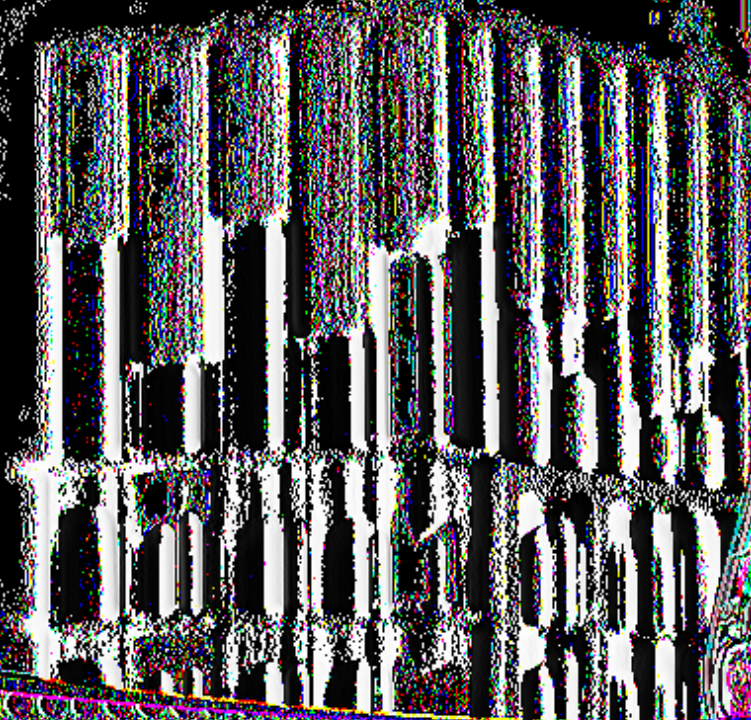




Programming to the Point



Java Programming Interviews Exposed

Noel Markham

www.allitebooks.com

Java[®] Programming Interviews Exposed

Noel Markham



Java® Programming Interviews Exposed

Published by
John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2014 by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-1-118-72286-2
ISBN: 978-1-118-72292-3 (ebk)
ISBN: 978-1-118-72288-6 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permissions>.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or website may provide or recommendations it may make. Further, readers should be aware that Internet websites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at <http://booksupport.wiley.com>. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2013958289

Trademarks: Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Java is a registered trademark of Oracle America, Inc. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

For my wife and best friend, Rebecca.

ABOUT THE AUTHOR

NOEL MARKHAM is a developer with almost 15 years' experience using Java across financial, technology, and gaming industries. Most recently, he has been working for startups in social gaming and digital entertainment. He has hosted interviews for all experienced levels of developer, from graduates to technical leaders, and has set up assessment centers in the UK and overseas to set up full development teams.

ABOUT THE CONTRIBUTING AUTHOR

GREG MILETTE is a programmer, author, consultant and entrepreneur who loves implementing great ideas. He's been developing Android apps since 2009 when he released a voice controlled recipe app called Digital Recipe Sidekick. In between chatting with his Android device in the kitchen, Greg co-authored *Professional Android Sensor Programming* in 2012. Currently, he is the founder of Gradison Technologies, Inc, where he continues to develop great apps.

ABOUT THE TECHNICAL EDITORS

IVAR ABRAHAMSEN is a Software Engineer/Architect with over 15 years of experience working mostly with Java and more recently with Scala. He has worked for small and very large banks, telecoms, and consultancy companies in several countries. Ivar is currently a passionate and opinionated Tech Lead at a startup in Shoreditch, London. For more see flurdy.com.

RHEESE BURGESS is a Technical Lead, working closely with recruitment to fill positions in his current workplace. He has worked with a number of languages and technologies, and has experience working in social, gaming, and financial products, ranging from startups to investment banks.

CREDITS

EXECUTIVE EDITOR

Robert Elliott

PROJECT EDITOR

Ed Connor

TECHNICAL EDITORS

Ivar Abrahamsen

Rheese Burgess

SENIOR PRODUCTION EDITOR

Kathleen Wisor

COPY EDITOR

Kim Cofer

EDITORIAL MANAGER

Mary Beth Wakefield

FREELANCER EDITORIAL MANAGER

Rosemarie Graham

ASSOCIATE DIRECTOR OF MARKETING

David Mayhew

MARKETING MANAGER

Ashley Zurcher

BUSINESS MANAGER

Amy Knies

VICE PRESIDENT AND EXECUTIVE GROUP**PUBLISHER**

Richard Swadley

ASSOCIATE PUBLISHER

Jim Minatel

PROJECT COORDINATOR, COVER

Todd Klemme

COMPOSITOR

Maureen Forys,
Happenstance Type-O-Rama

PROOFREADER

Nancy Carrasco

INDEXER

Robert Swanson

COVER IMAGE

©iStockphoto.com/RapidEye

COVER DESIGNER

Wiley

ACKNOWLEDGMENTS

THANK YOU to James Summerfield for putting me in touch with Wiley to make this book happen.

The team at Wiley has been so encouraging and supportive all through the writing process: Thank you to editors Ed Connor and Robert Elliott, and to the many others behind the scenes.

Thank you to Greg Milette for writing the Android chapter.

Thanks to my colleagues Rheese Burgess and Ivar Abrahamsen for taking the time to edit and review the book.

Thank you to many of my other colleagues at Mind Candy, who have helped in some way: Sean Parsons, Olivia Goodman, Amanda Cowie, and to my good friend, Luca Cannas.

My family: I thank my parents for their unending help and support with everything I've ever done.

Finally, my wife, Rebecca: Thank you for putting up with seeing so little of me for these last few months, and for bringing me all of those cups of tea and other little treats, always smiling. Quite simply, it wouldn't have happened without you.

CONTENTS

INTRODUCTION

xi

PART I: NAVIGATING THE INTERVIEW PROCESS

CHAPTER 1: DISSECTING INTERVIEW TYPES	3
Looking at the Phone Screening Process	4
Reviewing Technical Tests	5
Handling Face-to-Face Interviews	7
Making the Decision	8
Summary	9
CHAPTER 2: WRITING A NOTICEABLE RESUME	11
How to Write a Resume and Cover Letter	11
Writing a Cover Letter	14
Summary	15
CHAPTER 3: TECHNICAL TEST AND INTERVIEW BASICS	17
Technical Written Tests	17
At-Computer Tests	18
Face-to-Face Technical Interviews	19
Summary	21
CHAPTER 4: WRITING CORE ALGORITHMS	23
Looking at Big O Notation	23
Sorting Lists	24
Searching Lists	32
Summary	33
CHAPTER 5: DATA STRUCTURES	35
Lists	35
The Relationship between Arrays and Lists	36
Trees	39
Maps	45
Sets	48
Summary	49

CHAPTER 6: DESIGN PATTERNS	51
Investigating Example Patterns	51
Commonly Used Patterns	60
Summary	64
CHAPTER 7: IMPLEMENTING POPULAR INTERVIEW ALGORITHMS	65
Implementing FizzBuzz	65
Demonstrating the Fibonacci Sequence	67
Demonstrating Factorials	71
Implementing Library Functionality	72
Using Generics	80
Summary	83
PART II: CORE JAVA	
CHAPTER 8: JAVA BASICS	87
The Primitive Types	88
Using Objects	91
Java's Arrays	98
Working with Strings	98
Understanding Generics	101
Autoboxing and Unboxing	107
Using Annotations	109
Naming Conventions	111
Classes	111
Variables and Methods	111
Constants	111
Handling Exceptions	112
Using the Standard Java Library	115
Looking Forward to Java 8	119
Summary	120
CHAPTER 9: TESTING WITH JUNIT	123
The JUnit Test Life Cycle	125
Best Practices for Using JUnit	127
Eliminating Dependencies with Mocks	138
Creating System Tests with Behavior-Driven Development	143
Summary	146

CHAPTER 10: UNDERSTANDING THE JAVA VIRTUAL MACHINE	147
Garbage Collection	147
Memory Tuning	149
Interoperability between the JVM and the Java Language	152
Summary	157
CHAPTER 11: CONCURRENCY	159
Using Threads	159
Working with Concurrency	165
Actors	169
Summary	174
PART III: COMPONENTS AND FRAMEWORKS	
CHAPTER 12: INTEGRATING JAVA APPLICATIONS WITH DATABASES	177
SQL: An Introduction	177
JDBC: Combining Java and the Database	191
Testing with In-Memory Databases	198
Summary	199
CHAPTER 13: CREATING WEB APPLICATIONS	201
Tomcat and the Servlet API	201
Jetty	207
Play Framework	213
Summary	218
CHAPTER 14: USING HTTP AND REST	221
The HTTP Methods	221
HTTP Clients	224
Creating HTTP Services Using REST	226
Summary	230
CHAPTER 15: SERIALIZATION	231
Reading and Writing Java Objects	231
Using XML	234
JSON	240
Summary	243

CHAPTER 16: THE SPRING FRAMEWORK	245
Core Spring and the Application Context	245
Spring JDBC	255
Integration Testing	259
Spring MVC	262
Summary	269
CHAPTER 17: USING HIBERNATE	271
Using Hibernate	
Summary	284
CHAPTER 18: USEFUL LIBRARIES	287
Removing Boilerplate Code with Apache Commons	287
Developing with Guava Collections	290
Using Joda Time	296
Summary	300
CHAPTER 19: DEVELOPING WITH BUILD TOOLS	301
Building Applications with Maven	301
Ant	309
Summary	311
CHAPTER 20: ANDROID	313
Basics	314
Components	314
Intents	315
Activities	318
BroadcastReceivers	321
Services	322
User Interface	326
Persistence	333
Android Hardware	336
Summary	340
APPENDIX: INTRODUCING SCALA	341
INDEX	353

INTRODUCTION

INTERVIEWS CAN BE OVERWHELMING for some, especially when you are being examined, one to one, on your technical abilities.

This book has been designed to give you guidance and preparation for finding your next job as a Java developer. It has been written to help you overcome any sense of fear or worry around Java programming interviews by giving you enough material to practice with so that you can feel confident about any questions you might be asked.

OVERVIEW OF THE BOOK AND TECHNOLOGY

This book is based on Java SE 7. If you are an experienced Java developer, but not up to date with Java 7, following are some new language features and APIs that have been introduced.

The Diamond Operator

Where possible, the compiler can infer the type of a generic instance. This means you can write `List<Integer> numbers = new ArrayList<>()` instead of `List<Integer> numbers = new ArrayList<Integer>()`. This can greatly reduce the amount of boilerplate code when working with collections, and especially nested collections of collections.

