create a new class **Friend**. Add **Friend** objects to a **Vector**. Sort on the email address. Satisfy the following (this may require refactoring):

```
val friends = Vector(
  new Friend(
    "Zach", "Smith", "zach@smith.com"),
  new Friend(
    "Mary", "Add", "mary@add.com"),
  new Friend(
    "Sally","Taylor","sally@taylor.com"))
val sorted = // call sort here
sorted is "Vector(Mary Add, " +
"Sally Taylor, Zach Smith)"
```

5. What if you want to sort on a primary field (e.g., last name) and resolve any "ties" with a secondary field (e.g., first name)? Hint: sortBy is "stable" so if you sort the list first by the tiebreaker and then by the primary field, you accomplish the goal. Satisfy:

```
val friends2 = Vector(
  new Friend(
    "Zach", "Smith", "zach@smith.com"),
  new Friend(
    "Mary", "Add", "mary@add.com"),
  new Friend(
    "Sally","Taylor","sally@taylor.com"),
  new Friend(
    "Mary", "Smith", "mary@smith.com"))
val s1 = // call first sort here
val s2 = // sort s1 here
s2 is "Vector(Mary Add, Mary Smith, " +
"Zach Smith, Sally Taylor)"
```

6. Sort in a different way than in the previous example. Use the first name as your primary sort key and the last name as your tie breaker. Satisfy the following test:

```
val friends3 = Vector(
  new Friend(
    "Zach", "Smith", "zach@smith.com"),
  new Friend(
    "Mary", "Add", "mary@add.com"),
  new Friend(
    "Sally","Taylor","sally@taylor.com"),
  new Friend(
    "Mary", "Smith", "mary@smith.com") )
val s3 = // call first sort here
val s4 = // sort s1 here
s4 is "Vector(Mary Add, Mary Smith, " +
"Sally Taylor, Zach Smith)"
```

Lists & Recursion

In the previous atom, every operation that **testSeq** performed on a **Vector** also works with a **List**. In almost all cases, choose **Vector** as your sequence container because it performs most operations in the most efficient manner. Sometimes Scala chooses a **List** instead. Here, for example, if we ask for a **Seq**, we get a **List**:

```
scala> Seq(1,3,5,7)
res0: Seq[Int] = List(1, 3, 5, 7)
```

Lists are optimized for a special type of operation, called *recursion*. In recursion, you operate on the first element of the sequence, and then call the same method you're inside (*make a recursive call* or simply *recurse*), passing it the rest of the sequence – that is, the sequence minus the first element. The recursion ends when you run out of elements. Here's a simple example:

```
1  // RecursivePrint.scala
2  def rPrint(s:Seq[Char]):Unit = {
3    print(s.head)
4    if(s.tail.nonEmpty)
5      rPrint(s.tail) // Recursive call
6  }
7
8  rPrint("Recursion")
```

The call to **head** returns the first element, and **tail** produces the rest of the sequence minus the first element. With each recursion, the sequence passed to **rPrint** gets smaller until **nonEmpty** becomes false when there's nothing left, at which point the recursion ends. The string passed to **rPrint** on line 8 automatically becomes a **Seq**.
Note the explicit return type on line 2, which Scala requires for a recursive method.

Recursion is often used for calculations on a sequence. For example, when summing a sequence you can avoid a variable (**var**) by creating the sum piece-by-piece during the recursion. Here's a recursive method that takes a list to sum and an integer to hold the sum:

```
1  // RecursiveSum.scala
2  import com.atomicscala.AtomicTest._
3
4  def sumIt(toSum:List[Int], sum:Int=0):Int =
5    if(toSum.isEmpty)
6      sum
7    else
8      sumIt(toSum.tail, sum + toSum.head)
9
10 sumIt(List(10, 20, 30, 40, 50)) is 150
```

The top-level call to **sumIt** on line 10 uses the default value of zero for **sum**. If the list isn't empty, line 8 adds **sum** to the **head** of **toSum**, then calls the method again, passing **tail** as the new list to sum. The method recurses until it hits the end of the list (becomes empty), then returns **sum**. Here's what happens in detail for the call on line 10:

sumIt is called with the **List(10, 20, 30, 40, 50)** and a **sum** of 0. The list is not empty, so we add **sum** to the **head** (10), calculating 0+10 = 10.
sumIt is then called with the **List(20, 30, 40, 50)** and a **sum** of 10. The list is not empty, so we add **sum** to the **head** (20), calculating 10+20 = 30.
sumIt is then called with the **List(30, 40, 50)** and a **sum** of 30. The list is not empty, so we add **sum** to the **head** (30), calculating 30+30=60.
sumIt is then called with the **List(40, 50)** and a **sum** of 60. The list is not empty, so we add **sum** to the **head** (40), calculating 60+40=100.
sumIt is then called with the **List(50)** and a **sum** of 100. The list is not empty, so we add **sum** to the **head** (50), calculating 100+50=150.

sumIt is then called with an empty list. Because the list is empty, we return 150.

Before you write a recursive method on a **List**, consider whether one might already be available. Scala collections have a built-in **sum**, so instead of writing **sumIt**, you say:

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

List(10, 20, 30, 40, 50).sum is 150
Vector(10, 20, 30, 40, 50).sum is 150
Seq(10, 20, 30, 40, 50).sum is 150
Set(10, 20, 30, 40, 50, 50, 50).sum is 150
(10 to 50 by 10).sum is 150
```

Recursion can be a little tricky, and it's only occasionally useful. In those cases, because a **List** is built as a head and a tail, it's ideally suited for recursion.

Exercises

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

1. Write a recursive method **max** to find the maximum value in a **List**, without using **List**'s **max** method. Satisfy the following tests:
   ```
   val aList = List(10, 20, 45, 15, 30)
   max(aList) is 45
   ```
2. Add **println** statements to **RecursiveSum.scala** to trace what happens during recursion.

3. In map and reduce, you implemented a method **sumIt** that used **reduce** to do a summation. There, you used a variable argument list. Reimplement using a **List**. Compare this to your solution for Exercise 1, above. Satisfy the following tests:
```
sumIt(List(1, 2, 3)) is 6
sumIt(List(45, 45, 45, 60)) is 195
```

4. In Reaching into Java, we used a math library method **Frequency** to calculate the frequency of "cat" in a **List** of animals. Use a recursive method to do the same thing. Satisfy the following tests:
```
calcFreq(animalList, "cat") is 4
calcFreq(animalList, "dog") is 1
```

Combining Sequences with zip

It's often useful to take two sequences and pair them up; this is called "zipping" because it mimics the behavior of the zipper on your jacket:

```
1  // Zipper.scala
2  import com.atomicscala.AtomicTest._
3
4  val left = Vector("a", "b", "c", "d")
5  val right = Vector("q", "r", "s", "t")
6
7  left.zip(right) is
8  "Vector((a,q), (b,r), (c,s), (d,t))"
9
10 left.zip(0 to 4) is
11 "Vector((a,0), (b,1), (c,2), (d,3))"
12
13 left.zipWithIndex is
14 "Vector((a,0), (b,1), (c,2), (d,3))"
```

On line 7 we combine **left** with **right**. The result is a **Vector** of tuples pairing each element from **left** with each element in **right**.

Line 10 combines **left** with the **Range 0 to 4**, also a sequence. What if you want to put an index on each element in a sequence? There's a dedicated method for this, **zipWithIndex**, shown on line 13.

Here's a method that puts the numbers as the first tuple element instead of the second (a clever functional programmer could probably find a way to do this from the output of **zipWithIndex**):

```
1  // IndexWithZip.scala
2  import com.atomicscala.AtomicTest._
```

```
3
4  def number(s:String) =
5    Range(0, s.length).zip(s)
6
7  number("Howdy") is
8  Vector((0,'H'), (1,'o'), (2,'w'),
9         (3,'d'), (4,'y'))
```

Notice that zipping with a **String** automatically breaks the **String** into its component characters.

We finish with an example that combines **zip** and **map** to give you a taste of what's possible with functional programming:

```
1  // ZipMap.scala
2  import com.atomicscala.AtomicTest._
3
4  case class Person(name:String, ID:Int)
5  val names = Vector("Bob", "Jill", "Jim")
6  val IDs = Vector(1731, 9274, 8378)
7
8  names.zip(IDs).map {
9    case (n, id) => Person(n, id)
10 } is "Vector(Person(Bob,1731), " +
11 "Person(Jill,9274), Person(Jim,8378))"
```

Line 8 uses **zip** to produce a sequence of name-id tuples, which it then passes to **map**. The **map** method can apply a function, but it can also apply a **match** clause (without the need to say **match**) as seen here. The **case** statement unpacks each tuple and passes the values to the **Person** constructor. The result is a **Vector** of initialized objects.

Note the succinctness of the expression on lines 8-10. As you become fluent in the functional programming style, you'll write concise expressions like this, and because you know the built-in methods like

zip and **map** are correct, it gives you confidence that the combined expression is also correct.

Exercises

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

1. Write code to pair people up to do exercises in a programming seminar. Take the list of attendees and split it into two lists. Use **zip** to create the pairs. Satisfy the following tests:

```
val people = Vector("Sally Smith",
  "Dan Jones", "Tom Brown", "Betsy Blanc",
  "Stormy Morgan", "Hal Goodsen")
val group1 = // fill this in
val group2 = // fill this in
val pairs = // fill this in
pairs is Vector(
  ("Sally Smith","Betsy Blanc"),
  ("Dan Jones","Stormy Morgan"),
  ("Tom Brown","Hal Goodsen"))
```

2. What happens when the initial list is an odd number, so the groups split into uneven sizes? Try it.

3. Repeat Exercise 1 using a **List** instead of a **Vector**. Did you have to make any other modifications?

4. Taking a similar approach as **ZipMap.scala**, modify **IndexWithZip.scala** to use the result of **zipWithIndex**.

Sets

A **Set** ensures that it contains only one element of each value, so it automatically removes duplicates. The most common thing to do with a **Set** is to test for membership using the **()** operator:

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

val set =
  Set(1, 1, 2, 3, 9, 9, 4, 22, 11, 7, 6)
// No duplicates:
set is Set(1, 6, 9, 2, 22, 7, 3, 11, 4)

// Set membership:
set(9) is true
set(99) is false

// Is this set contained within another?
Set(1, 6, 9, 2).subsetOf(set) is true

// Two different versions of set union:
set.union(Set(2, 3, 4, 99)) is
  Set(1, 6, 9, 2, 22, 7, 3, 11, 99, 4)
set | Set(2, 3, 4, 99) is
  Set(1, 6, 9, 2, 22, 7, 3, 11, 99, 4)

// Set intersection:
set & Set(0,1,11,22,87) is Set(1,22,11)
set intersect Set(0,1,11,22,87) is
  Set(1,22,11)

// Set difference:
set &~ Set(0, 1, 11, 22, 87) is
  Set(6, 9, 2, 7, 3, 4)
set -- Set(0, 1, 11, 22, 87) is
```

```
31    Set(6, 9, 2, 7, 3, 4)
```

Many of the operations in **Vector** and **List** also show up in **Set**; here we only show some that are unique to **Set**.

Line 7 shows that placing duplicate items into a **Set** automatically removes the duplicates. Lines 10 and 11 use the **()** operator to test for membership. You can also perform the usual Venn-diagram operations like checking for subset, union, intersection and difference. You can either use operators (like **&**) or their equivalent descriptive names (like **intersect**).

If you have a sequence and you want to remove duplicates, use **toSet** to convert it to a **Set**:

```
1   // RemoveDuplicates.scala
2   import com.atomicscala.AtomicTest._
3
4   val ch = for(i <- 0 to 2) yield 'a' to 'd'
5   ch is "Vector(NumericRange(a, b, c, d), " +
6     "NumericRange(a, b, c, d), " +
7     "NumericRange(a, b, c, d))"
8
9   ch.flatten is "Vector(a, b, c, d, " +
10    "a, b, c, d, a, b, c, d)"
11
12  ch.flatten.toSet is "Set(a, b, c, d)"
```

The comprehension on line 4 yields three copies of **'a' to 'd'**. But notice lines 5-7, which show that this is actually a **Vector** holding three containers rather than just the letters tossed into the **Vector**. This issue of producing a container of containers, instead of just a container of the things you want, happens so often that there's a method **flatten** (for virtually all sequences) to smash everything into a single-level sequence. Notice the effect on lines 9-10, and that the

result has duplicate entries. Now if we apply **toSet**, the result is a **Set** with no duplicates.