Using Strings in switch Statements

The initial `switch` statement in Java could only deal with numeric types in a `switch` statement. The introduction of enumerated types in Java 5 allowed them to be used in a `switch`. From Java 7, `String` objects can be used in a `switch` statement.

A New File I/O Library

Java 7 introduces a new I/O library, with the focus on platform independence and non-blocking I/O.

Many more features have been introduced, including automatic resource management and the representation of binary literals. This book uses Java 7 throughout. You should expect any interviewer to be using the most up-to-date version of Java that is available, so you should aim to keep your skills in line as new versions of the language are released.

Some Java frameworks and libraries are not fully compatible with Java 7 yet, so check with the most current documentation when you use a particular component.

HOW THIS BOOK IS ORGANIZED

This book has been split into three distinct parts.

Part I: Navigating the Interview Process

The chapters on the interview process cover two main topics: how to present yourself to the interviewer, and some general topics that are not specific to Java, but are likely to come up in a technical interview.

Chapter 1: Dissecting Interview Types

This chapter covers how employers often break down the recruitment process, from phone screen interviews, to face-to-face technical tests, to talking with hiring managers.

Chapter 2: Writing a Noticeable Resume

Your resume and cover letter are the first chance to make an impression on what could potentially be your next employer. This chapter covers how to make your resume stand out, with tips on the appropriate language to use and some pointers on what recruiters look for.

Chapter 3: Technical Test and Interview Basics

Any potential employer will want to examine your technical skills, and will try to do so as efficiently as possible. This chapter discusses the different ways programming and technical tests can be applied, and how best to be prepared for each scenario.

Chapter 4: Writing Core Algorithms

Popular topics for technical tests are some core computer science concepts, including sorting and searching. This chapter guides you through some different algorithms around sorting and searching, and discusses the pros and cons of each approach.

Chapter 5: Data Structures

Often paired with the computer science problems of sorting and searching, efficient storage and representation of data is another popular topic for interviews. Chapter 5 discusses lists, trees, maps, and sets, and their representation and use.

Chapter 6: Design Patterns

This chapter covers several object-oriented design patterns, and also shows some example usage within Java's library classes.

Chapter 7: Implementing Popular Interview Algorithms

This chapter takes some popular interview questions, and steps through an implementation in Java. Many of the questions have been taken from the popular technical interview website, interviewzen.com.

Part II: Core Java

The chapters in this section cover most areas that any interviewer would expect to see in an experienced candidate for a Java developer position.

Chapter 8: Java Basics

This chapter covers many of the language features of Java, and can be treated as a recap for any experienced Java developer.

Chapter 9: Testing with JUnit

A core theme throughout the book is the use and focus on unit testing. This chapter introduces JUnit and describes how to use it to check any assumptions and assertions.

Chapter 10: Understanding the Java Virtual Machine

All competent developers have an understanding of the platform they use, and Java is no exception. This chapter covers some features of the JVM, and interaction between the JVM and the Java language.

Chapter 11: Concurrency

This chapter examines Java's threading model and how to use it. It also introduces actors, a more recent approach to concurrent execution.

Part III: Components and Frameworks

This part inspects different domains in which Java is used, from databases to web servers, from some popular frameworks, such as Hibernate and Spring, to the tools to build and distribute enterprise-level applications. It is likely that when interviewing for a particular role, the interviewer will expect you to have specific knowledge presented in some of these chapters, and that would often be part of a job specification, which may have attracted you initially to apply for the role.

Chapter 12: Integrating Java Applications with Databases

Many sufficiently large Java applications will have a database component. This chapter introduces SQL, the language standard for manipulating database data, and how Java and SQL work together.

Chapter 13: Creating Web Applications

Java is a popular language for creating applications to serve data over HTTP. This chapter looks at three popular frameworks: Tomcat, Jetty, and Play.

Chapter 14: Using HTTP and REST

This chapter looks at another use for HTTP: creating and using web services, using a style known as Representational State Transfer, or REST.

Chapter 15: Serialization

Serialization is the method for transmitting structured data. This chapter covers three methods: Java's own serialization mechanism, and platform-independent methods using XML or JSON.

Chapter 16: The Spring Framework

One of the more popular application frameworks, Spring is used by many employers for some, if not all, of their application setup. This chapter looks at several components of the Spring Framework, including the core application context, database integration, and integration testing.

Chapter 17: Using Hibernate

Hibernate is a framework for mapping relational database data into Java objects. This chapter introduces Hibernate, and how to use it to create and manipulate objects.

Chapter 18: Useful Libraries

Java has many useful, reusable libraries. Chapter 18 examines three of the more popular libraries: Apache Commons, Guava, and Joda Time.

Chapter 19: Developing with Build Tools

Any large Java application, particularly one with several developers, will need a well-managed process for building and packaging the application. This chapter covers Maven and Ant, the two most popular tools for building Java applications.

Chapter 20: Android

This final chapter introduces a more modern use of the Java language: developing mobile applications on Android. This chapter looks at the Android SDK, its key components, and how they are integrated.

Appendix: Introducing Scala

This appendix introduces Scala, a new language gaining in popularity with Java development teams because it uses the JVM as its platform. This chapter introduces some language basics, functional programming concepts, and some conventions around immutability.

WHO SHOULD READ THIS BOOK

This book has been written for a Java developer with some experience: You know the language and have been using it for some time, but some of the topics in the chapters may be unfamiliar or completely new to you. If you have never used Java at all, this book may be helpful, especially the chapters in Part II, although you should probably read this in tandem with some more dedicated introductory material.

You may even find this book of some use if you sit on the other side of the interviewer's table: Please use it to take inspiration for some questions to ask in an interview.

TOOLS YOU WILL NEED

You can download the latest Java JDK from <http://www.oracle.com/technetwork/java/javase/downloads/index.html>. Most professional developers will write Java source code using an *Integrated Development Environment*, or *IDE*. Two of the more popular IDEs are IntelliJ (the free Community Edition is available at <http://www.jetbrains.com/idea/download/index.html>), and Eclipse (available at <http://www.eclipse.org/downloads/>). Some of the more experienced interviewers will provide you with a computer and IDE for any technical part of your test, so you should familiarize yourself with the basics of both of these IDEs.

DOWNLOADING THE EXAMPLE SOURCE CODE

All of the example code throughout this book is available on the Internet at www.wiley.com/go/javaprogramminginterviews. You may find it helpful to download, edit, and compile this source code, especially for some of the topics you are less familiar with.

SUMMARY

This book is not a shortcut for getting a job as a Java developer, but should be treated as a companion for finding your next role. It should help with some common topics that interviewers often use to help with their recruitment process.

Experience is very helpful in an interview situation, both with the subject of the interview, and also with the alien, often daunting, interview itself. You must practice and practice the skills and techniques, and your skills will improve as you spend more time as a developer. Gaining experience in an interview situation can be tough. Every employer performs the interview process differently, and the whole process isn't perfect. A potential employer must consider many variables other than the question "is this candidate good enough?" before making a job offer. Budget constraints, team fit, and even the mood of the interviewers all have a part to play.

If you are turned down for a role, try not to get too down about it, but instead try to draw on the experience and consider what will be asked when you have your next interview.

Remember, too, that an interview is a two-way process: Ask about the role, the colleagues, and life in the office. Do not be afraid to turn a job down if it doesn't feel right to you.

Good luck with navigating the interview process, and try to enjoy the ride! Hopefully this book can help you land that dream job.

PART I

Navigating the Interview Process

Part I covers many of the essentials for a technical developer interview, from writing an appropriate resume to more technical topics, regardless of any language specifics.

Chapter 1 discusses the different types of interviews that are used for assessing candidates.

Chapter 2 explains how to write an appropriate resume and cover letter, and highlights the points that interviewers often look for when reading resumes.

Chapter 3 describes how to approach the different interview types.

Chapter 4 covers some basic algorithms, such as sorting and searching. These are popular topics for an interview and are essential for understanding in a day-to-day programming job.

Chapter 5 covers some important data structures. These too are often discussed in interviews, and knowledge of these is essential in most development roles.

Chapter 6 is on design patterns. Design patterns are often a favorite interview topic.

Chapter 7 looks at some useful interview questions, which are often used across a variety of programming languages, not just for Java interviews.

1

Dissecting Interview Types

Simply put, employers conduct interviews to recruit new talent, or to fill a necessary gap to improve a team's productivity. Within a development team or department, you will find a wide range of skills, and on the whole, this is key to making a team gel. It is simply not possible for one person to develop and manage a professional-grade application, there is too much to manage: developing feature requests for product owners, maintaining test environments, and answering any ad-hoc queries from the operations team are all daily tasks for the team managing the development of an application. Development teams will often need one or several application developers. They may even have dedicated front-end developers and database developers. Some teams are even lucky enough to have a dedicated build manager.

Varying ranges of experience are important, too. Having developers who have put several applications live and supported them is key to any new project's success. Graduates, or developers with a couple of years' experience, are also vital: These employees can often bring a different perspective and approach to developing an application, and the more experienced team members can mentor and coach them in the delicate process of developing large enterprise-scale applications within a team of many people.

When it comes to interviewing for new talent, the process and experience for the interviewee can be quite different from company to company. Understandably, it falls on the shoulders of the team's developers to interview for new developers. First and foremost, a developer's day job is to write and produce tested, working applications, and not to interview people. As a result, a developer interviewing people for additional roles may often be under-prepared, and perhaps even uninterested in performing an interview. In a face-to-face interview, this is going to be one of the first hurdles to cross: You need to make the interviewer interested in you.

Companies, and especially technology and digital companies, are getting much better at realizing how important the recruitment process is. Some of the more progressive employers often put emphasis on recruitment within internal company objectives. This puts the responsibility on the employees to make the company an attractive place to work, with the hope that this will attract the top talent, which in turn will bring productivity, success, and profitability.

The first impression for an interviewer will be the resume, sometimes called a *curriculum vitae*, or CV. How to make an eye-catching resume is covered in the next chapter.

As you move through the interview process for a particular role or company, you will encounter different styles and methods of interviews. Generally, the “interview pipeline” for a candidate is designed to be as efficient as possible for the employer, with the face-to-face interviews coming late in the process.

The interview pipeline will usually start with a phone screening, followed by one or more technical tests, and finally some face-to-face interviews.