Exercises

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

1. Create sets for fruits, vegetables, and meats. Create a grocery list, and calculate what percentage of your list is in each category, including an "other" category determined by not matching any category. Satisfy the following tests:

```
val fruits = Set("apple", "orange",
  "banana", "kiwi")
val vegetables = Set("beans", "peas",
  "carrots", "sweet potatoes",
  "asparagus", "spinach")
val meats = Set("beef", "chicken")
val groceryCart = Set("apple",
  "pretzels", "bread", "orange", "beef",
  "beans", "asparagus", "sweet potatoes",
  "spinach", "carrots")
percentMeat(groceryCart) is 10.0
percentFruit(groceryCart) is 20.0
percentVeggies(groceryCart) is 50.0
percentOther(groceryCart) is 20.0
```

2. Using your solution for Exercise 1, add a set for protein that includes the set for meats, and an additional set for vegetarian proteins. Satisfy the following tests:

```
val vegetarian = Set("kidney beans",
  "black beans", "tofu")
val groceryCart2 = Set("apple",
  "pretzels", "bread", "orange", "beef",
  "beans", "asparagus", "sweet potatoes",
  "kidney beans", "black beans")
percentMeat(groceryCart2) is 10.0
```

```
percentVegetarian(groceryCart2) is 20.0
percentProtein(groceryCart2) is 30.0
```

3. Write code that produces a container of containers of containers. Use **flatten** to reduce your container to a single-level sequence. Hint: you may want to do this in multiple steps. Satisfy the following tests:

```
val box1 = Set("shoes", "clothes")
val box2 = Set("toys", "dishes")
val box3 = Set("toys", "games", "books")
val attic = Set(box1, box2)
val basement = Set(box3)
val house = Set(attic, basement)
Set("shoes", "clothes", "toys",
  "dishes") is attic.flatten
Set("toys", "games", "books") is
  basement.flatten
Set("shoes", "clothes", "toys",
  "dishes", "games", "books") is
/* fill this in -- call flatten */
```

Maps

History leaves us with some slightly confusing terminology. The **map** *operation* described in map and reduce is quite different from the **Map** *class*, which connects *keys* to *values*. A **Map** looks up a value when given a key. You create a **Map** by giving it a set of key-value pairs, where each key is separated from its associated value by an arrow, as seen on lines 4-5:

```
1  // Maps.scala
2  import com.atomicscala.AtomicTest._
3
4  val constants = Map("Pi" -> 3.141,
5    "e" -> 2.718, "phi" -> 1.618)
6
7  Map(("Pi", 3.141), ("e", 2.718),
8    ("phi", 1.618)) is constants
9
10 Vector(("Pi", 3.141), ("e", 2.718),
11   ("phi", 1.618)).toMap is constants
12
13 // Look up a value from a key:
14 constants("e") is 2.718
15
16 constants.keys is "Set(Pi, e, phi)"
17
18 constants.values is
19 "MapLike(3.141, 2.718, 1.618)"
20
21 // Iterate through key-value pairs:
22 (for(pair <- constants)
23   yield pair.toString) is
24 "List((Pi,3.141), (e,2.718), (phi,1.618))"
25
```

```
26 // Unpack during iteration:
27 (for((k,v) <- constants)
28   yield k + ": " + v) is
29 "List(Pi: 3.141, e: 2.718, phi: 1.618)"
```

Lines 7-8 show that a **Map** can also be initialized using a comma-separated list of tuples. Lines 10-11 take this one step further by creating a **Vector** of tuples, then converting *that* to a **Map**.

With a **Map**, the **()** operator is used for lookup (line 14). You can get all the keys with the **keys** method and all the values with the **values** method. **Map**'s **keys** method produces a **Set** because all keys in a **Map** must already be unique (otherwise you'd have ambiguity during a lookup). **MapLike** is another sequence, so you can iterate through it using a **for** loop, for example.

Iterating through the **Map** itself produces key-value pairs as tuples, as on line 22. Because these are tuples, you can unpack them as you iterate, shown on line 27.

You can store class objects as values in a **Map**. Here, we create some pets using the **Name** trait defined in A Little Reflection:

```
1  // PetMap.scala
2  import com.atomicscala.AtomicTest._
3  import com.atomicscala.Name
4
5  trait Pet extends Name
6  class Bird extends Pet
7  class Duck extends Bird
8  class Cat extends Pet
9  class Dog extends Pet
10
11 val petMap = Map("Dick" -> new Bird,
12   "Carl" -> new Duck, "Joe" -> new Cat,
13   "Tor" -> new Dog)
```

```
14
15 petMap.keys is
16  Set("Dick", "Carl", "Joe", "Tor")
17  petMap.values.toVector is
18  "Vector(Bird, Duck, Cat, Dog)"
```

It's also possible to use class objects as keys in a **Map**, but that's trickier and is beyond the scope of this book.

Maps look like simple little databases. Although they are quite limited compared to a full-featured database, they are nonetheless remarkably useful (and far more efficient than a database).

Exercises

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

1. Modify **Maps.scala** so the numbers are keys and the strings are values.

2. **Map**s store information using unique keys. An email address can serve as a unique key. Create a class **Name** containing **firstName** and **lastName**. Create a **Map** that associates **emailAddress** (a **String**) with **Name**. Satisfy the following test:
```
val m = Map("sally@taylor.com"
  -> Name("Sally","Taylor"))
m("sally@taylor.com") is
  Name("Sally", "Taylor")
```

3. Adding to your solution for the previous exercise, add Jiminy Cricket to the existing map, where the email address is "jiminy@cricket.com." Satisfy the following tests:
```
m2("jiminy@cricket.com") is
Name("Jiminy", "Cricket")
m2("sally@taylor.com") is
Name("Sally", "Taylor")
```

4. **Map** keys must be distinct values. Create a **Map** with keys for the following languages: English, French, Spanish, German, and Chinese. What happens when you try to add Turkish?

5. Adding to your solution for the previous exercise, try to add a language that already exists to the **Map** (for example, French). Write tests to show what happens.

6. Remove "Spanish" from your **Map** in Exercise 4. Remove "jiminy@cricket.com" from the **Map** in Exercise 3. Write tests to show the removals.

7. Case classes *can* be used as keys for **Map**s. Create a class for **Person(name:String)**. Create a mapping of **Person** to **String**. Remove the **case** keyword and see what error message(s) you get. Fix it and satisfy the following test:
   ```
   m(Person("Janice")) is "CFO"
   ```

References & Mutability

We've said that **var**s can be changed while **val**s cannot. This is an oversimplification. Consider:

```
1  // ChangingAVal.scala
2  import com.atomicscala.AtomicTest._
3
4  class X(var n:Int)
5  val x = new X(11)
6  x.n is 11
7  x.n = 22
8  x.n is 22
9  // x = new X(22) // Not allowed
```

Although **x** is a **val**, its object can be modified. The **val** prevents it from being reassigned to a new object. Similarly:

```
1  // AnUnchangingVar.scala
2  import com.atomicscala.AtomicTest._
3
4  class Y(val n:Int)
5  var y = new Y(11)
6  y.n is 11
7  // y.n = 22 // Not allowed
8  y = new Y(22)
```

Even though **y** is a **var**, its object cannot be modified. However, **y** can be reassigned to a new object.

We've talked about identifiers like **x** and **y** as if they were objects. However, they actually just refer to objects – **x** and **y** are called *references*. One way to see this is to observe that two identifiers can reference the same object:

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

class Z(var n:Int)
var z1 = new Z(13)
var z2 = z1
z2.n is 13
z1.n = 97
z2.n is 97
```

When **z1** modifies the object, **z2** sees the modification. Print **z1** and **z2** to see that they produce the same address.

Thus, **var** and **val** control references rather than objects. A **var** allows you to rebind a reference to a different object, and a **val** prevents you from doing so.

Mutability

Mutability means an object can change its state. In the examples above, **class X** and **class Z** create mutable objects while **class Y** creates immutable objects.

Many classes in the Scala standard library are immutable by default but also have mutable versions. When you ask for a plain **Map**, it's immutable:

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

val m = Map(5->"five", 6->"six")
m(5) is "five"
// m(5) = "5ive" // Fails
m + (4->"four") // Doesn't change m
m is Map(5 -> "five", 6 -> "six")
val m2 = m + (4->"four")
```

```
10  m2 is
11  Map(5 -> "five", 6 -> "six", 4 -> "four")
```

Line 4 creates a **Map** associating **Int**s with **Strings.** If we try to replace a **String** as on line 6, we see:

```
value update is not a member of
scala.collection.immutable.Map[Int,String]
```

An immutable **Map** doesn't include an **=** operator.

Lines 7-8 show that **+** just creates a new **Map** that includes both the old elements and the new one, but doesn't affect the original **Map**. Immutable objects are "read-only." The only way to add an element is to create a new **Map** as in line 9.

Scala's collections are immutable by default, meaning that if you don't explicitly say you want a mutable collection, you won't get one. Here's how you create a mutable **Map**:

```
1   // MutableMaps.scala
2   import com.atomicscala.AtomicTest._
3   import collection.mutable.Map
4
5   val m = Map(5 -> "five", 6 -> "six")
6   m(5) is "five"
7   m(5) = "5ive"
8   m(5) is "5ive"
9   m += 4 -> "four"
10  m is
11  Map(5 -> "5ive", 4 -> "four", 6 -> "six")
12  // Can't reassign val m:
13  // m = m + (3->"three")
```

Notice that, once we import the mutable version of **Map**, the default **Map** becomes the mutable one – we define a **Map** on line 5 without

qualification. Line 7 modifies an element of the **Map**. Line 9 adds a key-value pair to the map.

Exercises

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

1. Create a **var** reference to an immutable **Map** and demonstrate what this means (prove you can't change its contents, nor append to it. Show that you can rebind the reference). Now create a **val** reference to a mutable **Map** and demonstrate what this means.
2. Show the difference between a mutable and immutable **Set**.
3. Show the difference between a mutable and immutable **List**.
4. **Vector** doesn't have a mutable equivalent. How do you change the contents of a **Vector**?
5. We don't declare method arguments as **var** or **val**. Discover whether Scala uses **var** or **val** method arguments by creating a simple class, then a method that takes an argument of that class. Inside the method, attempt to rebind the argument to a new object and observe the error message.
6. Create a class containing a **var** field. Write a method that takes an argument of this class. Inside the method, modify the **var** field to see if your method has side effects.
7. Create a class that has both mutable and immutable fields. Is the resulting class mutable or immutable?

Pattern Matching with Tuples

Consider an **Enumeration** that represents paint colors:

```
// PaintColors.scala
package paintcolors

object Color extends Enumeration {
  type Color = Value
  val red, blue, yellow, purple,
    green, orange, brown = Value
}
```

We'll create a method **blend** to show the resulting color when you combine two colors. Conveniently, you can pattern match on a tuple:

```
// ColorBlend.scala
import paintcolors.Color
import paintcolors.Color._

package object colorblend {

def blend(a:Color, b:Color) =
  (a, b) match {
    case _ if a == b => a
    case (`red`, `blue`) |
         (`blue`, `red`) => purple
    case (`red`, `yellow`) |
         (`yellow`, `red`) => orange
    case (`blue`, `yellow`) |
         (`yellow`, `blue`) => green

    case (`brown`, _) |
```

```
17            (_, `brown`) => brown
18        case _ => // Interesting, not accurate:
19            Color((a.id + b.id) % Color.maxId)
20    }
21
22  }
```

To put **blend** inside a package, we introduce a shorthand: the **package object** on line 5, which not only creates the **colorblend** object but simultaneously makes it a **package**.

In line 8 a tuple is pattern-matched, like any other type. The first **case** on line 9 says, "if the colors are identical, the output is the same."

Lines 10-15 show that a **case** can also be a tuple, and that cases can be ORed together using the single '|', the *short-circuiting* OR. "Short circuiting" means that if you have a chain of expressions ORed together, the first expression that succeeds stops the evaluation of the chain (since it's an OR, all you need is one **true** to make the entire expression **true**). The short-circuit single '|' is the only OR allowed in **case** statements.

There's something else odd here. On the left side of the "rocket" (but *not* the right side), we've put single back-quotes (sometimes called "backticks") on all the color names. This is an idiosyncrasy of **case** statements: if it sees a non-capitalized name on the left side of a rocket, it creates a local variable for calculating the pattern match. We've intentionally not capitalized the values of **Color** to show this issue. If you put backticks around the name, it tells Scala to treat it as a symbol.

Lines 16-17 say, "if **brown** is involved, the result will always be **brown**, regardless of the other color." It uses the wildcard '_' as the other color.

So far, this method is only an approximation to the colors you'll produce (paint quantities have an effect as well). Lines 18-19 are just an attempt to produce an interesting but inaccurate result for all the other possibilities. The ordinal values (**id**) of each color are summed. The maximum **id** is used as a modulus to force the calculation result within the available **id** values, and that result is used to index into **Color** and produce a new value.

Here are some tests for **blend**:

```
1  // ColorBlendTest.scala
2  import com.atomicscala.AtomicTest._
3  import paintcolors.Color._
4  import colorblend.blend
5
6  blend(red, yellow) is orange
7  blend(red, red) is red
8  blend(yellow, blue) is green
9  blend(purple, orange) is blue
10 blend(purple, brown) is brown
```

While the output of most of these tests is plausible, line 9 is obviously produced by the last **case** in **blend**.

Producing output from a tuple of inputs is often called a *table lookup*. Using a **Map**, we solve the problem a second way, by generating the table ahead of time instead of calculating the result every time. Sometimes this is a more useful approach.