LOOKING AT THE PHONE SCREENING PROCESS

Companies often start the interview process with a telephone screening. This is advantageous from the company’s side, because it can take a lot of people to organize a face-to-face interview: finding available times for all the interviewers, HR, and possibly a recruitment team, meeting rooms, and so on. This is also helpful for you, because you won’t need to take much time away from work. Many people like to keep it quiet from their current employer that they are looking for jobs, so it shouldn’t be too hard to find a quiet corner of the office or a meeting room for an hour to take a phone interview.

If you do an interview over the telephone, make sure you are prepared well before the time of the interview. If you *have to* take the call while at work, book a quiet meeting room or find a quiet corner of the office. If that is not possible, you could go to a local coffee shop. Make sure beforehand that the noise level is appropriate: You don’t want to be distracted by baristas calling across the café or loud music playing in the background.

If you are expecting to do a remote live-coding exercise, make sure you have Internet access wherever you are. Have a pen and paper handy in case you want to make notes during the call for questions to ask later. Use a hands-free kit when typing: It will make the call much clearer for the interviewer. You don’t want to waste precious time repeating questions or re-explaining your answers—this is very frustrating for all involved.

It might not even hurt to have a few notes in front of you for any topics that may come up during the call. This is not cheating; it can help calm any nerves and get you settled into the uncomfortable, alien experience of having to convey technical explanations over the phone to someone you have never met before. Note, however, that if and when you have an interview face-to-face with the team, you won’t be able to have any notes in front of you.

Usually a phone screen lasts for 30 to 60 minutes, and you should expect some very high-level questions. The kinds of questions that come up in a telephone interview are often about language-agnostic algorithms. You may be asked to verbally describe these, or you may even be asked to attempt some live-coding challenges in a shared, collaborative document editor, such as Google Docs, or perhaps a bespoke interview-hosting website, such as Interview Zen (www.interviewzen.com).

At any point in the interview process you should know the basics about a company. Look at the “about us” page on their website. Read their blog; find out if they have a Twitter account. If you are applying for a small company or a startup, try to find out some more about the senior members

of the company, such as the CEO and CTO. They can often be quite active in a local development community.

At the end of any interview, the interviewer will often ask if you have any questions for them. Answering *no* to this question will leave a bad impression; it will show that you do not care enough about the role. Considering you now know that this question will more than likely come up, think about what questions you want to ask long before you even go for the interview. Think about what you want to know about the team dynamic, how the team works together, and what the office environment is like. Only you can come up with these questions. This is not the time to talk about the salary or anything else to do with the package that comes with the job. There will be plenty of time to talk about that once you have been offered the role.

REVIEWING TECHNICAL TESTS

A technical test can be used as a supplement to or instead of a phone-screen interview. This may even happen as part of a phone screening, or you may be invited to the office for a technical test.

A technical test often consists of some simple questions covering a multitude of areas related to the nature of the role. If the role is for working on a web application, you could expect questions about the Servlet API and perhaps some frameworks such as Spring MVC or Tomcat. You should be aware of the nature of the role, and any languages and frameworks used.

These tests are usually performed alone, and can take a variety of approaches. Some interviewers rely on a pen-and-paper test, often asking some simple definite-answer questions, or others may ask you to write a simple algorithm, often around 10 to 20 lines long.

If you find yourself writing code on paper, you should take extra effort to make sure the code is understandable. Any decent employer should understand that you may make some elementary mistakes around method naming, or miss a semicolon, but you should always aim to give no excuse to an interviewer to not ask you back for any further interviews.

Another popular technique is to provide a coding test on a computer with a functioning IDE. This is a fairer way of testing, because it is a similar environment to how candidates work when doing their day job. They may not provide Internet access, but may provide offline Java documentation instead.

When you are tested with a fully functioning IDE, or even just a compiler, you have no excuse for producing code that does not compile.

Whatever you are asked to do, write unit tests. Write them *first*. Although this may take additional time, it will ensure that the code you do write is correct. Even if a test makes no mention of writing unit tests, this will show to anyone examining your test that you are diligent and have an eye for detail. It will show that you take pride in your work, and that you think for yourself.

For example, method names in JUnit 4 and onward are free from any kind of naming convention (as discussed in Chapter 9), so if you saw a question similar to the following: *Write a simple algorithm to merge two sorted lists of integers*, you could quickly start by writing a list of test cases, and these will then become the method names for your JUnit tests.

Covered in depth in Chapter 9, method names in JUnit 4 and onward are free from any kind of naming convention, so you could quickly start by writing a list of test cases, making them method names for JUnit tests. Some example test cases for the given question include:

```
twoEmptyLists
oneEmptyListOneSingleElementList
oneEmptyListOneMultipleElementList
twoSingleElementLists
oneListOfOddNumbersOneListOfEvenNumbers
oneListOfNegativeNumbersOneListOfPositiveNumbers
twoMultipleElementLists
```

The contents of each of these test cases should be easy to understand from the name. Given a well-defined API, writing the test cases should take no more than five to ten minutes, and you can run these tests as a frequent sanity check when implementing the actual merge. This will keep you on track as you are writing and refactoring your real code.

These given test cases probably do not cover every possible code path for merging two sorted lists, but should give you and the interviewer confidence in your actual implementation.

Most modern IDEs allow you to automatically import the JUnit JAR without needing any Internet access at all; it comes bundled with the IDE. Make sure you know how to do this for a few IDEs, because you may not get a choice of which one to use in a test situation. At a bare minimum, you should be able to do this for Eclipse and IntelliJ. They are both smart enough to recognize the `@Test` annotation and prompt you to import the JUnit JAR.

Try to write test cases concisely and quickly. Any reasonable interviewer will not mind the odd missed test case, understanding that you want to show the range of your abilities in the time given.

If you are ever given a paper test and a test on a computer, you have the advantage of being able to check your code on paper by writing it on the laptop! Some companies do take this approach—they will provide some simple, exploratory Java questions on paper, and then ask you to write a more in-depth, small application.

Considering you are applying for a job as a developer, it is reasonable to expect to demonstrate your technical ability during the interview process. It is worth noting that this is not always the case—interview standards can be so varying that it is not unheard of to have a brief chat with a hiring manager, and that is all. This approach does not normally work out well for the employer, and thankfully this does not happen as often as it used to.

Like all parts of the interview process, you must be prepared. The job specification page should contain a lot of technical information about what languages, technologies, and frameworks are used. A job specification page should be available on the company's recruitment website pages, or if you cannot find it, you should ask the company for it. You should not be expected to know every bullet point on a job specification inside and out, but the more you can show, the better. Employers will be looking for a willingness to learn and adapt, so make sure you convey this in an interview. Most importantly, you must *want* to learn the technologies the company uses.

Again, understand what the company does; you may even be asked specifically for what you understand about the company, or specific applications developed by the team you are interviewing for. If

the company has a public-facing website, make sure you have used it. If the role is part of a game, make sure you have played it. Some employers have a rule that candidates are a definite “no” if they have not used the product developed by the team.

HANDLING FACE-TO-FACE INTERVIEWS

Following a successful technical test, you should be asked for a more personal interview, meeting members of the immediate team you would be working with as well as a hiring manager, and perhaps some members of a recruitment team or human resources. This can often happen over the course of half a day, or a full day. It may even be set over several meetings on several days, usually depending on the office location—how easy it is for you to keep returning for more interviews—the availability of the interviewers, and your availability, too.

As a rule, the interviews will get less technical as the process progresses. A typical set of interviews may be as follows:

- A technical interview, covering any work done in a pre-screening or an individual test
- Several more technical interviews, covering more topics and questions of interest to the hiring team
- You may be expected to write some code on a whiteboard or to do some pair programming.
- An interview with one or several members of the business side of the team, typically a product manager or project manager
- An interview with a hiring manager. This will cover some softer skills, away from the technical side. This is to see if you are a good fit for the team.
- A final debrief from an in-house recruiter or HR representative. Salary expectations and other contractual conversations may happen here, as well as any information about next steps.

In preparation for any upcoming interview, try to think about possible questions that will be asked. If you are expecting to cover any code you wrote in a phone screening, online, or in a previous technical test, try to remember what you actually wrote. It may help to try to write out from memory any small applications that you wrote as part of your test. It does not matter if it is not exactly as you submitted it the first time, but it will help jog your memory, and you may find some improvements to discuss in the interview.

When doing any face-to-face interview, be prepared to be stretched to the limit of your ability. Answering with “I don’t know” is not an admission of failure; it is simply that the interviewer is trying to understand the limits of your knowledge. Of course, the later you reach that point, the better. However, you cannot expect to be an expert in all domains, so there should not be much concern if you can demonstrate a deep understanding of, say, Spring and Hibernate, but your knowledge of database transactions and concurrency is not up to the same level.

The interview process itself is not perfect. This is not an examination; just knowing the right answer is not enough. The interview team is checking to see if you will be a valuable addition, perhaps

looking to see if you would be a suitable mentor if you are interviewing for a more senior role, or if you show drive, determination, and passion if you are trying for a junior role.

Sometimes even this is not enough; perhaps the interviewer is in a bad mood, or having a bad day. The interviewer could even be mentally distracted from a broken build that happened just minutes before stepping into the interview. Unfortunately there is not much you can do in these circumstances. Always be polite and try not to upset the interviewer during any discussions.

Often late in the process, once the developers are happy that your ability matches their expectations, you will meet with the hiring manager. The role of this interview is to discuss your fit within the team, talk about some of the expectations and softer skills that are required for the role, and discuss your own career development within the company.

Like in all steps of the process, your manner and engagement are being examined here. The hiring manager wants to see that you are keen, smart, and able to hold a decent conversation. It is very rare that a developer sits alone and writes code all day with no interaction with anyone else. Talking to other developers, product managers, and many other people is necessary on a daily basis, and one of the main points of this conversation is to ensure that your communication with others is at an acceptable level.

Sometimes, if the interviewing team feels that you are not quite the right fit, they may terminate the interview early, informing you that they are not going to pursue the role any further. If this happens, it can be quite a shock, especially if you are not expecting it. Be assured that it is nothing personal, and is not a slight against your ability; it is purely that your skills do not match up with what they need. Try not to let this affect your confidence, and ask yourself if there is anything you need to brush up on before any interviews with a different employer.

MAKING THE DECISION

Being offered a role is not the end of the process. First, you must be absolutely sure that you want the role. If you have any doubts, even after an offer stage, approach the hiring manager or recruitment agent. They can often allay any fears you may have, or fill in the answers to any questions you didn't ask during the interview process.

Even after any further discussions, if you still feel that the role is not right for you, do not take it. You would be better to carry on looking rather than taking the job, being unmotivated, and then looking to move soon after, or worse, not making it through any probationary period.