Here, we populate a **Map** using **colorblend.blend**:

```
1  // ColorBlendMap.scala
2  import com.atomicscala.AtomicTest._
3  import paintcolors.Color
4  import paintcolors.Color._
5
```

```
6  val blender = (
7    for {
8      a <- Color.values.toSeq
9      b <- Color.values.toSeq
10     c = colorblend.blend(a, b)
11   } yield ((a, b), c)
12 ).toMap
13
14 blender.foreach(println)
15
16 def blend(a:Color,b:Color) = blender((a,b))
17
18 blend(red, yellow) is orange
19 blend(red, red) is red
20 blend(yellow, blue) is green
21 blend(purple, orange) is blue
22 blend(purple, brown) is brown
```

To initialize **blender**, we create a sequence of tuples of two elements: the first element is a tuple of the two input colors, and the second element is the resulting blended color. A **Map** is created from the sequence of tuples using **toMap**.

Line 8 iterates through each of the **Color** values, and for each of those values, line 9 iterates through all the **Color** values again, generating all possible combinations of two inputs, which are then blended using **colorblend.blend**. Line 14 displays each **Map** key-value pair for verification, line 16 produces a new version of **blend** by creating a tuple to pass to the **Map**, and then we apply the same tests as before.

Exercises

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

1. Remove the backticks from one of the labels in **ColorBlend.scala** and see what error message is produced.

2. Remove the default case from **ColorBlend.scala**. Satisfy the following tests:
```
blend(red, yellow) is orange
blend(red, red) is red
blend(yellow,blue) is green
```

3. Add another color (**magenta**) to **PaintColors.scala** and verify that the rest of the examples in the atom still work correctly. Satisfy the following tests:
```
blend2(red, yellow) is orange
blend2(red, red) is red
blend2(yellow,blue) is green
blend2(yellow, magenta) is purple
blend2(red, magenta) is purple
```

4. Building on your solution for the previous exercise, add the color **white** to **PaintColors.scala**. In the match expression, return the "other" color whenever **white** is blended with it. Satisfy the tests:
```
blend3(red, yellow) is orange
blend3(red, red) is red
blend3(yellow,blue) is green
blend3(yellow, magenta) is purple
blend3(red, magenta) is purple
blend3(purple, white) is purple
blend3(white, red) is red
```

Error Handling with Exceptions

Improved error reporting is one of the most powerful ways to increase the reliability of your code. Error reporting is especially important in Scala, where one of the primary goals is to create program components for others to use. To create a robust system, each component must be robust. With consistent error reporting, components can reliably communicate problems to client code.

Ideally, you catch errors when Scala analyzes your program, before it runs. Many errors cannot be detected this way, and must be handled at run time by producing error information from methods.

The next few atoms explore "what to do when things go wrong." It turns out that handling errors doesn't have a single obvious solution; indeed, the topic continues to evolve. We look at the multiple approaches available in Scala and their appropriate use, starting with *exceptions*.

The word "exception" is used in the same sense as the phrase "I take exception to that." An exceptional condition prevents the continuation of the current method or scope. At the point the problem occurs, you might not know what to do with it, but you do know that you cannot continue. You don't have enough information in the current context to fix the problem. So you must stop, and hand the problem to an outside context where someone is qualified to take appropriate action.

It's important to distinguish an exceptional condition from a normal problem, in which you have enough information in the current context to cope with the difficulty somehow. With an exceptional

condition, you cannot continue processing. All you can do is jump out of that context and relegate that problem to a higher context. This is what happens when you throw an exception.

An exception is an object "thrown" from the site of the error, and can be "caught" by an appropriate *exception handler* that matches that type of error.

Division is a simple example. If you're about to divide by zero, it's worth checking for that condition. But what does it mean that the denominator is zero? Maybe you know, in the context of the problem you're trying to solve in that particular method, how to deal with a zero denominator. If it's an unexpected value, however, you cannot continue along that execution path. One solution is to throw an exception – escape and force some other part of the code to manage the issue.

Here's a basic example showing the configuration and use of exceptions:

```
1  // DivZero.scala
2  import com.atomicscala.AtomicTest._
3
4  class Problem(val msg:String)
5    extends Exception
6
7  def f(i:Int) =
8    if(i == 0)
9      throw new Problem("Divide by zero")
10   else
11     24/i
12
```

```
13 def test(n:Int) =
14   try {
15     f(n)
16   } catch {
17     case err:Problem =>
18       s"Failed: ${err.msg}"
19   }
20
21 test(4) is 6
22 test(5) is 4 // Integer truncation
23 test(6) is 4
24 test(0) is "Failed: Divide by zero"
25 test(24) is 1
26 test(25) is 0 // Also truncation
```

Scala inherits many exception types from Java, but defines hardly any of its own. You define a custom exception by inheriting from the **Exception** class (lines 4-5).

The **f** method doesn't know what to do with an argument of zero, so it throws an exception on line 9 by creating a new **Problem** exception and throwing it via the **throw** keyword. When you throw an exception, the current path of execution (the one you can't continue) stops and the exception object ejects from the current context. At this point, the exception-handling mechanism takes over and begins to look for an appropriate place to continue executing the program. Execution ends up in the exception handler.

The **test** method shows how to set up an exception handler. You begin with the **try** keyword, followed by a block of code containing expressions that can throw exceptions. Note that the **try** block is an expression; if successful, its result is returned from **test**.

This is followed by the exception handler: the **catch** keyword and a sequence of **case** statements matching all the different types of

exceptions you are prepared to handle (here we only match a **Problem** exception). If an exception isn't handled at this level, it continues to move out to higher levels, searching for a matching handler. If it finds a handler, the search stops. If it never finds a handler, it aborts the program and prints a long and noisy *stack trace*, detailing where it came from. To see stack traces in the REPL, enter:

```
scala> throw new Exception
scala> throw new Exception("Disaster!")
```

One of the most important aspects of exceptions is that if something bad happens, they allow you to (if nothing else) force the program to stop and tell you what went wrong, or (ideally) force the programmer to deal with the problem and return the program to a stable state.

Often a method generates more than one type of exception – that is, it has several ways to fail. For reuse, the following is in a **package**:

```
1  // Errors.scala
2  package errors
3
4  case class Except1(why:String)
5    extends Exception(why)
6  case class Except2(n:Int)
7    extends Exception(n.toString)
8  case class Except3(msg:String, d:Double)
9    extends Exception(s"$msg $d")
10
11 object toss {
12   def apply(which:Int) =
13     which match {
14       case 1 => throw Except1("Reason")
15       case 2 => throw Except2(11)
16       case 3 =>
17         throw Except3("Wanted:", 1.618)
18       case _ => "OK"
```

```
    }
}
```

Each **Exception** subtype passes a **String** to the base-class **Exception** constructor; that **String** becomes the message returned by **Exception**'s **getMessage** method.

When we compile Scala code, we can't have free-standing methods as we can with scripts, thus we create **toss** as an object with an **apply** method so it *looks* like a standalone method when we use it (we could instead have imported a named method):

```
// MultipleExceptions.scala
import com.atomicscala.AtomicTest._
import errors._

def test(which:Int) =
  try {
    toss(which)
  } catch {
    case Except1(why) => s"Except1 $why"
    case Except2(n) => s"Except2 $n"
    case Except3(msg, d) =>
      s"Except3 $msg $d"
  }

test(0) is "OK"
test(1) is "Except1 Reason"
test(2) is "Except2 11"
test(3) is "Except3 Wanted: 1.618"
```

Every time you call **toss**, you must **catch** the exceptions it emits if those exceptions are germane to the result (otherwise, you let them "bubble up" to be caught elsewhere).

Exceptions are essential for interacting with Java libraries, because Java uses exceptions for everything – both exceptional conditions and ordinary errors. For this reason, much of your exception code will occur when using Java libraries, rather than dealing with truly exceptional conditions. In the next atom, you'll see an example that captures exceptions from a Java library.

Although Scala includes language support for exception handling, you'll see in subsequent atoms that it tends to emphasize other forms of error handling and reserves exceptions for situations where you really don't know what else to do. Indeed, it's important to understand that there are at least two kinds of error conditions: expected and exceptional, and that you must respond to these conditions in different ways. If you treat every error condition as an exception, your code will get messy indeed.

Exercises

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

1. Create a method that throws an object of class **Exception** inside a **try** block. Pass a **String** argument to the constructor. Catch the exception inside a **catch** clause and test the **String** argument.

2. Create a class with a simple method **f**. Create a **var** of that class and initialize it to the special pre-defined value **null**, which means "nothing." Try to call **f** using this **var**. Now wrap the call in a **try-catch** clause to catch the exception.

3. Create a **Vector** containing some elements. Try to index outside the range of that **Vector**. Now write code to catch the exception.

4. Inherit your own subclass of **Exception**. Write a constructor for this class that takes a **String** argument and stores it inside the base-class **Exception** object. Write a method that displays the stored **String**. Create a **try-catch** clause to test your new exception class.

5. Create three new subtypes of **Exception**. Write a method that throws all three. In another method, call the first method but only use a single catch clause to catch all three types of exception.

6. Create a class with two methods, **f** and **g**. In **g**, throw a new type of exception that you define. In **f**, call **g**, catch its exception and, in the **catch** clause, throw a different exception (of a second type that you define). Test your code.

7. Demonstrate that a derived-class constructor cannot catch exceptions thrown by its base-class constructor.

8. Create a class called **FailingConstructor** with a constructor that can fail partway through the construction process and throw an exception. In another method, write code that properly guards against this failure.

9. Create a three-level inheritance hierarchy of exceptions. Now create a base class **A** with a method **f** that throws the exception at the base of your hierarchy. Inherit **B** from **A** and override **f** so it throws the exception at level two of your hierarchy. Repeat by inheriting class **C** from **B**. Create a **C** and assign it to an **A** (this is called "upcasting"), then call **f**.

10. The exception-handling mechanism includes another keyword, **finally**. A **finally** clause is executed regardless of what happens in the **try** or **catch** clauses. A **finally** clause can directly follow a **try** clause (with no **catch**) or it can be placed after a **catch** clause. Demonstrate that the **finally** clause always executes.

Constructors & Exceptions

Constructors are special because they create objects. A **new** expression cannot return anything except a newly-created object, so if construction fails we can't just return an error. Returning a partially-constructed object is not a useful option, either, because the client programmer could easily assume that the object is OK.

There are two basic approaches:

1. Write a constructor so simple it cannot fail. While ideal, this is often not convenient or simply not possible.

2. Throw exceptions for failure. Since you can't produce a return value, and you don't want to produce a badly-created object, this seems like the only choice.

Companion objects give us a third possibility: since **apply** is usually written as a factory to generate new objects, and it's a method rather than a constructor, we can return error information from **apply**.

Here's a class that opens a source-code file and turns it into a **Vector**-like container, so you can index any line as well as iterate through the code listing, and perform other operations provided by Scala containers. The **apply** captures exceptions and converts them to error messages, which are also stored in the container. Thus, **apply** always returns a **Vector[String]** which can then be treated uniformly:

```
1  // CodeListing.scala
2  package codelisting
3  import java.io.FileNotFoundException
4
5  class ExtensionException(name:String)
6    extends Exception(
7      s"$name doesn't end with '.scala'")
8
9  class CodeListing(val fileName:String)
10 extends collection.IndexedSeq[String] {
11   if(!fileName.endsWith(".scala"))
12     throw new ExtensionException(fileName)
13   val vec = io.Source.fromFile(fileName)
14     .getLines.toVector
15   def apply(idx:Int) = vec(idx)
16   def length = vec.length
17 }
18
19 object CodeListing {
20   def apply(name:String) =
21     try {
22       new CodeListing(name)
23     } catch {
24       case _:FileNotFoundException =>
25         Vector(s"File Not Found: $name")
26       case _:NullPointerException =>
27         Vector("Error: Null file name")
28       case e:ExtensionException =>
29         Vector(e.getMessage)
30     }
31 }
```

The **String** argument passed to the **Exception** constructor on lines 6-7 becomes the message for our new exception type.

To create a container-like class that holds each line as a **String**, we inherit from **collection.IndexedSeq[String]**, which installs all the necessary mechanisms. To actually hold the lines, we use composition with an ordinary **Vector** generated by Scala's **io.Source.fromFile** method. This opens and reads the file, then **getLines** turns it into a sequence of lines, and finally **toVector** converts the sequence into a **Vector**.

When you inherit from **IndexedSeq**, you must define **apply** and **length** (otherwise you get error messages telling you to do so). However, that's everything necessary to produce a new type of container (in comparison to numerous other languages, this is remarkably straightforward).

On line 13 we use a name in only one place, so we fully qualify it rather than using an **import**.

Even though **io.Source.fromFile** is part of the Scala standard library, it uses elements from Java that throw the Java exceptions **FileNotFoundException** and **NullPointerException**. In addition, inside the constructor, we check to ensure that the file name ends with **.scala** and throw our own exception if it doesn't. The factory (**apply**) catches and converts all these exceptions. Note lines 24 and 26 do not capture the exception in an identifier; they are only concerned about the exception type, whereas line 28 uses the identifier **e** so it can call the **getMessage** method.

Because we revisit this example in later atoms, we create a reusable test. The argument to **CodeListingTester** is any function or method that takes a **String** (the name of the file) and produces an **IndexedSeq[String]**. We use **IndexedSeq[String]** instead of specifying a **CodeListing** because it makes the test more flexible. Each test uses **makeList** to create the object to be tested:

```
1  // CodeListingTester.scala
2  package codelistingtester
3  import com.atomicscala.AtomicTest._
4
5  class CodeListingTester(
6    makeList:String => IndexedSeq[String]) {
7
8    makeList("CodeListingTester.scala")(4) is
9    "class CodeListingTester("
10
11   makeList("NotAFile.scala")(0) is
12   "File Not Found: NotAFile.scala"
13
14   makeList("NotAScalaFile.txt")(0) is
15   "NotAScalaFile.txt " +
16   "doesn't end with '.scala'"
17
18   makeList(null)(0) is
19   "Error: Null file name"
20
21 }
```

The result of creating a **CodeListing** object always looks like a **Vector**, which we index (remember indexing starts at 0). For example, line 8 selects element 4.

The **null** on line 18 is the keyword to indicate "nothing."

The resulting test code is minimal:

```
1  // CodeListingTest.scala
2  import codelistingtester._
3  import codelisting._
4  new CodeListingTester(CodeListing.apply)
```

The only things we must change from one test to the next is the **import** on line 3 and the **makeList** argument, as you'll see when we create different versions of **CodeListing** in subsequent atoms.

Exercises

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

1. Working from **CodeListingTester.scala**, write a script that uses **CodeListing.scala** to open a source-code file and print all the lines in the file.
2. Add line numbering to your solution for the previous exercise.
3. Use your new script on a file that does not exist. Do you need to make additional modifications?

Error Reporting with Either

If treating every error as an exception creates code that's too messy, what's the alternative? Historically, programming language designers and users tried returning values that were out-of-bounds for a particular method, or setting a *global flag* to indicate that an error had occurred. (A *global* is something that everything everywhere in the program can see, and often modify, and is the source of untold problems in the history of programming). There were a confusing number of approaches, none of them worked particularly well, and information in globals and return values quickly vanished if you weren't rigorous. The result was that no one used these approaches with any consistency. Worse, the client programmer would ignore error conditions and pretend that every return value was a good one – encouraging, in effect, the creation of buggy code.

Scala introduces *disjoint unions* to return results from methods. A disjoint union combines two completely different (thus "disjoint") types: one type to show success and carry the return value, and one to indicate failure and hold failure information. When you call a method, you get back one of these unions and unpack it to see what happened. One benefit of this approach is that you must always explicitly look at success and failure; there is no easy way to assume that a method call "just works."

The initial experiment uses a union called **Either**, which combines **Left** and **Right** types. **Either** was created apart from error handling and has nothing to do with it, so the experimenters arbitrarily decided that **Left** indicates an error (no doubt following the centuries-old prejudice against the left, or *sinister*, side) and **Right** carries successful return information.

Here's the basic divide-by-zero example using **Either**. No imports are required to use **Left** and **Right**, and the return type (**Either[String, Int]**) is inferred by Scala:

```
1  // DivZeroEither.scala
2  import com.atomicscala.AtomicTest._
3
4  def f(i:Int) =
5    if(i == 0)
6      Left("Divide by zero")
7    else
8      Right(24/i)
9
10 def test(n:Int) =
11   f(n) match {
12     case Left(why) => s"Failed: $why"
13     case Right(result) => result
14   }
15
16 test(4) is 6
17 test(5) is 4
18 test(6) is 4
19 test(0) is "Failed: Divide by zero"
20 test(24) is 1
21 test(25) is 0
```

Left is a way to carry information which could be an exception type or anything else – the documentation for your method interface must explain to the client programmer what to do with the method results, and the client programmer must write appropriate code when calling your method.

The resulting syntax is elegant: on line 11 you see the call, followed by a **match** expression that handles both the failure and success cases.

So you say, “This is what I’m trying to do, and this is how I handle the result.”

Here’s the new version of **toss** using **Either**:

```
// MultiEitherErrors.scala
import com.atomicscala.AtomicTest._
import errors._

def tossEither(which:Int) = which match {
  case 1 => Left(Except1("Reason"))
  case 2 => Left(Except2(11))
  case 3 => Left(Except3("Wanted:", 1.618))
  case _ => Right("OK")
}

def test(n:Int) = tossEither(n) match {
  case Left(err) => err match {
    case Except1(why) => s"Except1 $why"
    case Except2(n) => s"Except2 $n"
    case Except3(msg, d) =>
      s"Except3 $msg $d"
  }
  case Right(x) => x
}

test(0) is "OK"
test(1) is "Except1 Reason"
test(2) is "Except2 11"
test(3) is "Except3 Wanted: 1.618"
```

We happen to be putting exceptions inside the **Left** objects, but you can put any information you want inside **Left** as your error report. **Either** does not provide any guidelines about what **Left** and **Right** mean, so the client programmer must figure it out for each different method.

Here's the factory in **CodeListing.scala** (from the previous atom) using **Either**:

```
// CodeListingEither.scala
package codelistingeither
import codelisting._
import java.io.FileNotFoundException

object CodeListingEither {
  def apply(name:String) =
    try {
      Right(new CodeListing(name))
    } catch {
      case _:FileNotFoundException =>
        Left(s"File Not Found: $name")
      case _:NullPointerException =>
        Left("Error: Null file name")
      case e:ExtensionException =>
        Left(e.getMessage)
    }
}
```

Note that **apply** is a translation of the exceptions into **Left** results, or the successful completion into a **Right** result. Using this class becomes a matter of unpacking the **Either**:

```
// ShowListingEither.scala
import codelistingtester._
import codelistingeither._

def listing(name:String) = {
  CodeListingEither(name) match {
    case Right(lines) => lines
    case Left(error) => Vector(error)
  }
```

```
10 }
11
12 new CodeListingTester(listing)
```

Finally, let's look at an interesting trick for collections of **Either**: you can **map** directly to a **match** clause, without saying **match** (you saw another example of this at the end of Combining Sequences with zip):

```
1  // EitherMap.scala
2  import com.atomicscala.AtomicTest._
3
4  val evens = Range(0,10) map {
5    case x if x % 2 == 0 => Right(x)
6    case x => Left(x)
7  }
8
9  evens is Vector(Right(0), Left(1),
10   Right(2), Left(3), Right(4), Left(5),
11   Right(6), Left(7), Right(8), Left(9))
12
13 evens map {
14   case Right(x) => s"Even: $x"
15   case Left(x) => s"Odd: $x"
16 } is "Vector(Even: 0, Odd: 1, Even: 2, " +
17   "Odd: 3, Even: 4, Odd: 5, Even: 6, " +
18   "Odd: 7, Even: 8, Odd: 9)"
```

Lines 4 and 13 show the tricky part – you see **map** where **match** would usually be. It's shorthand: the **map** applies the **match** expression to each element.

Lines 4-7 start with a **Range**, and the **match** expression produces a **Right** for even values and a **Left** for odd values. You see the resulting **Vector** on lines 9-11. Then lines 13-16 again use the shorthand syntax to divide the **Left**s and **Right**s via **case** statements.

Using **Right** and **Left** to indicate "success" and "failure" instantly becomes awkward. Why apply a non-specific tool to such an important task? Why not create a disjoint union dedicated to error reporting, and call the types **Success** and **Failure**? It was an experiment, one that succeeded, and now we have artifacts of that experiment in the Scala libraries – and possibly always will, for older code. In newer versions of Scala, efforts to solve the error problem are redoubled, so we hope for improvements.

Exercises

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

1. Add explicit return type information to **DivZeroEither.scala**.
2. Modify **TicTacToe.scala** from Summary 2 to use **Either**.
3. Using the techniques shown in **EitherMap.scala**, start with the range **'a' to 'z'** and divide it into vowels and consonants. Print the divided results. Satisfy the following test:
```
letters is "Vector(Left(a), Right(b)," +
"Right(c), Right(d), Left(e), Right(f)," +
"Right(g), Right(h), Left(i), Right(j)," +
"Right(k), Right(l), Right(m), Right(n)," +
"Left(o), Right(p), Right(q), Right(r)," +
"Right(s), Right(t), Left(u), Right(v)," +
"Right(w), Right(x), Right(y), Right(z))"
```
4. Adding to your solution for the previous exercise, write a **testLetters** method that separates the mapping into **Left** and **Right**, as you saw in **EitherMap.scala**. Satisfy the following tests:
```
testLetters(0) is "Vowel: a"
testLetters(4) is "Vowel: e"
testLetters(13) is "Consonant: n"
```

Handling Non-Values with Option

Consider a method that can sometimes produce results that "have no meaning." When this happens, the method doesn't produce an error per se; nothing went wrong, there's just "no answer." What you'd like to do is write code as if every value returned from that method is legitimate, and if it happens to be a "nothing" value, it gets quietly ignored. This is the intent of the **Option** type.

The following method is only interested in values between 0 and 1; it accepts the other values but nothing useful comes from them. We can use **Left** and **Right** to report the result:

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

def banded(input:Double) =
  if(input > 1.0 || input < 0.0)
    Left("Nothing")
  else
    Right(math.round(input * 100.0))

banded(0.555) is Right(56)
banded(-0.1) is Left("Nothing")
banded(1.1) is Left("Nothing")
```

"Not returning a value of interest" merits a special system instead of using **Either**. **Option** is another disjoint union created for this purpose, like a special case of **Either** with a predefined value for **Left** called **None**, while **Right** is replaced with **Some**:

```
1  // BandedOption.scala
2  import com.atomicscala.AtomicTest._
3
4  def banded2(input:Double) =
5    if(input > 1.0 || input < 0.0)
6      None
7    else
8      Some(math.round(input * 100.0))
9
10 banded2(0.555) is Some(56)
11 banded2(-0.1) is None
12 banded2(1.1) is None
13
14 for(x <- banded2(0.1))
15   x is 10
16
17 val result = for {
18   d <- Vector(-0.1, 0.1, 0.3, 0.9, 1.2)
19   n <- banded2(d)
20 } yield n
21 result is Vector(10, 30, 90)
```

Note that **banded2**'s return type **Option[Long]** (the return value of **math.round** is **Long**) is omitted; if you return **Some** and **None** then Scala infers the **Option**.

The **for** on line 14 looks like it's iterating over values in a container. However, you know that **banded2** is only returning a single value – it happens to be "contained in" an **Option**. The left-arrow "iteration-over" operation unpacks the value out of the **Option**; **x** is what is contained *inside* the **Option**. Here's an example showing the interaction between comprehensions and **Option**s in detail:

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

```
def cutoff(in:Int, thresh:Int, add:Int) =
  if(in < thresh)
    None
  else
    Some(in + add)

def a(in:Int) = cutoff(in, 7, 1)
def b(in:Int) = cutoff(in, 8, 2)
def c(in:Int) = cutoff(in, 9, 3)

def f(in:Int) =
  for {
    u <- Some(in)
    v <- a(u)
    w <- b(v)
    x <- c(w)
    y = x + 10
  } yield y * 2 + 1

f(7) is Some(47)
f(6) is None

val result =
  for {
    i <- 1 to 10
    j <- f(i)
  } yield j

result is Vector(47, 49, 51, 53)
```

The **cutoff** method has a threshold **thresh**; if **in** is below the threshold the result is **None**, otherwise the calculation is returned inside a **Some**. Methods **a**, **b** and **c** are created from **cutoff** and used inside **f** to show how a comprehension automatically extracts the contents of an **Option** *and* skips operations on **None**.

The first line of a comprehension determines the type of the rest of the lines as well as the **yield** type. In **f**, we want to work with **Option**s but the **input** argument is a plain **Int**, so we wrap it in a **Some** to establish the type of the comprehension. Each of lines 16-19 automatically extracts the contents of the **Option** and wraps the results in an **Option**; the result of the **yield** is also an **Option**. If any intermediate result in the comprehension is a **None**, the final result is also **None** – but you don't have to check each intermediate result. That's actually the point of **Option**: you don't have to test for **None** every time you perform an operation, and the resulting code is clean, easier to read than wading through all those tests, and still safe.

Note that line 21 **yield**s an expression, not just a single value.

The calculation of **result** on lines 26-30 is especially impressive – any calculation that produces **None** is automatically dropped from **result**, as you see on line 32.

Option's container-like behavior is not limited to comprehensions; it also provides the basic set of operations you find in most containers, including **foreach** and **map** – both of which do the right thing when working with **None**. Thus, when you perform operations on an **Option**, you *don't* have to check to see whether it's **Some** or **None**, extract the value, then put the result in an **Option** – **foreach** and **map** do it all for you!