If you are happy with the role, you will be offered a salary and perhaps other benefits as part of your employment package, such as stock options, a pension, and medical benefits. You may have some room to negotiate on the package at this point. Remember, the hiring manager and the rest of the team now want you to work for them. They have invested a lot of time and effort in examining you, and if they cannot provide you with an attractive offer now, they will have to start the process all over again. Do not push your luck, though: A developer salary and package will fall only within a certain range in the team's budget.

SUMMARY

If you are inexperienced with interviews, they can be quite daunting. The best way to remove any fear is simply with experience; the more interviews you have under your belt—successful or otherwise—the more you understand exactly what interviewers are looking for, and how to tackle the questions given. Although the questions in this book will provide a good insight into the types and styles of questions you might encounter during an interview, the actual execution of answering these questions effectively will only come with practice and experience.

Remember at all times that interviews are a two-way process. Although the interviewing team dictates how and when the interview process happens, you are interviewing the team, too. It is solely up to you to decide if you actually *want* to work for this team. Make sure you answer that question before you commit and move jobs.

People can find rejection hard from interviews, especially when they have had much success in academic life. Interviewing for a job is not the same as taking a university or college exam; getting the question right is not always sufficient. Every single aspect of you is being examined, from your personality to your skills, but also whether your skills match with what the hiring team needs. There is no shame in being rejected for a role. Getting the questions right is only half of the battle.

The biggest key to acing any interview is preparation. Make sure you understand the domain properly, and try to visualize what exactly the interviewers are going to want from the interview. Think about the questions they may ask, and make sure you have some questions of your own to ask them.

Chapter 3 visits the different interview types discussed in this chapter, and delves a little deeper into your conduct and how to produce good and relevant answers. Before that, however, Chapter 2 looks at the very first stage of the hiring process, before any interview takes place: how to write an appropriate resume and cover letter to get noticed by the recruiting team.

2

Writing a Noticeable Resume

A common joke amongst recruiters and interviewers is that they always throw half of the resumes in the trash without reading them, because they do not want unlucky people working for them.

Getting your resume noticed can be quite hard, and may involve a little luck. The advice and writing technique shown in this chapter will hopefully reduce the reliance on luck as much as possible.

HOW TO WRITE A RESUME AND COVER LETTER

The written part of a job application generally consists of two parts: a resume and a cover letter. A resume is an account of your professional qualifications and other relevant skills, whereas a cover letter is a personal letter to the interviewer that describes how the specific role you are applying for suits you, and what extra qualities you can bring to the team along with the role.

Even though your resume should appear generic, you should tailor it to a particular job. When you submit your resume, make sure you tweak it in favor of the job you are applying for, focusing on your most prominent skills that match the job specification.

A resume should never be more than two pages in length. Most interviewers are developers in their day job, and simply do not have time to trawl through pages and pages of a career history. Keep it concise and relevant, and concentrate on your most recent roles.

With this in mind, you should make sure that any key information you definitely want an interviewer to see is toward the top of the first page. Focus on the key strengths you want to convey here. Do not waste space by adding a title that says “Resume” or “Curriculum Vitae”—everyone reading this document knows what it is. One of the most important things you want the interviewer to remember is your name; make this as your heading, and make it the largest font on the page.

Some candidates now put a small headshot photograph on their resume. This can help interviewers put a face to a name when discussing candidates after performing several interviews,

but it is up to you whether you want to include a photo; it will not have any overriding positive or negative effect on your chances of a job offer.

A common approach nowadays is to have a developer profile at the top of the page. This is a short paragraph, perhaps three or four lines, that sums up your aspirations and career to date, and describes what you are looking for from your next role. An example profile might say:

An experienced Java developer with four years experience in the retail and banking industries. Always keen to learn new technologies, and looking to take more leadership over project delivery.

By placing this toward the top of the page, the likelihood of an interviewer reading this statement is high. The profile shown here is written in the third person. It is purely a matter of preference if you write it in the first or third person, but, like the rest of your resume, make sure you are consistent.

Something else to include early in the document is a section on key skills. This can be a list, perhaps a set of bullet points that you see as your most employable traits.

Whatever you write here, make sure you cover as much ground as possible. If you have experience in languages other than Java, mention them. Say what operating systems you are experienced with. Mention which tools you use, such as the IDEs you prefer, as well as other aids to your role as a developer such as source control systems.

Do not just write a list of acronyms. Do not just write a list of every library and tool you have ever touched. Interviewers will not pay much attention to several lines of acronyms—“SQL, XML, JDBC, HTTP, FTP, JMS, MQ,” and so on. It is much better, and more engaging for the interviewer, if you describe your key skills as:

Expert with Spring JMS and the HTTP protocol, some experience with Apache Cassandra.

Whatever you decide to include in this list, make sure the items are appropriate. Any key skills you list should be reflected elsewhere on your resume, such as in your description of previous roles.

Include a small section on your education and qualifications. Start with your most recent qualification. Include the dates you attended the institutions. If you have a university or college degree, include the titles of any theses, dissertations, or papers you submitted. You do not need to be too detailed about your high-school qualifications. Include a single line highlighting when you attended the school, and any key examinations.

If you do not have a university qualification, be a little more detailed about your grades in key subjects, such as math or English.

As you gain more experience, you can place less emphasis on the education section in your resume. If you are applying for senior or leadership roles, employers will be much more interested in seeing evidence of what you have achieved in the workplace. Regardless, you should always include your highest qualification.

Unless you have just left school, the majority of your resume will be made up of your work experience. Treat each role as a different section, and for each role, include the dates and locations of

where you worked. You should include a brief description of your responsibilities and what you achieved. Where appropriate, mention any technologies or libraries you used. Similar to your key skills section, summarizing your experience with bullet points is fine: Any interviewer will want to read and understand your experience as quickly as possible.

Remember that the interviewers are interested in what *you* have achieved, what *your* role was in any team success. Rather than describing a role as:

I worked on the team that released the game XXX. It received two million daily active users within the first month.

It would be much more informative to say:

I was responsible the development of the server for game XXX, focusing on its stability. It handled two million daily active users within the first month, with no downtime.

This way, you have made it clear what your role was on the team, what your responsibilities were, and what was achieved thanks to your input.

Where appropriate, try to use powerful adjectives when describing your work. By describing yourself as a “lead” developer, or having the “sole responsibility,” or being a “core team member,” you are emphasizing your value to any work within a development team. Remember that you must be able to validate your descriptions.

If you have any gaps between employment dates in your resume, you *must* be prepared to talk about them in an interview. If you decided to take some time off and go travelling, that is fine; if you were made redundant or were laid off and it took you several months to find a new job, that is fine, too. But employers will be interested in knowing what you did with your time off. Prospective employers will be looking for candidates who are productive, forward thinking, and proactive. If you can truthfully say you were helping with some open-source projects or doing some kind of charitable work, this will put you in a lot better favor over someone who cannot explain what he or she did with any time off.

Even if you tried to start your own business and it failed, there is no shame in this at all. In fact, most employers would see this as a positive personality trait.

If you have any appropriate online presence, providing links is often attractive to employers. If you have a Github profile, a link to it will give employers additional insight into how you work. If you write a blog, include a link to that, too. If you answer questions on question-and-answer sites such as Stack Overflow (stackoverflow.com), provide links to some of your popular answers or to your profile page. This is very informative as to how you work and think in a “real-world” environment, rather than the strange, alien environment of a set of interviews.

You should expect any interviewers to perform an online search for you. If you have a Twitter account or Facebook account, or similar account that is easily traceable to you, make sure the content posted is something you are comfortable sharing in an interview. If it is not, make sure you have the privacy settings configured appropriately.

Candidates often include a small section toward the end of a resume of some personal interests. This can have very little bearing on a decision to call someone in for an interview. Although it can show

that you are an interesting individual who does more than just work, a decision to interview someone will be made on the relevant professional credentials on a resume. If you are struggling to fit the contents of your resume into the hard two-page limit, leave this off.

Some candidates also include one or two references with their contact details. There is no issue if your resume makes no mention of references, but if you choose to include them, make sure they are up to date. Speak to your references beforehand, and make sure they are prepared to expect a call and talk about your best qualities. Failure to do so will reflect poorly on your reliability.

WRITING A COVER LETTER

The cover letter is a chance to show that you have done your homework about the role you are applying for. It should be relatively brief, much less than one page, with the aim of persuading a recruiter or interviewer to invite you in for an interview.

Use this letter to highlight any key entries on your resume that you feel are relevant for the role. Also mention your most recent role and what you did there. If a current employee referred you, mention him or her by name: Many companies offer a referral bonus for any successful hire. You want to make sure to follow through on this—if you are hired, you might even get a cut of the bonus!

If you are not currently employed, the cover letter is an ideal place to say what you have been doing since you left your most recent role.

You should also state why you want to join the company you are applying to. If it is in a different industry, explain why you would like to move industries, too.

Say why you are looking for a new role. There is nothing wrong with wanting to leave to develop yourself and your career, but you must be able to articulate that in the letter and in any subsequent interviews. You should also mention any softer skills you feel you would bring to the role.

Even if you have contact details on your resume, include a line in your cover letter explaining how best to get in touch with you. You want to leave no excuse for not being invited in for an interview.

The body of an example cover letter could look like:

Please find my resume attached for the software developer role.

My current role is as a server-side Java developer for Company X. I have been at the company for three years and worked on two different teams as a Java developer in that time.

I feel that both Company X and I have gained a lot from my employment there, and I feel it is time for me to move on and apply my technology skills in a different industry, and hopefully gain some new ones too.

I am confident I can bring my enthusiasm and experience to Company Y and to this role. I look forward to hearing from you soon.

It is these small attentions to detail that may differentiate your job application from another candidate's.

SUMMARY

Your resume is the first thing that any prospective employer will see about you. It is an advertisement, your brand. You have full control over this document, and so you should make every effort to make sure it is perfect.

There should be no mistakes at all on your resume. Double-check all the dates, make sure there are no spelling mistakes, and make sure the grammar is correct. Simple errors on your resume will give the impression that you do not care, and, therefore, can't be trusted to produce professional-grade, properly tested applications.

Keep the formatting consistent. Your resume does not need to have fancy headings and several typefaces: You are not applying to be a graphic designer. Keep your formatting clean, simple, concise, and easy to read.

Get others to check your resume, and take any feedback from them constructively. Read and re-read it several times; you will always find minor adjustments and other small changes to make.

Schedule a small amount of time every two to six months to keep your resume up to date. You will find it easier to remember any key points you want to highlight. Also, updating your resume more frequently means you will be making fewer changes at any one time, so the chances of making an obvious error or typo are limited.

Always try to keep your resume to two pages at maximum. This allows an interviewer to print it on a single piece of paper, double-sided. Over time, as your experience grows, you will have to be judicious in what you decide to keep and what you decide to remove to keep the document under a two-page limit. This can be hard at times, and you may find yourself fiddling with margin widths and font sizes to fit on as much as possible, but sticking to a two-page limit is key. Interviewers simply do not have the time or the interest to read an eight-page resume.

If you take only one thing from this chapter, be aware that anything you write on your resume is fair game for an interviewer. Do not lie; do not say you have skills or experience where you do not. If, or perhaps, when, any falsehoods are uncovered in an interview, you can almost be certain you will not be receiving a job offer. Even if you manage to get through an interview with a false resume, most employers will perform thorough background checks, and any wrong qualifications or false roles will be uncovered there.

The next chapter looks at how to handle assessments once an interviewing team is happy with your resume, and would like to find out if you would be a good fit in terms of skills, knowledge, and personality.

3

Technical Test and Interview Basics

Undertaking an interview for a Java developer will inevitably require an employer to test your technical ability.

This chapter covers some of the testing formats used by interviewers to see if your programming skills match what is required for the role they are trying to fill.

The questions provided throughout this book will usually be asked within one of the interview formats discussed here. If you have never had to do a technical interview, or it has been a while since you last did one, this chapter should set an expectation of how recruiters assess candidates.

TECHNICAL WRITTEN TESTS

Technical written tests are often used as a screening technique for interview candidates, and are usually done before any face-to-face interview. Candidates will be asked to take a short, written test so that the interviewing team can assess whether they are worth interviewing.

This is beneficial to the interviewing team, because they can get a good idea about the ability of the candidate without having to put in much effort. Quite often, interviewers can find it hard to differentiate candidates from the resume alone, because many resumes consist solely of lists of technologies used in previous roles. This should be a motivator to make sure your resume stands out from the list of average resumes a recruiter receives.

Offline, written-on-paper tests are not very good ways of assessment, because this is not the way you would usually write code. You have no IDE, no compiler, no Internet access to rely on; it is purely reliant on what is in your head.

Although it is not a very good method of assessment, it is still a popular recruitment technique, because of the minimal investment needed from the development team.

Whenever you are asked to take a written test, try to follow the same advice and guidelines you were given when taking exams at school. Make sure you read and understand the entire question before attempting to answer it.

Take a look at all of the questions first, so that you don't get any nasty surprises as you are running out of time. The better written tests will have questions that increase in difficulty, meaning you should be spending more time on the later questions.

Sometimes, a written test will have more questions to answer than is physically possible in the time allowed. Don't feel too deflated if you cannot answer all the questions. The assessors want to see just how quickly you work and figure out the answers, to understand the speed of your problem-solving ability.

Write legibly. It is highly unlikely that you will be hired for the quality of your prose, so if you are asked questions that require an explanation, do not be afraid to be terse and to the point with your answers. Use bullet points if necessary; just make sure you answer a question as completely as possible.

Writing code on paper is hard. It is very simple to forget a semicolon, the ordering of arguments to a method, or similar things you take for granted. Refactoring involves scribbling out method or variable names, or perhaps starting again with whatever you have written. Hopefully your interviewer will take this into account and accept that code written on paper will never be perfect.

If you are asked to sketch an algorithm on paper, try to think it through in your head first, before committing pen to paper. Do not be afraid to make a few attempts before being happy with your implementation: After all, that is how many developers write code in the real world.

AT-COMPUTER TESTS

A slightly more professional approach used by interview teams is to ask you to complete a test by writing real, compiled code. Sometimes a recruiter will send you some questions to do at home, or perhaps ask you to come into the office to take a test.

This method should be a little closer to how you write code in real life, and so, any interview nerves aside, you should feel a little more comfortable with this type of assessment.

If you are asked to perform the written test at home, try to mimic your own working conditions as much as possible. You will have full Internet access for any documentation, and you will have, within reason, much more time to do the test than you would in a one- or two-hour interview slot in the offices of the company you are interviewing for.

If you are asked to go in to the office, and, if you are lucky, the computer you are using for any test will be connected to the Internet, you will be able to look up any Java API documentation if necessary. If you do have Internet access, resist the urge to rely on search engines or question-and-answer websites—try to attempt the questions using your own ability as much as possible.

Some at-computer tests are performed offline, but they will give you access to Java API documentation stored on the computer's hard drive.

Hopefully an at-computer test will make use of an IDE, although you may not get a choice, so make sure you are familiar with the basics of the more popular IDEs, including Eclipse and IntelliJ. Most IDEs are also shipped with the Java API documentation for the standard Java libraries, so you should be able to look up how certain methods work directly in the code.

Of course, the best thing about using a modern IDE is that it will check and compile your code as you type, so you can fix any trivial issues quickly.

Sometimes a test will ask a series of questions, and you will be provided with an interface, or method signature. You will not be able to change this interface because it is likely your code will be part of a larger application that needs that interface to run, such as a web application server.

Alternatively, the reason for not being able to change any method definitions will be because the interview team may have a suite of unit tests to run against your code. This is another practical approach from the interviewers—they are able to see if your code meets the test requirements quickly, without much input from them.

From their point of view, making the tests pass is not the full story. They will also check any successful code for its quality, making sure the code is professionally formatted, any reasonable comments included, and just to get a general feel for how you write code.

You may be asked a less rigid question, where you have no predefined method template or interface to work from, such as writing a small application from scratch. It can be a lot more daunting to get started when faced with an empty file, rather than filling in the blanks on a template with the method name, return type, and parameters filled in.

If you are not sure how to start, some people find it helpful to draw some diagrams of the components of the application, and how they expect it to work and fit together. This can then trigger the creation of any domain objects and the application logic of how they all fit together.

Regardless of your approach, make sure you write tests. Even if you have not been told to write tests, you should do it anyway. Tests will guide the writing of your application, and you will find that as you are pressed for time, the tests will keep checking that all your code fits together as you expect. If any refactoring prompts test failures, you can quickly roll that change back, rather than carrying on obliviously and then having recruiters reject your test straight away because it didn't work as expected.

Most IDEs have the JUnit jar packaged within the application and the ability to auto-import the jar into the project when it sees you are writing a JUnit test case, so do use that if you can. Most interviewers will be impressed, especially if you have not been explicitly asked to write tests.

FACE-TO-FACE TECHNICAL INTERVIEWS

Often after some solo technical assessment, you will have one (or several) interviews face-to-face with other developers. Often, they will ask you some technical questions, and will ask you to answer by writing out what you would do on paper, or perhaps on a whiteboard.

This can be quite a strange environment if it is not something you have ever done before, or if you are used to the comfort of an IDE and compiler.

Interviewers will often ask you to write in a specific language, or perhaps in pseudocode.

WRITING PSEUDOCODE

If you are unfamiliar with pseudocode, it is a high-level language that is supposed to mimic how an application or algorithm would be written in an imperative style. The actual syntax does not matter too much; it is used for discussion, or to write ideas down before implementing them in a real language.

For example, writing an algorithm to find the maximum value in a list of positive numbers may look like this:

```
int largest = -1
for i in 1 to length of list
    if list(i) > largest then largest = list(i)

if (largest == -1) print "No positive numbers found in the list"
else print "Largest number in the list: " + largest
```

There are no semicolons, and there is no formal syntax; the use of `for` and `if` statements rely on the reader having some experience as a programmer.

This example could have been written in a number of different ways, but the most important thing is that everyone who reads it understands the intention.

Pseudocode is used in situations other than interviews. Technical specifications may use it to provide an outline of how an algorithm should work, or sometimes blog posts or technical articles may use pseudocode to keep any examples language-independent.

If you are asked to sketch out an algorithm on paper or on a whiteboard during an interview, and you are asked to write in Java (or a different language, for that matter), try to get the syntax as correct as possible. Try to remember all semicolons, and try to get any API library method names correct. However, the interviewer should not expect this to be perfect; after all, that is why you use a compiler in the first place.

Usually the approach for an active offline coding interview is that the interviewer will give you the requirements for an algorithm, and will then leave it to you to solve.

One useful approach is effectively to think out loud. Build up your answer, providing a commentary as you do. The interviewer will usually be involved, so treat it as a conversation. If you are not sure about a certain approach, make this clear. There is no harm in going down a wrong path, if as soon as you realize, you can explain why and what changes you are making because of it.

More often than not, an interviewer is interested in your thought process just as much as the correct answer. If you are asked a question you have had before, and you write out a perfect solution in 60 seconds, the interviewer won't be able to measure your problem-solving ability. It would be much better to let an interviewer know that you know the solution to the given problem.

Do not be afraid to ask questions. When the interviewers asked the question at the start of the interview, they were merely setting the problem in motion. Often there is more than one acceptable solution to a given problem, so asking appropriate, well-thought-out questions can help guide you to an acceptable solution.

When fleshing out an algorithm on a whiteboard or on paper, write quickly, and again, be sure to be legible, too. If the interview team looks at your code after interviewing other candidates all day, and they cannot read what you have written, they won't be able to make a positive decision about your interview.

SUMMARY

Whenever you are writing code for an interview, you will often be under pressure from the clock. This is for several reasons: It is simply practical for the interviewer, and for you, to have the interview process time boxed. They will have other people to interview, as well as their day jobs. You may have other interviews, or other things to do in your daily life. More importantly, interviews and coding tests are conducted within a time limit to see just what you are capable of achieving in the time given.

You can find several online resources for practicing test questions. Look around on the Internet for some sample tests to do in a restricted time frame to get yourself in the mindset. You may even find the challenges fun!

Whenever discussing a technical test implementation with an interviewer, one question to always expect is, "Does the code you have written work?" If you don't explicitly write tests as part of the interview, try to think of some test cases for your code as you write it, because this will form a productive discussion toward the end of the interview.

Think through the code you have written, think about any improvements you would make, and be prepared to talk about those in an interview. Think about any limitations of the code, even limitations supplied by the given questions. One common question is to ask what you would change in your code to scale to several million users.

Preparation is key: Try to put yourself in the shoes of the interviewer. What would *you* ask?

The next chapter covers algorithm basics, sorting and searching in particular. These core computer science topics are often the basis for common technical interview questions.