```
1  // OptionOperations.scala
2  import com.atomicscala.AtomicTest._
3
4  def p(s:Option[String])= s.foreach(println)
5
6  p(Some("Hi")) // Prints "Hi"
7  p(Option("Hi")) // Prints "Hi"
8  p(None) // Doesn't do anything!
9
10 def f(s:Option[String]) = s.map(_ * 2)
```

```
11
12 f(Some("Hi")) is Some("HiHi")
13 f(None) is None
14
15 Option(null) is None
```

Line 4 uses **foreach** for a side-effect operation (**foreach** doesn't produce a result). On line 6 we pass a **Some** and this generates output. You can also hand a value to **Option** as on line 7, and this also produces a **Some**. If you hand it a **None**, though, it doesn't do anything, not even print a blank line. The argument of **foreach** is not executed at all.

On lines 10-13 we see similar behavior for **map**, except a result is generated from the **map** operation and wrapped in an **Option**. Note that **map** automatically extracts the contents of a **Some** before applying its argument.

Line 15 uses the **null** keyword, a wart we inherit from Java. In Java, **null** is like our **None** *except* you must constantly check for **null** and it tends to hide, only to blow up when you least expect it. Conveniently, when **Option** sees a **null** it converts it to a **None**, so when you use a Java library that can produce a **null**, just wrap any calls in **Option**.

While **Left** and **Right** have no particular meaning, **Some** and **None** are biased, and operations can take advantage of that bias. Here's a further example showing **foreach** and **map** ignoring **None** values. We chain operations together and everything still works without fail:

```
1  // OptionChaining.scala
2  import com.atomicscala.AtomicTest._
3
4  def f(n:Int, div:Int) =
5    if(n < div || div == 0)
6      None
7    else
```

```
8       Some(n/div)
9
10  f(0,0) is None
11  f(0,0).foreach(println) // Nothing printed
12  f(11,5) is Some(2)
13  f(11,5).foreach(println) // 2
14
15  def g(n:Int, div:Int) = f(n,div).map(_ + 2)
16
17  g(5,11) is None
18  g(11,5) is Some(4)
```

The **f** method returns an **Option**, and this is chained to **foreach** on line 11. A **None** is ignored; it does nothing at all. Line 13 produces a **Some** from **f**, so the **foreach** extracts and prints the contents. Line 15 shows **map** applied to the **Option** produced by **f**; **None**s pass through without attempting the **map** calculation.

Just like comprehensions, **map** and **foreach** don't care about quantity; they keep going until they run out of elements in a sequence. If the number of elements happens to be a single value held inside an **Option**, that works too.

Let's revisit **EitherMap.scala** from the previous atom. **Option** also has a **filter** method which creates **evens**:

```
1   // OptionMap.scala
2   import com.atomicscala.AtomicTest._
3
4   val evens = Range(0,10).
5     map(Option(_).filter(_ % 2 == 0))
6
7   evens is Vector(Some(0), None, Some(2),
8     None, Some(4), None, Some(6),
9     None, Some(8), None)
10
11  evens map {
```

```
12    case Some(x) => s"Even: $x"
13    case None => "Odd"
14  } is "Vector(Even: 0, Odd, Even: 2, " +
15    "Odd, Even: 4, Odd, Even: 6, " +
16    "Odd, Even: 8, Odd)"
```

On line 5 we take each element in the **Range** and turn it into an **Option**. **Option**'s **filter** method requires a function or method (we use the shorthand techniques shown in Brevity) that returns a Boolean result; if the result is **true**, the value is stored as a **Some**, otherwise it's a **None**. You see the resulting **Vector** on lines 7-9. The **map-match** shorthand trick works like it did in **EitherMap.scala**.

A particularly helpful application of **Option** is to add new arguments to an existing argument list without breaking code written for the old argument list. Suppose you create an **Art** class that includes **title** and **artist**. You publish this class and client programmers begin using it. Then you discover you want to include the **style** of the art. In many languages, your best approach is to use overloading and duplicate some of your code – but as we've pointed out, code duplication is a path to an un-maintainable code base. With **Option**s, you add the argument to the existing class or method:

```
1  // AddNewArguments.scala
2
3  case class Art(title:String, artist:String,
4    style:Option[String] = None)
5
6  val oldCall = Art("Guernica", "Picasso")
7  val newCall = Art("Soup Cans", "Warhol",
8    Option("Pop"))
```

All the new code you add to **Art** that manipulates **style** must treat it as an **Option**, which prevents you from accidentally assuming that it's always valid. This makes it much easier to extend the code safely, and

helps eliminate code duplication. Tools that allow you to evolve your code promote an incremental approach to system development; as you learn new things about your system, you more easily modify it to incorporate your new understanding.

Option is probably a poor term – at first you're probably wondering what your "options" are. You don't really have any; there's either something there ... or not. Names are important.

Exercises

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

1. Rewrite **DivZeroEither.scala** from Error Reporting with Either to use **Option** instead of **Either**. Satisfy the following tests:
```
f(4) is Some(6)
f(5) is Some(4)
f(6) is Some(4)
f(0) is None
f(24) is Some(1)
f(25) is Some(0)
```
2. Add explicit return types to the previous exercise.
3. Modify **TicTacToe.scala** from Summary 2 to use **Option**.
4. Create a method that ensures that its argument is numeric or alphabetical. Return **None** for any other characters. Satisfy the following tests:
```
alphanumeric(0) is Some(0)
alphanumeric('a') is Some('a')
alphanumeric('m') is Some('m')
alphanumeric('$') is None
alphanumeric('Z') is Some('Z')
```

Converting Exceptions with Try

To further reduce the amount of exception-management code, **Try** captures all exceptions and produces an object. If you say:

```
Try(expression that can throw exceptions)
```

You get back a **Success** object containing the result if no exceptions are thrown, and a **Failure** object containing error information if there's an exception. Thus, **Try** converts exceptions into objects, so you don't need a **catch** clause, or to worry that an exception will accidentally escape from the current context.

Success and **Failure** are subclasses of **Try**. **Try** appeared in Scala 2.10 so it won't work with earlier versions.

When we use **Try** in the divide-by-zero example, the method on line 5 becomes simple:

```
1  // DivZeroTry.scala
2  import com.atomicscala.AtomicTest._
3  import util.{Try, Success, Failure}
4
5  def f(i:Int) = Try(24/i)
6
7  f(24) is Success(1)
8  f(0) is "Failure(" +
9  "java.lang.ArithmeticException: " +
10 "/ by zero)"
11
```

```
12 def test(n:Int) =
13   f(n) match {
14     case Success(r) => r
15     case Failure(e) =>
16       s"Failed: ${e.getMessage}"
17   }
18
19 test(4) is 6
20 test(0) is "Failed: / by zero"
```

On line 7 you see that a successful call no longer returns a raw **Int**, but instead a **Success** object that wraps the result (this is identical to the way that **Option** returns **Some** and **Either** conventionally returns **Right**). The exception on line 8 is caught and wrapped in a **Failure** object.

The most basic way to "unwrap" the result is with pattern matching, as in the **test** method. Because **Success** and **Failure** are **case** classes, the **match** expression dissects them for us as on lines 14-16, where **r** and **e** are automatically extracted.

Failure and **Success** are more descriptive and memorable error-reporting objects than **Either**'s **Left** and **Right**.

An interesting aside: dividing by zero with **Double**s (instead of **Int**s) does *not* produce an exception. Instead, it produces a special "Infinity" object:

```
scala> 1.0/0.0
res0: Double = Infinity
```

Multiple types of exceptions can be further partitioned with a nested **match**:

```
// Try.scala
import com.atomicscala.AtomicTest._
import util.{Try, Success, Failure}
import errors._

def f(n:Int) = Try(toss(n)) match {
  case Success(r) => r
  case Failure(e) => e match {
    case Except1(why) => s"Except1 $why"
    case Except2(n) => s"Except2 $n"
    case Except3(msg, d) =>
      s"Except3 $msg $d"
  }
}

f(0) is "OK"
f(1) is "Except1 Reason"
f(2) is "Except2 11"
f(3) is "Except3 Wanted: 1.618"
```

At this point, things are getting slightly messy. Fortunately, **Try** has additional devices to simplify your code. The **recover** method takes any exception and converts it to a valid result:

```
// TryRecover.scala
import com.atomicscala.AtomicTest._
import util.Try
import errors._

def f(n:Int) = Try(toss(n)).recover {
  case e:Throwable => e.getMessage
}.get

```

```
def g(n:Int) = Try(toss(n)).recover {
  case Except1(why) => why
  case Except2(n) => n
  case Except3(msg, d) => s"$msg $d"
}.get

f(0) is "OK"
f(1) is "Reason"
f(2) is "11"
f(3) is "Wanted: 1.618"

g(0) is "OK"
g(1) is "Reason"
g(2) is "11"
g(3) is "Wanted: 1.618"
```

Success objects pass untouched through **recover**, but **Failure**s are captured and **match**ed with your **recover** clause. Whatever you produce from each **case** is returned, wrapped in a **Success**. Thus, when you use **recover** you only produce **Success** objects, and they must all make sense.

The **recover** in **f** catches any **Throwable** and gets the contained message. The **recover** in **g** matches on the specific exceptions of interest.

When you call **get** on a **Success** object, you get the contents back. However, if you call **get** on a **Failure** it generates an exception – the original exception placed inside the **Failure** object. For example, trying to convert the string "pig" to an **Int** will produce an exception that **Try** will capture into a **Failure** object:

```
// PigInt.scala
import util.Try

val result = Try("pig".toInt)
```

```
5
6   assert(
7     result.toString.startsWith("Failure"))
8
9   assert((try {
10    result.get
11  } catch {
12    case _:Throwable => "Yep, an exception"
13  }) == "Yep, an exception")
```

The **assert** on lines 6-7 shows that a **Failure** object is indeed produced (**startsWith**, as the name implies, is a **String** method that compares its argument with the beginning of its **String** object). On line 10 we call **get** on a **Failure**. The resulting exception is caught in the **catch** clause; the **assert** verifies that it happens. It makes sense to throw an exception for a bad **get** – it's generally a programming error (in **TryRecover.scala**, it means your **recover** clause didn't include all possible cases).

Notice that, like **Option**, **Try** feels like a container that holds a single item. Here, **Try** provides container operations:

```
1   // ContainerOfOne.scala
2   import com.atomicscala.AtomicTest._
3   import util.{Try, Success}
4
5   Try("1".toInt).map(_ + 1) is Success(2)
6   Try("1".toInt).map(_ + 1).foreach(println)
7   // Doesn't print anything:
8   Try("x".toInt).map(_ + 1).foreach(println)
```

We apply **map** to the result of **Try** on line 5, and the result is another **Try** object. On line 6 we chain an additional operation on that **Try**. It's the same as applying those operations to a **Vector** containing only one element. However, when the **Try** fails on line 8, the chained operations are automatically ignored and nothing further happens.

If you want to perform operations for *both* **Success** and **Failure**, **transform** takes one function for **Success** and another for **Failure**:

```
1  // TryTransform.scala
2  import com.atomicscala.AtomicTest._
3  import util.Try
4  import errors._
5
6  def f(n:Int) = Try(toss(n)).transform(
7    i => Try(s"$i Bob"), // Success
8    e => e match { // Failure
9      case Except1(why) => Try(why)
10     case Except2(n) => Try(n)
11     case Except3(msg, d) => Try(d)
12   }
13 ).get
14
15 f(0) is "OK Bob"
16 f(1) is "Reason"
17 f(2) is "11"
18 f(3) is "1.618"
```

The first argument to **transform** is used on a **Success** result, and the second argument is used on a **Failure** result. The return value in either case must be a **Try** object, although for this example none of the **Try**s within the **transform** can fail (note the call to **get** on line 13), so we could have made them all **Success** objects.

Depending on your needs, **Try** can produce some succinct error checking and handling code. For example, if you have a default fallback value, use **getOrElse** (also available for **Option**):

```
// IntPercent.scala
import com.atomicscala.AtomicTest._
import util.Try

def intPercent(amount:Int, total:Int) =
  Try(amount * 100 / total).getOrElse(100)

intPercent(49, 100) is 49
intPercent(49, 1000) is 4
intPercent(49, 0) is 100
```

The **getOrElse** argument can also be an expression.

If you only want to capture a subset of exceptions, create a **Catch** object. The **catching** method is a factory that takes a list of the exception classes you want to catch. **classOf** produces a class reference:

```
// Catching.scala
import com.atomicscala.AtomicTest._
import util.control.Exception.catching
import errors._

val ct2 = catching(classOf[Except2])

val ct13 = catching(classOf[Except1],
  classOf[Except3])

ct2.toTry(toss(0)) is "OK"
ct13.toTry(toss(0)) is "OK"
ct13.toTry(toss(1)) is
"Failure(errors.Except1: Reason)"
ct13.toTry(toss(3)) is
"Failure(errors.Except3: Wanted: 1.618)"

```