4

Writing Core Algorithms

Many of the algorithms you are asked to define or demonstrate in interviews are operations on lists, usually sorting or searching. This chapter examines several ways to assemble a list into an ordered list, and discusses the advantages of each approach. It also looks at a common approach to searching lists for a given value.

The algorithms described here are often covered in a core computer science course, so if you have ever studied computer science you will probably be able to treat this chapter as a refresher.

LOOKING AT BIG O NOTATION

A formal approach to describing the performance or complexity of an algorithm is called *Big O Notation*. This describes how an algorithm performs as its input changes.

For instance, an algorithm described as having a worst-case performance of $O(n^2)$ means that as the size of the input doubles, the algorithm takes four times longer to run. An $O(n^2)$ algorithm is often not the most efficient implementation, although this is entirely dependent on the exact goal of the algorithm.

Big O tends to be applicable for larger values of n . When n is small, the running time or space consumption of an algorithm is often negligible. Regardless, for any algorithm you do write, it is always worth considering Big O notation, even for small values of n , as, over time, you may find you are running your algorithm with larger and larger input.

An algorithm often has three values of complexity: best-case, worst-case, and average-case. As you would expect, a best-case performance is how the algorithm performs when the input given means the algorithm does as little work as possible.

The performance descriptions here are often called the *time complexity* of the algorithm. Algorithms have a *space complexity*, too; that is, how much extra space the algorithm needs to do its work.

There is often a trade-off. You may find that the best performing algorithm uses much more space than you can afford, or that the algorithm that uses only one extra piece of memory is much less efficient than alternatives that use more space.

With any algorithm you write, try to think in terms of both time and space complexity. Although computer memory is cheap nowadays and often large enough that it is not a concern, it is a good exercise to understand what exactly your algorithm is doing.

SORTING LISTS

A common operation on collections—lists in particular—is to rearrange the list into some kind of sorted order. Lists are often sorted using *natural ordering*, such as sorting numbers from lowest to highest or sorting characters into alphabetical order. However, you may have different requirements for your sorting. Java provides two interfaces for helping with sorting: `Comparable` and `Comparator`.

What is the difference between the `Comparable` and `Comparator` interfaces?

Because these interfaces are public, they can be put to any use. By convention, the `Comparable` interface is used for natural ordering, and `Comparator` is used to give exact control over the ordering.

When sorting an array, you generally use a built-in library, such as the implementation included in the `Arrays` and `Collections` classes. However, for an interview you may be asked to write your own implementation.

The `Arrays` and `Collections` classes have several overloaded `sort` methods, which can broadly be split into two: one that takes an array as a parameter, and one that takes an array and a `Comparator` object. The method is overloaded for each of the primitive types and one for the reference type (`Object`).

For the implementation without a `Comparator` object, the type is sorted by its natural ordering. Listing 4-1 shows this for `ints`, sorting the array from low to high.

LISTING 4-1: Naturally sorting an array of `ints`

```
@Test
public void sortInts() {
    final int[] numbers = {-3, -5, 1, 7, 4, -2};
    final int[] expected = {-5, -3, -2, 1, 4, 7};

    Arrays.sort(numbers);
    assertEquals(expected, numbers);
}
```

For an array of `Objects`, the type being sorted must implement the `Comparable` interface, as shown in Listing 4-2.

LISTING 4-2: Sorting objects naturally

```

@Test
public void sortObjects() {
    final String[] strings = {"z", "x", "y", "abc", "zzz", "zazzy"};
    final String[] expected = {"abc", "x", "y", "z", "zazzy", "zzz"};

    Arrays.sort(strings);
    assertEquals(expected, strings);
}

```

The `String` class implements the `Comparable` interface, so the sorting works as you would expect. If the type being sorted does not implement `Comparable`, this will throw a `ClassCastException`.

For your own class definitions, you will need to implement `Comparable`, such as the implementation described in Listing 4-3.

LISTING 4-3: Sorting without a Comparable interface

```

private static class NotComparable {
    private int i;
    private NotComparable(final int i) {
        this.i = i;
    }
}

@Test
public void sortNotComparable() {
    final List<NotComparable> objects = new ArrayList<>();
    for (int i = 0; i < 10; i++) {
        objects.add(new NotComparable(i));
    }

    try {
        Arrays.sort(objects.toArray());
    } catch (Exception e) {
        // correct behavior - cannot sort
        return;
    }

    fail();
}

```

It is not possible to use the `Collections.sort` method because the compiler expects the type of the parameter to be an implementation of `Comparable`. The method signature is:

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

If you want to provide your own ordering, you provide an implementation of the `Comparator` interface to the `sort` method. This interface has two methods: `int compare(T o1, T o2)` for the

implementing type `T`, and boolean `equals(Object o)`. The `compare` method returns an `int` in one of three states: negative if the first argument is sorted before the second, zero if the two are equal, or positive if the second argument is sorted first.

If you were to provide a reverse numerical order `Comparator`, the implementation may look like Listing 4-4.

LISTING 4-4: A reverse numerical order `Comparator`

```
public class ReverseNumericalOrder implements Comparator<Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o2 - o1;
    }
    // equals omitted
}
```

Listing 4-5 uses this `Comparator`.

LISTING 4-5: Using a custom ordering

```
@Test
public void customSorting() {
    final List<Integer> numbers = Arrays.asList(4, 7, 1, 6, 3, 5, 4);
    final List<Integer> expected = Arrays.asList(7, 6, 5, 4, 4, 3, 1);

    Collections.sort(numbers, new ReverseNumericalOrder());
    assertEquals(expected, numbers);
}
```

For the examples in this chapter, a simple natural ordering is assumed.

How would you implement a bubble sort algorithm?

The bubble sort algorithm is extremely simple to describe and implement. Here is an example in pseudocode, assuming a zero-indexed array:

```
for i between 0 and (array length - 2):
    if (array(i + 1) < array(i)):
        switch array(i) and array(i + 1)

repeat until a complete iteration where no elements are switched.
```

Following is a simple example with a small list:

```
6, 4, 9, 5 -> 4, 6, 9, 5: When i = 0, the numbers 6 and 4 are switched
4, 6, 9, 5 -> 4, 6, 5, 9: When i = 2, the numbers 9 and 5 are switched
```


4, 6, 5, 9: The first iteration switched numbers, so iterate again

4, 6, 5, 9 -> 4, 5, 6, 9: When $i = 1$, the numbers 6 and 5 are switched

4, 5, 6, 9: The second iteration switched numbers, so iterate again

4, 5, 6, 9: No more numbers to switch, so this is the sorted list.

Listing 4-6 shows an implementation of the bubble sort in Java.

LISTING 4-6: A bubble sort implementation

```
public void bubbleSort(int[] numbers) {
    boolean numbersSwitched;
    do {
        numbersSwitched = false;
        for (int i = 0; i < numbers.length - 1; i++) {
            if (numbers[i + 1] < numbers[i]) {
                int tmp = numbers[i + 1];
                numbers[i + 1] = numbers[i];
                numbers[i] = tmp;
                numbersSwitched = true;
            }
        }
    } while (numbersSwitched);
}
```

Although this implementation is simple, it is extremely inefficient. The worst case, when you want to sort a list that is already sorted in reverse order, is a performance of $O(n^2)$: For each iteration, you are only switching one element. The best case is when a list is already sorted: You make one pass through the list, and because you have not switched any elements, you can stop. This has a performance of $O(n)$.

How would you implement the insert sort algorithm?

The insert sort algorithm is another simple algorithm to describe:

```
Given a list l, and a new list nl
for each element originallistelem in list l:
    for each element newlistelem in list nl:
        if (originallistelem < newlistelem):
            insert originallistelem in nl before newlistelem
        else move to the next element
    if originallistelem has not been inserted:
        insert at end of nl
```

Listing 4-7 shows an implementation.

LISTING 4-7: An insert sort implementation

```

public static List<Integer> insertSort(final List<Integer> numbers) {
    final List<Integer> sortedList = new LinkedList<>();

    originalList: for (Integer number : numbers) {
        for (int i = 0; i < sortedList.size(); i++) {
            if (number < sortedList.get(i)) {
                sortedList.add(i, number);
                continue originalList;
            }
        }
        sortedList.add(sortedList.size(), number);
    }

    return sortedList;
}

```

There are several points to note about this implementation. Notice that the method returns a new `List` unlike the bubble sort, which sorted the elements in place. This is mainly the choice of the implementation. Because this algorithm created a new `List`, it made sense to return that.

Also, the list implementation returned is a `LinkedList` instance. A linked list is very efficient in adding elements in the middle of the list, simply by rearranging the pointers of the nodes in the list. If an `ArrayList` had been used, adding elements to the middle would be expensive. An `ArrayList` is backed by an array, so inserting at the front or middle of the list means that all subsequent elements must be shifted along by one to a new slot in the array. This can be very expensive if you have a list with several million rows, especially if you are inserting early in the list. If the difference between list data structures, such as array lists and linked lists, are new to you, look at Chapter 5 on data structures and their implementations.

Finally, the outer loop in the implementation is labeled `originalList`. When an appropriate place to insert the element has been found, you can move on to the next element in the original array. Calling `continue` inside a loop will move on to the next iteration of the enclosing loop. If you want to continue in an outer loop, you need to label the loop and call `continue` with its identifier.

As hinted at with the outer loop labeling and the `continue` statement, once the element has successfully been placed, the algorithm moves on to the next element. This is an advantage over bubble sort: Once the list to return has been constructed, it can be returned immediately; there are no redundant cycles to check that the list is in order.

The worst-case performance for this algorithm is still $O(n^2)$, though: If you attempt to sort an already-sorted list, you need to iterate to the end of the new list with each element to insert. Conversely, if you sort a reverse-order list, you will be putting each element into the new list at the head of the list, which is $O(n)$.

The algorithm described here uses twice as much space to sort the list because a new list is returned. The bubble sort algorithm only uses one extra slot in memory, to hold the value temporarily while being swapped.

How would you implement the quicksort algorithm?

The description for the quicksort algorithm is:

```
method quicksort(list l):
    if l.size < 2:
        return l

    let pivot = l(0)
    let lower = new list
    let higher = new list
    for each element e in between l(0) and the end of the list:
        if e < pivot:
            add e to lower
        else add e to higher

    let sortedlower = quicksort(lower)
    let sortedhigher = quicksort(higher)
    return sortedlower + pivot + sortedhigher
```

This algorithm is recursive. The base case is when a list is provided with zero or one element. You can simply return that list because it is already sorted.