```
18 (try {
19   ct13.toTry(toss(2))
20 } catch {
21   case e:Throwable => "Except2"
22 }) is "Except2"
```

The **Catch** objects are configured as **ct2** to catch **Except2** and **ct13** to catch either **Except1** or **Except3**. Although **Catch** objects contain a litany of functionality (they predated **Try**), we only show **toTry** which produces **Try** objects; its argument is the expression you want to test. Notice on lines 14 and 16 that exceptions become **Failure** objects. Lines 18-22 show that if a **Catch** object doesn't handle a particular exception, that exception is passed through and must be caught in a conventional **catch** clause.

In general, **Catch** seems less useful than **Try**.

Try also works with comprehensions. Here is **ComprehensionOption.scala** from the previous atom, translated to use **Try** instead of **Option**:

```
1  // TryComprehension.scala
2  import com.atomicscala.AtomicTest._
3  import util.{Try, Failure, Success}
4
5  def cutoff(in:Int, thresh:Int, add:Int) =
6    if(in < thresh)
7      Failure(new Exception(
8        s"$in below threshhold $thresh"))
9    else
10     Success(in + add)
11
12 def a(in:Int) = cutoff(in, 7, 1)
13 def b(in:Int) = cutoff(in, 8, 2)
14 def c(in:Int) = cutoff(in, 9, 3)
15
```

```
16 def f(in:Int) =
17   for {
18     u <- Try(in)
19     v <- a(u)
20     w <- b(v)
21     x <- c(w)
22     y = x + 10
23   } yield y * 2 + 1
24
25 f(7) is Success(47)
26 f(6) is "Failure(java.lang.Exception: " +
27   "6 below threshhold 7)"
28
29 val result =
30   for {
31     i <- 1 to 10
32     j <- f(i).toOption
33   } yield j
34
35 result is Vector(47, 49, 51, 53)
```

Almost everything here is a straightforward translation of **ComprehensionOption.scala** *except* for line 32. In the original version of the example, we just said **j <- f(i)** and the result of **f(i)** was automatically unpacked into **j**. If you try the same thing here, you get an error message complaining that it wants a **GenTraversableOnce** instead of a **Try**. This happens because the first line of a comprehension (line 31) establishes the type of the sequence of the comprehension; here the **Range 1 to 10** is promoted to a **Vector**. Because the comprehension starts with something **Vector**-like for **i**, that's what it also wants to see for **j**, and that's what produces the complaint: Scala wants something "traversable" (like a **Vector**), and it doesn't know how to make that from a **Try**.

As we saw in **ComprehensionOption.scala**, Scala *does* know how to "traverse" (in this case, unpack) an **Option**, so by using **toOption** we

get the desired result. We don't actually want an **Option**, but the conversion tells Scala how to get around the problem.

Here's the **CodeListingEither.scala** factory method from Error Reporting with Either, rewritten using **Try** and **recover**:

```
// ShowListingTry.scala
import util.Try
import java.io.FileNotFoundException
import codelisting._
import codelistingtester._

def listing(name:String) =
  Try(new CodeListing(name)).recover{
    case _:FileNotFoundException =>
      Vector(s"File Not Found: $name")
    case _:NullPointerException =>
      Vector("Error: Null file name")
    case e:ExtensionException =>
      Vector(e.getMessage)
  }.get

new CodeListingTester(listing)
```

Note the improvement: previously we had to wrap a successful call in a **Right**, but **Try** wraps a good result in a **Success** for us. Unsuccessful calls were wrapped in **Left**s and later unpacked, but **recover** allows us to produce a usable result.

In Error Reporting with Either, we asked, "What about a disjoint union specific to errors?" **Try** appears be such a union, including the **Success** and **Failure** names. Why not replace **Left** and **Right** with **Failure** and **Success**? The difficulty with this approach is addressed in the next atom, which presents an alternative error-handling approach.

A later version of Scala may include a redesign for error reporting and handling. We have high hopes that the result will produce much simpler and more intuitive error-handling code.

We've focused on handling errors. There's also a third-party library that helps you *validate* data (a somewhat subtle distinction). This is the **Validation** component of the **scalaz** library, which you can explore on your own.

Exercises

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

1. Modify **TryTransform.scala** to show that all the **Try** calls within the **transform** argument list can be replaced with **Success**. Satisfy the following tests:
   ```
   f(0) is "OK Bob"
   f(1) is "Reason"
   f(2) is "11"
   f(3) is "1.618"
   ```
2. Remove the **.get** acting on the result of the transform. What must you do to make the tests pass?
3. Modify **ShowListingTry.scala** to include line numbers. Were you able to use the **CodeListingTester** from your solution in Constructors and Exceptions?

Custom Error Reporting

What should you use to report errors? As we've pointed out, **Either** was a useful experiment but **Left** and **Right** are not particularly meaningful names for error reporting (and although it's possible to use **Either** in comprehensions, extra syntax is required that makes it complex and less readable). **Try**'s **Success** and **Failure** are better names and more useful, but can we use them for ordinary error reporting?

To begin answering this question, let's create our own disjoint union to report errors, producing approximately the same effect as **Either** but with meaningful names. **Good** contains valid result data, and **Bad** contains an error message:

```
1  // CustomErrors.scala
2  import com.atomicscala.AtomicTest._
3
4  sealed trait Result
5  case class Good(x:Int, y:String)
6    extends Result
7  case class Bad(errMsg:String)
8    extends Result
9
10 def tossCustom(which:Int) = which match {
11   case 1 => Bad("No good: 1")
12   case 2 => Bad("No good: 2")
13   case 3 => Bad("No good: 3")
14   case _ => Good(which, "OK")
15 }
16
17 def test(n:Int) = tossCustom(n) match {
18   case Bad(errMsg) => errMsg
19   case Good(x, y) => (x, y)
20 }
```

```
21
22 test(47) is (47, "OK")
23 test(1) is "No good: 1"
24 test(2) is "No good: 2"
25 test(3) is "No good: 3"
```

The **sealed** keyword on line 4 was introduced in Tagging Traits and Case Objects.

The argument(s) to **Bad** could also be exceptions or any other useful types. Note that defining **Good** and **Bad** is about the same effort as defining your own exceptions.

This is an acceptable solution, as far as it goes. Return information is different for different methods. The caller must understand and deal with the return value, and the disjoint union forces the client programmer to acknowledge that they can get a **Bad** result. However, this approach still misses the syntactic power provided by **Option** and **Try**; for example, the ability to write comprehensions like we see in **ComprehensionOption.scala** and **TryComprehension.scala**.

What keeps us from adapting **Try** to report errors? The primary objection is that **Failure** requires a **Throwable** argument, and **Throwable** is the base class for exceptions. It thus comes burdened with a *stack trace*, which is all the information about the exception, where it comes from and all the intermediate steps it takes. Creating a stack trace in order to produce a simple error report is arguably too much overhead and will keep some people from using that technique.

Fortunately, there's a solution: **util.control** contains a trait called **NoStackTrace** that suppresses the creation of the stack trace, thus removing objections to using **Success** and **Failure** as return values. Here's a simple library that captures **String** error messages inside **Failure** objects:

```
1  // Fail.scala
2  package com.atomicscala.reporterr
3  import util.Failure
4  import util.control.NoStackTrace
5
6  class FailMsg(val msg:String) extends
7  Throwable with NoStackTrace {
8    override def toString = msg
9  }
10
11 object Fail {
12   def apply(msg:String) =
13     Failure(new FailMsg(msg))
14 }
```

Because **FailMsg** incorporates **NoStackTrace**, it works as an exception object but doesn't create a stack trace:

```
1  // FailMsgDemo.scala
2  import com.atomicscala.reporterr.FailMsg
3
4  try {
5    throw new FailMsg("Caught in try block")
6  } catch {
7    case e:FailMsg => println(e.msg)
8  }
9
10 throw new FailMsg("Uncaught")
11 println("Beyond uncaught")
12
13 /* Output:
14 Caught in try block
15 Uncaught
16 */
```

Lines 4-8 show **FailMsg** behaving as an exception. On line 10 you see that when you throw a **FailMsg** and don't catch it, it goes "all the way

out," just like any exception. With other exceptions this produces a long and noisy stack trace, but all you see is "Uncaught." Also, note that the **println** on line 11 never happens because throwing an exception halts the ordinary forward progress of the program.

Now we can use **Success** and **Failure** objects as method return values. The **apply** method in **object Fail** produces a simple syntax when reporting an error (lines 8 and 10):

```
1  // UsingFail.scala
2  import com.atomicscala.AtomicTest._
3  import util.{Try, Success}
4  import com.atomicscala.reporterr.Fail
5
6  def f(i:Int) =
7    if(i < 0)
8      Fail(s"Negative value: $i")
9    else if(i > 10)
10     Fail(s"Value too large: $i")
11   else
12     Success(i)
13
14 f(-1) is "Failure(Negative value: -1)"
15 f(7) is "Success(7)"
16 f(11) is "Failure(Value too large: 11)"
17
18 def calc(a:Int, b:String, c:Int) =
19   for {
20     x <- f(a)
21     y <- Try(b.toInt)
22     sum  = x + y
23     z <- f(c)
24   } yield sum * z
25
26 calc(10, "11", 7) is "Success(147)"
```

```
27 calc(15, "11", 7) is
28 "Failure(Value too large: 15)"
29 calc(10, "dog", 7) is
30 "Failure(java.lang." +
31 "NumberFormatException: " +
32 """For input string: "dog")"""
33 calc(10, "11", -1) is
34 "Failure(Negative value: -1)"
```

The **calc** method shows that the **Success** and **Failure** objects produced by both **Try** and our **f** method are usable in a comprehension. Notice line 21, which calls **toInt** to convert a **String** to an **Int**; if this fails it throws an exception to be captured by **Try** and reported in the same way that our custom errors are reported.

Here's the divide-by-zero example using **reporterr**:

```
1  // DivZeroCustom.scala
2  import com.atomicscala.AtomicTest._
3  import util.Success
4  import com.atomicscala.reporterr.Fail
5
6  def f(i:Int) =
7    if(i == 0)
8      Fail("Divide by zero")
9    else
10     Success(24/i)
11
12 def test(n:Int) = f(n).recover{
13     case e => s"Failed: $e"
14   }.get
15
16 test(4) is 6
17 test(5) is 4
18 test(6) is 4
19 test(0) is "Failed: Divide by zero"
```

```
test(24) is 1
test(25) is 0
```

Lines 12-14 show the benefit of adapting **Try** instead of creating our own error-reporting system from scratch (the **Good**/**Bad** example). Here you only see **recover** and **get**, but all the **Try** operations (including its use with comprehensions, as in **UsingFail.scala**) are automatically available when you use **reporterr**. **Try** and our **reporterr** package work seamlessly together.

Here's **CodeListing.scala** from Constructors and Exceptions converted to use **reporterr**:

```
// CodeListingCustom.scala
package codelistingcustom
import codelisting._
import java.io.FileNotFoundException
import util.Success
import com.atomicscala.reporterr.Fail

object CodeListingCustom {
  def apply(name:String) =
    try {
      Success(new CodeListing(name))
    } catch {
      case _:FileNotFoundException =>
        Fail(s"File Not Found: $name")
      case _:NullPointerException =>
        Fail("Error: Null file name")
      case e:ExtensionException =>
        Fail(e.getMessage)
    }
}
```

Remember, we can't use **Success** and **Fail** from inside the constructor, because you can't return anything from a constructor, so if the

constructor fails it must throw an exception. The **apply** factory method catches these exceptions and converts them to **Failure** objects.

To use it we again convert all errors to **Vector[String]**:

```
// ShowListingCustom.scala
import codelistingcustom._
import codelistingtester._

def listing(name:String) =
  CodeListingCustom(name).recover{
    case e => Vector(e.toString)
  }.get

new CodeListingTester(listing)
```

Let's take one more look at this example. We use composition, so the constructor controls the creation of the composed **Vector**. Because we create **Vector**s to hold error messages, we capture all errors within the constructor and place them in the composed **Vector**, to produce more succinct code:

```
// CodeVector.scala
package codevector
import util.Try
import java.io.FileNotFoundException

```

```
class CodeVector(val name:String)
extends collection.IndexedSeq[String] {
  val vec = name match {
    case null =>
      Vector("Error: Null file name")
    case name
      if(!name.endsWith(".scala")) =>
      Vector(
        s"$name doesn't end with '.scala'")
    case _ =>
      Try(io.Source.fromFile(name)
          .getLines.toVector).recover{
        case _:FileNotFoundException =>
          Vector(s"File Not Found: $name")
      }.get
  }
  def apply(idx:Int) = vec(idx)
  def length = vec.length
}
```

Now the constructor doesn't throw exceptions, and this eliminates all error-handling code outside the constructor. Note that if you use inheritance, there is no way to catch the exceptions thrown by the base-class constructor.

Since **CodeVector** has no **apply** method, we create an anonymous function (using shorthand notation) as the argument for **CodeListingTester**:

```
// ShowCode.scala
import codelistingtester._
import codevector._
new CodeListingTester(new CodeVector(_))
```

The "no exceptions from the constructor" approach produces the cleanest code we've seen so far. Of course, at some point you must

still deal with the problem; here the information is presumably transmitted to the end user.

Exercises

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

1. Rewrite **ShowListingEither.scala** (and other code as necessary) to use **Success** and **Fail**.

2. Modify **TicTacToe.scala** from Summary 2 to use **Success** and **Fail**.

3. Write a method **testArgs** that takes a variable argument list of tuples, where each tuple contains a **Boolean** expression and a **String** message for when the **Boolean** fails. For each tuple, produce a **Success** or **Failure**. Now create a method:
   ```
   f(s:String, i:Int, d:Double)
   ```
   Within the method, call **testArgs** passing it the following tuples:
   ```
   (s.length > 0, "s must be non-zero length"),
   (s.length <= 10, "length of s must be <= 10"),
   (i >= 0, "i must be positive"),
   (d > 0.1, "d must be > 0.1"),
   (d < 0.9, "d must be < 0.9")
   ```
   Take the output and filter it so only **Failure** objects remain. Satisfy the following tests:
   ```
   f("foo", 11, 0.5) is ""
   f("foobarbazbingo", 11, 0.5) is
   "Failure(length of s must be <= 10)"
   f("", 11, 0.5) is
   "Failure(s must be non-zero length)"
   f("foo", -11, 0.5) is
   "Failure(i must be positive)"
   f("foo", 11, 0.1) is
   "Failure(d must be > 0.1)"
   f("foo", 11, 0.9) is
   "Failure(d must be < 0.9)"
   ```

Design by Contract

Design by Contract (DbC) is attributed to Bertrand Meyer, the creator of the Eiffel programming language, from which Scala draws its DbC inspiration. DbC focuses on design errors by validating that arguments and return values conform – at run time – to expected rules (the "contract") that are determined during the design process. It also uses the concept of *invariants*: values that should be the same at the beginning and the end of a method call.

DbC is yet another way to discover or avoid errors, which follows the pattern we've seen: There are various approaches for revealing errors ... because it's not a trivial problem. A single tool doesn't seem to work for all situations, so we end up with multiple strategies.

Methods used the built-in **assert** to ensure that expressions were true. Scala contains similar methods **require** and **assume** (the latter is an alias for **assert**) for use as DbC tools. A failure of a **require** or **assume** represents a programming error so there's no real hope of continuing execution; you decide the best way of reporting the error and quitting the program (the default is an exception dump). That's one nice thing about **require** and **assume** – you insert them as program checks without putting in any other scaffolding, since they only fire when they fail (indicating a bug).

```
1  // DesignByContract.scala
2  import com.atomicscala.AtomicTest._
3  import util.Try
4
5  class Contractual {
6    def f(i:Int, d:Double) = {
7      require(i > 5 && i < 100,
8        "i must be within 5 and 100")
9      val result = d * i
```

```
    assume(result < 1000,
      "result must be less than 1000")
    result
  }
}

def test(i:Int, d:Double) =
  Try(new Contractual().f(i, d)).recover{
    case e => e.toString
  }.get

test(10, 99) is 990.0
test(11, 99) is
"java.lang.AssertionError: " +
"assumption failed: " +
"result must be less than 1000"
test(0, 0) is
"java.lang.IllegalArgumentException: " +
"requirement failed: " +
"i must be within 5 and 100"
```

A *precondition* typically looks at method arguments, to verify they are within the set of valid/acceptable values, before the main part of the method body. If a precondition fails, the method cannot execute. The **require** method says, "this must be true," so it takes a Boolean expression along with an optional **String** message, given as part of the error report. If **require** fails, it throws an **IllegalArgumentException** – another indicator that it tests method arguments.

Because you never know what kinds of arguments the client programmer will pass to your methods, once you put a precondition in place you usually never take it out – you can never guarantee that it won't be violated, because you can't predict what the client programmer will do.

A *postcondition* normally checks the results of a method call, and is tested with the **assume** method, which throws an **AssertionError** if it fails. While a precondition guarantees *argument* correctness, a postcondition helps verify *method* correctness, to ensure that your code doesn't do anything to violate the rules of your program. This means that, at some point after you've done sufficient testing, you've effectively proven that your postcondition will always be true (assuming the preconditions are true).

Once you've proven this, the postcondition becomes redundant and it would be nice to take it out, for efficiency's sake. But it would also be nice to leave something in place in case you change the code and want to re-enable testing. Conveniently, Scala provides a compilation flag to remove *elidable* expressions. This example demonstrates the effect:

```
1  // ElidingDBC.scala
2  import util.Try
3
4  object ElidingDBC extends App {
5    println(Try(require(false, "require!")))
6    println(Try(assume(false, "assume!")))
7    println(Try(assert(false, "assert!")))
8  }
```

We use **Try** to reduce the output of lines 5-7 to a single line each.

If you run the shell commands:

```
scalac -Xelide-below 2001 ElidingDBC.scala
scala ElidingDBC
```

Only the **require** in line 5 survives, and the **assume** and **assert** on lines 6 and 7 are removed by the compiler. When the value is 2000 or less, **assume** and **assert** survive compilation, while 2001 or greater removes

them. However, **require** is never removed because you can't guarantee that the client programmer will meet the argument preconditions.

There's an additional construct called **assuring** that supports the third part of DbC, invariants (not covered here). Learn more about DbC through Wikipedia and other web resources.

Exercises

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

1. Create three methods: the first checks only preconditions, the second checks only postconditions, and the third checks both. Each method has the same body: it takes a **String** argument which must be between 4-10 characters, and each of those characters must represent a digit. Each method converts each digit into an **Int** and then adds up all the digits to produce the result. The postcondition should verify the result is in the expected range of values.

2. Write an **App** (see Applications) with a method that takes the command-line argument of a **String** of letters, converts it to lowercase, and then converts each character to its numerical value in the alphabet, with 'a' being 1, 'b' being 2, etc. Sum the values and display the result. Use preconditions to verify that the input is in the correct form, and postconditions to ensure that the result is in the expected range of values.

3. Write a method that takes an **Int** argument, multiplies it by 3, and has a postcondition that fails if the result is odd. Elide the postcondition and show the failure slipping through. Add a precondition to prevent the failure.

Logging

In some cases, all you can do when you discover an issue is report it. In a web application, for example, there's no option to shut down the program if something goes wrong. *Logging* records such events, giving the programmer and/or administrator of the application yet another tool to discover problems.

For simplicity, we adapt Java's built-in logging, which is good enough for our purposes and doesn't require additional library installations (many have found Java's approach insufficient, so there are numerous third-party logging packages). We've written it as a trait so it can be combined with any class:

```
1  // Logging.scala
2  package com.atomicscala
3  import java.util.logging._
4
5  trait Logging {
6    val log = Logger.getLogger(".")
7    log.setUseParentHandlers(false)
8    log.addHandler(
9      new FileHandler("AtomicLog.txt"))
10   log.addHandler(new ConsoleHandler)
11   log.setLevel(Level.ALL)
12   log.getHandlers.foreach(
13     _.setLevel(Level.ALL))
14   def error(msg:String) = log.severe(msg)
15   def warn(msg:String) = log.warning(msg)
16   def info(msg:String) = log.info(msg)
17   def debug(msg:String) = log.fine(msg)
18   def trace(msg:String) = log.finer(msg)
19 }
```

The Java logging library has *loggers*, to which you write messages, and *handlers*, which record messages to their respective mediums.

If the argument to **getLogger** is an empty string, the log messages will include:

```
java.util.logging.LogManager$RootLogger
```

If the **getLogger** argument is a non-empty string, the string itself will be ignored but the log messages will instead include (along with the name of the **Logging** method that was called for the log entry):

```
com.atomicscala.Logging$class
```

Each logger can talk to many handlers; the handler on lines 8-9 writes to a file, and the handler on line 10 writes to the console. There's also a default console handler, so to prevent duplicate output we turn that off via line 7.

We want to set the "logging level" at **Level.ALL** to show all messages (other levels show fewer messages). However, we can't just set the level of the logger by itself (line 11); both the logger and its handlers independently pay attention to or ignore messages based on their levels. Thus, we must also set the level for all the handlers (lines 12-13).

To use the library, mix the **Logging** trait into your class:

```
1  // LoggingTest.scala
2  import com.atomicscala.Logging
3
```

```
4  class LoggingTest extends Logging {
5    info("Constructing a LoggingTest")
6    def f = {
7      trace("entering f")
8      // ...
9      trace("leaving f")
10   }
11   def g(i:Int) = {
12     debug(s"inside g with i: $i")
13     if(i < 0)
14       error("i less than 0")
15     if(i > 100)
16       warn(s"i getting high: $i")
17   }
18 }
19
20 val lt = new LoggingTest
21 lt.f
22 lt.g(0)
23 lt.g(-1)
24 lt.g(101)
```

All the **Logging** methods become native parts of **LoggingTest**, so you call them as you do other methods in the class. For example, on line 5 we call **info** without any qualification.

After running the program, look at the **AtomicLog.txt** file. It contains a lot more than what appears on the console. If you've ever looked at the source code for an HTML file, this looks familiar – everything seems to have lots of angle brackets and tags describing all the pieces. The log file is written in *XML* (eXtensible Markup Language) and it's intended to be easy to process and manipulate (The Scala distribution contains libraries for handling XML). Because log files tend to be long (especially for a web application), anything that helps you extract information is a benefit.

We've now looked at numerous ways to reveal problems in your programs, but a preponderance of studies indicate that the single most effective way to discover errors is through the process of *code review*: you get several people together and walk through the code. Despite these studies (and the fact that code reviews are a great way to transfer knowledge), code reviews are rarely practiced; they are deemed "too expensive." Hope and magical thinking are apparently considered better business strategies.

Exercises

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

1. Add an additional **FileHandler** and **ConsoleHandler** to **Logging.scala** and verify that the outputs are duplicated for both.
2. Continue the previous exercise by adding a **FileHandler** and **ConsoleHandler** for each logging level, and set the level of each handler appropriately. Verify that each handler only captures the output for its level.
3. Rewrite **Logging.scala** and **LoggingTest.scala** to produce an **App** (see Applications) that uses its command-line argument to set the logging level. Verify that it works with all logging levels.

Extension Methods

Suppose you discover a library that does everything you need … almost. If it only had one or two additional methods, it would solve your problem perfectly. But it's not your code – either you don't have access to the source code or you don't control it (so you'd have to repeat the modifications every time a new version came out).

Scala supports *extension methods*: you can, in effect, add your own methods to existing classes. Extension methods are implemented using *implicit classes*. If you put the **implicit** keyword in front of a class definition, Scala can automatically use the class argument to produce an object of your new type, and then apply your methods to that object. There's a restriction, though: extension methods must be defined within an **object**. Here are two extension methods for the **String** class:

```
1  // Quoting.scala
2  import com.atomicscala.AtomicTest._
3
4  object Quoting {
5    implicit class AnyName(s:String) {
6      def singleQuote = s"'$s'"
7      def doubleQuote = s""""$s""""
8    }
9  }
10 import Quoting._
11
12 "Hi".singleQuote is "'Hi'"
13 "Hi".doubleQuote is "\"Hi\""
```

Because the class is **implicit**, Scala takes any **String** called with either **singleQuote** or **doubleQuote** and converts it to an **AnyName**, thus legitimizing the call.

The triple quotes on line 7 allow double quotes within the **String**. The backslashes on line 13 are necessary to "escape" the inner quote marks, so Scala treats them as characters and not the end of the **String**.

The name of the **implicit class** (**AnyName**) is unimportant, as Scala only uses it to create an intermediate object on which to call the extension methods. In some situations, the creation of this intermediate object is objectionable for performance reasons. To remove this issue, Scala provides the *value type*, which doesn't create an object but just makes the call. To turn **AnyName** into a value type, you inherit from **AnyVal**, and the single class argument must be a **val** as you see on line 4:

```
1  // Quoting2.scala
2
3  package object Quoting2 {
4    implicit class AnyName(val s:String)
5    extends AnyVal {
6      def singleQuote = s"'$s'"
7      def doubleQuote = s""""$s""""
8    }
9  }
```

Here, we wrap the **implicit class** inside a **package object** to create the **object Quoting2** and also make it into a **package**.

Now we import and use the extension methods as before, and the results look the same:

```
// Quote.scala
import com.atomicscala.AtomicTest._
import Quoting2._

"Single".singleQuote is "'Single'"
"Double".doubleQuote is "\"Double\""
```

The difference is that, under the covers, no intermediate **AnyName** objects are created when making the calls. By using **AnyVal**, the calls are made without the extra overhead (Note that, in many cases, this overhead is minimal and unimportant. The Scala designers didn't ever want it to become an issue so they added value classes).

Extension methods can have arguments:

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

case class Book(title:String)

object BookExtension {
  implicit class Ops(book:Book) {
    def categorize(category:String) =
      s"$book, category: $category"
  }
}
import BookExtension._

Book("Dracula") categorize "Vampire" is
"Book(Dracula), category: Vampire"
```

Because we use a single argument in **categorize**, we write the call using "dot-free" infix notation on line 14 (try the conventional notation to verify that it also works).

Ultimately, extension methods are syntax sugar; the previous example can be rewritten as a method **categorize(Book, String)**. However, people seem to find that extension methods make the resulting code more readable (the best argument for syntax sugar).