The second part of the algorithm is to pick an arbitrary element from the list, called the *pivot*. In this case the first element in the list was used, but any element could have been picked. The remaining elements are separated into two: those lower than the pivot, and those equal to or larger than the pivot.

The method is then called on each of the two smaller lists, which will return in each segment being sorted. The final list is the smaller elements, now sorted, followed by the pivot, and then the higher sorted elements.

Listing 4-8 is an implementation in Java.

LISTING 4-8: Quicksort

```
public static List<Integer> quicksort(List<Integer> numbers) {
    if (numbers.size() < 2) {
        return numbers;
    }

    final Integer pivot = numbers.get(0);
    final List<Integer> lower = new ArrayList<>();
    final List<Integer> higher = new ArrayList<>();

    for (int i = 1; i < numbers.size(); i++) {
        if (numbers.get(i) < pivot) {
            lower.add(numbers.get(i));
        } else {
            higher.add(numbers.get(i));
        }
    }

    return quicksort(lower) + pivot + quicksort(higher);
}
```

continues

LISTING 4-8 *(continued)*

```
        }  
    }  
  
    final List<Integer> sorted = quicksort(lower);  
  
    sorted.add(pivot);  
    sorted.addAll(quicksort(higher));  
  
    return sorted;  
}
```

If you ever write a recursive algorithm, you must make sure that it terminates. The algorithm in Listing 4-8 is guaranteed to terminate, because each recursive call is made with a smaller list, and the base case is on a zero- or one-element list.

The performance of Listing 4-8 is much more efficient than the bubble sort and insertion sort algorithms. The separation of the elements into two separate lists is $O(n)$, and each recursive call happens on half of each list, resulting in $O(n \log n)$. This is an average performance. The worst case is still $O(n^2)$. The choice of the pivot can make a difference: For the implementation given here, if you always pick the first element as the pivot, and the list is ordered in reverse, then the recursive steps only reduce by one each time.

It is worth noting that each division of the list and the subsequent recursive call is independent of any other sorting necessary, and could be performed in parallel.

How would you implement the merge sort algorithm?

The final sorting algorithm to examine in this chapter is the merge sort algorithm. The following pseudocode describes this recursive algorithm:

```
method mergesort(list l):  
    if list.size < 2:  
        return l  
  
    let middleIndex = l.size / 2  
    let leftList = elements between l(0) and l(middleIndex - 1)  
    let rightList = elements between l(middleIndex) and l(size - 1)  
  
    let sortedLeft = mergesort(leftList)  
    let sortedRight = mergesort(rightList)  
  
    return merge(sortedLeft, sortedRight)  
  
method merge(list l, list r):  
    let leftPtr = 0  
    let rightPtr = 0  
    let toReturn = new list  
  
    while (leftPtr < l.size and rightPtr < r.size):
```

```

        if(l(leftPtr) < r(rightPtr)):
            toReturn.add(l(leftPtr))
            leftPtr++
        else:
            toReturn.add(r(rightPtr))
            rightPtr++

    while(leftPtr < l.size):
        toReturn.add(l(leftPtr))
        leftPtr++

    while(rightPtr < r.size):
        toReturn.add(r(rightPtr))
        rightPtr++

    return toReturn

```

This is another divide-and-conquer algorithm: Split the list into two, sort each sublist, and then merge the two lists together.

The main code is in merging the two lists efficiently. The pseudocode holds a pointer to each sublist, adds the lowest value pointed to, and then increments the appropriate pointer. Once one pointer reaches the end of its list, you can add all of the remaining elements from the other list. From the second and third `while` statements in the preceding `merge` method, one `while` statement will immediately return `false`, because all the elements will have been consumed from that sublist in the original `while` statement.

Listing 4-9 shows an implementation.

LISTING 4-9: A merge sort implementation

```

public static List<Integer> mergesort(final List<Integer> values) {
    if (values.size() < 2) {
        return values;
    }

    final List<Integer> leftHalf =
        values.subList(0, values.size() / 2);
    final List<Integer> rightHalf =
        values.subList(values.size() / 2, values.size());

    return merge(mergesort(leftHalf), mergesort(rightHalf));
}

private static List<Integer> merge(final List<Integer> left,
                                   final List<Integer> right) {
    int leftPtr = 0;
    int rightPtr = 0;

    final List<Integer> merged =

```

continues

LISTING 4-9 (continued)

```

        new ArrayList<>(left.size() + right.size());

    while (leftPtr < left.size() && rightPtr < right.size()) {
        if (left.get(leftPtr) < right.get(rightPtr)) {
            merged.add(left.get(leftPtr));
            leftPtr++;
        } else {
            merged.add(right.get(rightPtr));
            rightPtr++;
        }
    }

    while (leftPtr < left.size()) {
        merged.add(left.get(leftPtr));
        leftPtr++;
    }

    while (rightPtr < right.size()) {
        merged.add(right.get(rightPtr));
        rightPtr++;
    }

    return merged;
}

```

There should be no surprise here; it is very similar to the pseudocode. Note that the `subList` method on `List` takes two parameters, `from` and `to`: `from` is *inclusive*, and `to` is *exclusive*.

A common question in interviews is often to ask you to merge two sorted lists, as was shown in the `merge` method in the preceding listing.

Again, merge sort has a performance of $O(n \log n)$. Each merge operation is $O(n)$, and each recursive call works on only half of the given list.

SEARCHING LISTS

How would you implement a binary search?

When searching through a list, unless the list is sorted in some fashion, the only sure way to find a given value is to look at every value in the list.

But if you are given a sorted list, or if you sort the list before searching, a binary search is a very efficient method to see if a given value is in the list:

```

method binarySearch(list l, element e):
    if l is empty:
        return false

    let value = l(l.size / 2)

```

```

    if (value == e):
        return true

    if (e < value):
        return binarySearch(elements between l(0) and l(l.size / 2 - 1))
    else:
        return binarySearch(elements between l(l.size / 2 + 1) and l(l.size))

```

The beauty of this algorithm is that you use the property of the sorted list to your advantage. You can throw away many elements without even examining them, because you know they definitely cannot be equal to the given element. If you have a list with one million elements, you can find a given element in as little as twenty comparisons. This algorithm has the performance of $O(n)$.

Listing 4-10 shows an implementation of a binary search.

LISTING 4-10: Binary search

```

public static boolean binarySearch(final List<Integer> numbers,
                                   final Integer value) {
    if (numbers == null || _numbers.isEmpty()) {
        return false;
    }

    final Integer comparison = numbers.get(numbers.size() / 2);
    if (value.equals(comparison)) {
        return true;
    }

    if (value < comparison) {
        return binarySearch(
            numbers.subList(0, numbers.size() / 2),
            value);
    } else {
        return binarySearch(
            numbers.subList(numbers.size() / 2 + 1, numbers.size()),
            value);
    }
}

```

SUMMARY

For many interviews, the core of the assessment is often around implementing algorithms and understanding the performance. Make sure you understand how the core sorting and searching algorithms work and how they perform, because this will prepare you for any complex algorithms you are asked to perform during an interview.

Some of the algorithms shown here are recursive. Make sure you understand the implications of a recursive algorithm: They can look elegant, but can have surprising side effects due to the

practicalities of calling a new method. The call stack has new values added, and if your call stack is too deep, you run the risk of a `StackOverflowException`.

Outside of an interview situation, whenever you need to sort a collection, use a library implementation. A tried and tested algorithm will almost always be more efficient than your own ad-hoc implementation. Any performance and memory concerns will have been examined. In fact, some of the sorting algorithms inside the Java standard libraries are implemented differently depending on the size of the list: Small lists are sorted in place using insert sort, but after a defined threshold, a merge sort is used.

The next chapter complements this chapter by providing an examination of some basic data structures, including lists, maps, and sets.

5

Data Structures

Every application uses data structures—lists of usernames, maps of query parameters, and so on. They are unavoidable, and are a core part of all of the Java APIs and most third-party APIs, as well.

This chapter examines the four most commonly used data structures: lists, trees, maps, and sets. You learn how they are implemented in Java, and how to use them depending on how you intend to read and write your data. If you only ever write to the front element of your list, you will find your application will perform much better with one list implementation over another.

Throughout the book, and for any serious development in Java, you will use the implementations discussed here. In any interview, you will be expected to understand the Java Collections API data structures, so make sure you fully understand the classes and interfaces presented in this chapter.

LISTS

Lists are sequential, ordered collections of values of a certain type. In Java, you will generally work with either `LinkedLists` or `ArrayLists`.

Lists differ from Java's built-in primitive collection type of arrays in that they are unbounded, so you do not need to specify the size of the array before using it.

At times it is more appropriate to use an `ArrayList` than a `LinkedList`, and vice versa. The use case for which of the lists to use will vary, but make sure you think it through, because it can have serious implications on your application's performance or its memory usage.

Whenever you are working with a list, you should always work to the `List` interface where possible. Method and constructor parameters should be the `List` interface, as should field definitions. This will make it much easier to swap implementing types depending on the scenario—although it may make sense to use an `ArrayList` in production code, you may be able to use an alternative implementation for tests. For example, seeding tests with dummy code from the `asList` method on the `Arrays` utility class in the standard Java library returns a list, compliant to the `List` interface, from the given parameters.

The Relationship between Arrays and Lists

Before investigating how different implementations of lists work, it is worth making sure you understand exactly how arrays work in Java. An array is defined as a type suffixed with square brackets. Listing 5-1 shows some array types and their definitions.

LISTING 5-1: Defining arrays

```
@Test
public void arrayDefinitions() {
    final int[] integers = new int[3];
    final boolean[] bools = {false, true, false};
    final String[] strings = new String[]{"one", "two"};

    final Random r = new Random();
    final String[] randomArrayLength = new String[r.nextInt(100)];
}
```

When defining an array, you must provide its size, either explicitly as a count, such as the `int` array in Listing 5-1, or implied, such as the `boolean` array. The compiler can calculate the length of that array for you.

You can use a computed value, such as the second `String` array. The JVM must know the size of the array when it is constructed.

You can access the element of an array directly by its index value. This is known as *random access*.

If you fill an array, and want to continue adding elements, you need to make the array larger. In effect, you need to create a new, larger array, copy all of the current elements into that array, and reassign the new array to the original reference. The JVM provides help with this operation, enabling you to copy the elements in bulk. The static `arrayCopy` method on the `System` object enables you to copy all or part of the array to a new array. Listing 5-2 shows how to use this method to extend the length of an array by one.