Exercises

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

1. Rewrite **ExtensionMethodArguments.scala** so you get the same results *without* using extension methods.
2. Modify **ExtensionMethodArguments.scala** by adding an additional extension method in that has *two* arguments. Write tests.
3. Rewrite **ExtensionMethodArguments.scala** to turn **Ops** into a value class.

Extensible Systems with Type Classes

In this final atom, we open your mind to some of Scala's deeper possibilities. We introduce a few additional concepts and features, and you might find these a bit more challenging. If you don't get it right away, it's the last atom in the book so it's not a problem if you come back and figure it out later.

Extensibility is important because you don't often know the full breadth of your system when you first build it. As you discover additional requirements, you build new versions by adding functionality. We saw one way to create an extensible system in Polymorphism: inherit a new class and override methods. Here, we look at *type classes*, a different way to build an extensible system.

Let's review the polymorphic approach. Suppose you're managing shapes (for graphics, or geometry), and can calculate the area of a shape. To extend the system, you inherit from **Shape** and define **area** implementations:

```
1  // Shape_Inheritance.scala
2  import com.atomicscala.AtomicTest._
3  import scala.math.{Pi, sqrt}
4
5  trait Shape {
6    def area:Double
7  }
8
9  case class Circle(radius:Double)
10 extends Shape {
11   def area = 2 * Pi * radius
12 }
```

```
13
14 case class EQLTriangle(side:Double)
15 extends Shape {
16   def area = (sqrt(3)/4) * side * side
17 }
18
19 val shapes = Vector(Circle(2.2),
20   EQLTriangle(3.9), Circle(4.5))
21
22 def a(s:Shape) = f"$s area: ${s.area}%.2f"
23
24 val result = for(s <- shapes) yield a(s)
25
26 result is "Vector(Circle(2.2) area: " +
27 "13.82, EQLTriangle(3.9) area: 6.59," +
28 " Circle(4.5) area: 28.27)"
```

The **area** method contains the standard mathematical formula for each type; **EQLTriangle** stands for "equilateral triangle," so all the sides are the same length. Here, we don't explicitly use the **override** keyword because we're extending a **trait** containing an abstract method.

Each object in the **shapes** sequence (a **Vector[Shape]**) is manipulated as a generic **Shape**. The **area** method is resolved – at run time – to its specific object type so the proper **area** is calculated.

Line 22 uses String Interpolation with the **f** interpolator, which gives you fine-grained control of formatting. Like **s**, **f** provides expression evaluation within **${}**. At the end of that line, we use the format string **%.2f** to format a floating-point number (our **Double** result from **area**) with two places to the right of the decimal point.

The polymorphism approach is built into all object-oriented languages. However, the extensibility of this system is tightly bound to the inheritance hierarchy. This can be a problem if you want to

create functionality *across* types, regardless of the hierarchies those types might or might not belong to. A type class system allows you to add functionality to new types with a minimum of code, and without injecting yourself into a type hierarchy. You can even add functionality to new types when you don't have control of those types, for example if they come from a library written by someone else. It's similar to Extension Methods but it works across types rather than just extending a single type.

Type classes allow you to decouple functionality from type. A separate inheritance relationship is dedicated only to functionality, and can be applied to any type of object once you've "trained" the system to work with that type. Best of all, Scala automatically and invisibly chooses the proper functionality to apply to the object. Here is the previous example rewritten to use type classes; this includes new features that will be explained:

```
1  // Shape_TypeClass.scala
2  import com.atomicscala.AtomicTest._
3  import scala.math.{Pi, sqrt}
4
5  trait Calc[S] {
6    def area(shape:S):Double
7  }
8
9  def a[S](shape:S)(implicit calc:Calc[S]) =
10   f"$shape area: ${calc.area(shape)}%2.2f"
11
12 case class Circle(radius:Double)
13
14 implicit object CircleCalc
15 extends Calc[Circle] {
16   def area(shape:Circle) =
17     2 * shape.radius * Pi
18 }
19
```

```
20 case class EQLTriangle(side:Double)
21
22 implicit object EQLTriangleCalc
23 extends Calc[EQLTriangle] {
24   def area(shape:EQLTriangle) =
25     (sqrt(3)/4) * shape.side * shape.side
26 }
27
28 a(Circle(2.2)) is "Circle(2.2) area: 13.82"
29 a(EQLTriangle(3.9)) is
30 "EQLTriangle(3.9) area: 6.59"
31 a(Circle(4.5)) is "Circle(4.5) area: 28.27"
```

The **trait Calc** is the root of the "functionality hierarchy." Note that it has an unconstrained type parameter **S** – this means you can't call any methods on it because you don't know its capabilities. The only action you can perform is to pass it as an argument to a method; the method **area**, in this case. When each of the (multiple) implementations of **Calc** is created, **S** is specified and this allows a particular implementation of **area** to call methods on its specific type of **shape**.

The definition of **a** on lines 9-10 introduces two new features. The first is that there appears to be *two* argument lists. This is called *currying*, and for our purposes it means each argument list is evaluated independently. Second, the argument in the second list is **implicit**, which means Scala can automatically insert that argument during a call. However, for that to work, the rule is that the candidate **object**s for insertion – **CircleCalc** and **EQLTriangleCalc** – must *also* be **implicit**. Notice in the calls to **a** on lines 28, 29 and 31 that only the first argument is provided, because Scala is automatically finding and inserting the second argument – this syntax is the result of combining currying and **implicit** arguments.

Notice that **a** calls the **area** method of **calc**, passing it the **shape**. Inside **a**, both **calc** and **shape** are parameterized on **S**, so when **a** is called their types will be known, and the compiler will check those types. This is an important difference between this example and the previous one; in traditional polymorphism, the actual type (and the correct overridden method) is determined at runtime, but with type classes everything is resolved at compile time, before the program runs.

Lines 5-10 create the type class framework. Now we can add any classes to that framework, along with their associated **Calc** objects, and **a** will work with those new classes. Note that **Circle** and **EQLTriangle** have no connection with each other or any other classes, unlike in **Shape_Inheritance.scala** where they had to be bound together via the **Shape** base trait. To add a new class to this example, we either create it, or import it from another library, and then write the associated **Calc** object.

Both **CircleCalc** and **EQLTriangleCalc** specify their associated object type when they extend **Calc**. This allows them to access the elements of that type; here, **radius** or **side**.

When you call **a**, you hand it an object which goes in the first argument list. Then Scala looks around for an **implicit** subtype of **Calc** to (silently) place in the second argument list. If it's unable to find that object, you get a compile-time error to that effect.

Notice the clean syntax. You pass your object to **a**, then Scala invisibly looks up the associated **Calc** object and performs your operation. If you want to add another type to your system, you create another **Calc** object. In many languages this form of extensibility is a lot messier and more confusing.

We don't iterate through a **Vector** of objects as we did in **Shape_Inheritance.scala**. By design, our objects have nothing in

common, so if we put them in a common collection they get treated as a generic type, usually **Serializable**. When you try to pass each object in the collection to **a**, it gets a **Serializable** and doesn't know what to do with it. There is a solution for this problem, but we leave it as an exercise for the reader (... to search the Internet for blog posts on the subject).

Exercises

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

1. Add **class Rectangle** to **Shape_Inheritance.scala** and verify that it works. Now add **class Rectangle** and its associated **RectangleCalc** to **Shape_TypeClass.scala** and verify that it works. Note the differences.

2. Add a new operation **checkSum** to **Shape_Inheritance.scala** that turns the area into a **String**, then sums each digit (and the decimal point) to produce an **Int** result. Verify that it works. Now do the same thing to **Shape_TypeClass.scala** and note the differences.

3. Add a new class to **Shape_TypeClass.scala** but do *not* create an associated **Calc** class. Try to use it and see what happens.

4. Try duplicating lines 19-20 and 24 of **Shape_Inheritance.scala** in **Shape_TypeClass.scala** and see what happens. Why does this make sense?

5. Create a type class **trait** called **Reporter** with a method **generate**. Write a method **report** that takes any object and its associated **Reporter** and produces a **String** (using **generate**) containing information about that object. Create **case** classes **Person**, **Store** and **Vehicle**, each containing different types of information. Create their associated **Reporter** objects and show that your type class system works correctly.

6. Create a type class **trait** called **Transformer** with a method **convert**, but **Transformer** takes *two* type parameters: the type it's converting from, and the type it's converting to. Write a method **transform** that takes any object and its associated **Transformer** and converts the object. Create several classes and their associated **Transformer**s and show that your type class system works correctly.

7. Start with the first example class and transformation in the previous exercise. Try adding a second method **transform2** that produces a different type of result. Why doesn't this work? Add code to fix the problem.

⚛ Where to Go Now

Here is our suggested order of study:

- Check **AtomicScala.com** for more information, including supplements, solution guides, seminars and other books.
- **Scala Koans**: A self-guided exercise tour for beginning Scala programmers at **www.scalakoans.org**.
- Twitter's *Scala School* at **twitter.github.com/scala_school** also treats Scala as a new language (it doesn't require Java knowledge). Some material will be review, but they also cover other topics and some deeper issues than we do here.
- Twitter's *Effective Scala* at **twitter.github.com/effectivescala** provides helpful usage guidelines.
- **Scala for the Impatient** by Cay Horstmann at **horstmann.com/scala**.
- **Programming in Scala, 2nd Edition** by Martin Odersky, Lex Spoon, and Bill Venners at **www.artima.com/shop/programming_in_scala**. This is "the big book of Scala" that covers as much as it can, including some fairly advanced topics.

Appendix A: AtomicTest

Here is the test framework we use in the book, but note that it includes Scala features that are more advanced than are covered in this book.

```
// AtomicTest.scala
/* A tiny little testing framework, to
display results and to introduce & promote
unit testing early in the learning curve.
To use in a script or App, include:
import com.atomicscala.AtomicTest._
*/
package com.atomicscala
import language.implicitConversions
import java.io.FileWriter

class AtomicTest[T](val target:T) {
  val errorLog = "_AtomicTestErrors.txt"
  def tst[E](expected:E)(test: => Boolean){
    println(target)
    if(test == false) {
      val msg = "[Error] expected:\n" +
        expected
      println(msg)
      val el= new FileWriter(errorLog,true)
      el.write(target + msg + "\n")
      el.close()
    }
  }
  def str = // Safely convert to a String
    Option(target).getOrElse("").toString
  def is(expected:String) = tst(expected) {
    expected.replaceAll("\r\n","\n") == str
  }
```

```
30    def is[E](expected:E) = tst(expected) {
31      expected == target
32    }
33    def beginsWith(exp:String) = tst(exp) {
34      str.startsWith(
35        exp.replaceAll("\r\n","\n"))
36    }
37  }
38
39  object AtomicTest {
40    implicit def any2Atomic[T](target:T) =
41      new AtomicTest(target)
42  }
```

Appendix B: Calling Scala from Java

This appendix is for Java programmers. Once you see that Java libraries can be effortlessly called from Scala, and that Scala compiles to **.class** files, the question inevitably arises: "Can I call Scala from Java?"

Yes, and with a little care you can make Scala libraries look just like Java libraries when called from Java code.

First, you must add **scala-library.jar** to your CLASSPATH. This file is part of your standard Scala installation. Like any Jar file, you must add the entire path including the name of the Jar file itself.

Scala has additional features that aren't available in Java. Although it's possible to access these features by writing special code, it's easier and clearer if you write your Scala interface in such a way that it looks like plain Java inside your Java code. That way, your Java code doesn't look strange or intimidating to readers unfamiliar with Scala. If necessary, write an "adapter" class to simplify the interface for Java.

Here's an example showing how simple it can be. It's the well-known *Sieve of Eratosthenes* that finds prime numbers. The Scala library is clever and dense and we won't explain it (or the additional Scala features it uses) here – there are numerous explanations on the Web. Suffice it to say this code is much more compact than possible in Java, and despite the complexity, easier to verify:

```
// Eratosthenes.scala
package primesieve

object Eratosthenes {
  def ints(n:Int):Stream[Int] =
    Stream.cons(n, ints(n+1))
  def primes(nums:Stream[Int]):Stream[Int]=
    Stream.cons(nums.head, primes(
      nums.tail.filter(
        n => n % nums.head != 0)))
  def sieve(n:Int) =
    primes(ints(2)).take(n).toList
}
```

To use it in Java, we simply import the library and call **sieve**. If your CLASSPATH is set properly, you should get no warnings or errors when you compile this code:

```
// FindPrimes.java
import primesieve.*;

public class FindPrimes {
  public static void main(String[] args) {
    System.out.println(
      Eratosthenes.sieve(17));
  }
}
```

In the Java code, you can't tell whether you're calling a Java library or a Scala library. Here, we wrapped our methods in an **object** but you can just as easily use **class**es.

This approach allows a Java project to benefit from the advantages of Scala without changing the code base all at once.

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
Second edition (Version 2.0), March 2015
Version 1.1, September 2013
First printing (Version 1.0), March 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