LISTING 5-2: Extending the size of an array

```
@Test
public void arrayCopy() {
    int[] integers = {0, 1, 2, 3, 4};

    int[] newIntegersArray = new int[integers.length + 1];
    System.arraycopy(integers, 0, newIntegersArray, 0, integers.length);
    integers = newIntegersArray;
    integers[5] = 5;

    assertEquals(5, integers[5]);
}
```

You cannot use the `final` modifier on the `integers` array, because you need to reassign the value.

One alternative to using arrays is to use the `List` interface. The `ArrayList` class is an implementation of the `List` interface that uses an array internally to store the data as a list representation.

Considering the interface is backed by an array, the class behaves similarly to an array. Directly accessing a specific element by its index is quick, because the array access is direct to a memory location.

When you construct an `ArrayList`, you have the option to specify the initial size of the underlying array. If you specify no value, the initial array size is ten. Whenever you attempt to add an element, and the underlying array is full, the `ArrayList` class will automatically reallocate your list into a larger array before continuing. This reallocation takes time and can use a large amount of memory, so if you know you are going to have a large collection when you construct the `ArrayList`, it makes sense to use a large number in the constructor. This will avoid making many costly array reallocations as your list grows.

If you want to add new elements at the beginning or middle of an `ArrayList`, all subsequent elements will need to shuffle along by one element to make room, which can be quite an expensive task for large arrays, especially if you are adding values closer to the start. This will be even more expensive if your new element requires the underlying array to be reallocated to a larger size.

Be aware that the array size reallocation is one-way; the array does not shrink if you remove many elements. If you have a list that oscillates between many and a few elements, an `ArrayList` might not be the best implementation for you; due to its memory requirements, a `LinkedList` might be more appropriate.

The `LinkedList` is the other main implementation of the `List`. Rather than having an array store the list elements, you use an internal object, which points to another object of the same type for the next element in the list. Listing 5-3 shows a simplified version of how to set up a `LinkedList`.

LISTING 5-3: A sample `LinkedList`

```
public class SimpleLinkedList<E> {
    private static class Element<E> {
        E value;
        Element<E> next;
    }

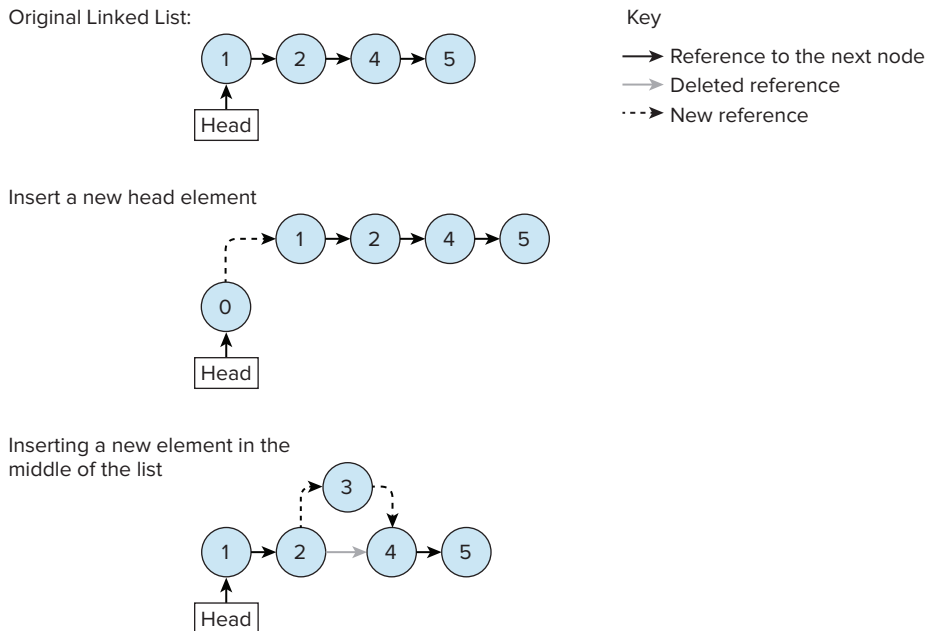
    private Element<E> head;
}
```

The `LinkedList` instance contains a reference to the head of the list, represented as an `Element`. The `Element` inner class is a *recursive data type*, with the `next` field pointing to the next element in the list. This enables you to traverse the list easily, visiting and processing each element in order.

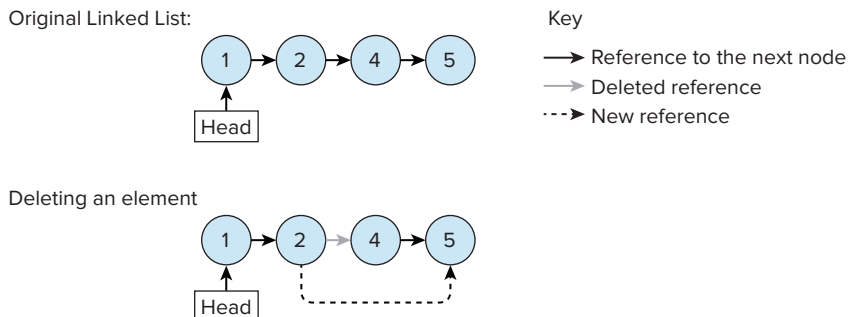
Like all of the differences between implementations, some trade-offs exist. If you want to get an element by its index, you need to traverse the length of the list, keeping count until you reach the given index. For an `ArrayList`, which has random access, using `get` is instant.

You may find that some linked list implementations also contain references to the previous element. This allows for doubly linked lists, with traversal in either direction, and it allows for easy list reversal.

Linked lists enable you to insert objects at the head or in the middle of the list without shuffling all subsequent elements along, as would be required with an `ArrayList`. Figure 5-1 shows inserting an element at the head, and also in the middle, of the list.

**FIGURE 5-1**

Deleting elements is also simple: The deleted element's next becomes the previous element's next. Figure 5-2 shows deleting an element in the middle of the list.

**FIGURE 5-2**

It is quite clear that you have powerful reasons to use both `LinkedLists` and `ArrayLists`. As a rule of thumb, you should use an `ArrayList` if you need random access to elements in the list, especially if your use case could result in large lists.

If you intend to make many insertions and deletions particularly if you make them at the beginning or in the middle of the list, then a `LinkedList` would make more sense. `LinkedLists` also do not suffer from the expensive array reallocation process that occurs with `ArrayLists`, and as your list shrinks, so does your memory consumption. If you were to create your own specialized data structure, such as a stack, using a `LinkedList` as the underlying data structure would make sense, because you would simply push and pop elements at the head of the list.

What is a Queue and Deque?

A `Queue` is a Java interface that represents a “first in, first out” data structure. The interface has the methods `add`, to add a new element, `remove` to remove the oldest element, and `peek`, which returns the oldest element, but does not remove it from the data structure. Listing 5-4 shows an example. The `LinkedList` class implements the `Queue` interface.

LISTING 5-4: Using a queue

```
@Test
public void queueInsertion() {
    final Queue<String> queue = new LinkedList<>();
    queue.add("first");
    queue.add("second");
    queue.add("third");

    assertEquals("first", queue.remove());
    assertEquals("second", queue.remove());
    assertEquals("third", queue.peek());
    assertEquals("third", queue.remove());
}
```

A `Deque` (pronounced “deck”) is an extension of `Queue`, and allows addition and removal from either end of the data structure.

TREES

A tree is a data structure in which an element can be succeeded by a number of different elements, known as *children*. A common use of the tree data structure is a *binary tree*, in which each element has a maximum of two children. Figure 5-3 shows an example tree.

With binary trees, the two children are often differentiated as *left* and *right*, and depending on what exactly the tree structure is representing, different values will reside on the left or right of a given node.

One such implementation of a binary tree is a *binary search tree*, which Figure 5-3 shows. With a binary search tree, elements “less than” the value of a given node are children on the left, and elements “greater than” the value of a given node are children on the right.

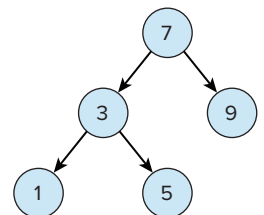


FIGURE 5-3

What exactly “less than” or “greater than” means depends on the use case of the actual tree, but for most instances, elements inserted into a tree will have a natural ordering, such as numbers or dates. Sometimes you may want to explicitly define how the ordering works; for instance, you may want to compare strings by length before comparing character values.

Listing 5-5 shows a sample implementation of a tree.

LISTING 5-5: A sample tree definition

```
public class SimpleTree<E extends Comparable> {  
    private E value;  
    private SimpleTree<E> left;  
    private SimpleTree<E> right;  
    ... // constructors, getters and setters omitted  
}
```

This tree is another recursive data type, with the children being `SimpleTrees` also. This tree definition is able to store any `Comparable` elements. A more complex example may be able to use a `Comparator` to compare any kind of object.

When searching for a given element, you have three scenarios to consider for a given node. These are shown in Listing 5-6.

LISTING 5-6: Finding values in a binary search tree

```
public boolean search(final E toFind) {  
    if (toFind.equals(value)) {  
        return true;  
    }  
  
    if (toFind.compareTo(value) < 0 && left != null) {  
        return left.search(toFind);  
    }  
  
    return right != null && right.search(toFind);  
}
```

If the value to find is equal, you have a match. If the value to find is less than the current node, *and the left node is not null*, repeat the search on the left node. Otherwise, if the right node is not null, repeat the search on the right node. If a child node to search is null, you have reached the bottom of the tree, and the value to find is not in the tree.

Inserting values into a tree follows a similar structure. You follow the tree, comparing each node with the value to insert, going left if the value to insert is less than the value at the current node, or right otherwise. If the next node to inspect is null, this is where your new value is to be inserted.

Listing 5-7 shows how you can write this functionality.

LISTING 5-7: Inserting values into a binary tree

```

public void insert(final E toInsert) {
    if (toInsert.compareTo(value) < 0) {
        if (left == null) {
            left = new SimpleTree<>(toInsert, null, null);
        } else {
            left.insert(toInsert);
        }
    } else {
        if (right == null) {
            right = new SimpleTree<>(toInsert, null, null);
        } else {
            right.insert(toInsert);
        }
    }
}

```

Listing 5-8 shows a sample test, illustrating that the data is stored appropriately.

LISTING 5-8: Ensuring the properties of the binary tree are correct

```

@Test
public void createTree() {
    final SimpleTree<Integer> root = new SimpleTree<>(7, null, null);
    root.insert(3);
    root.insert(9);
    root.insert(10);
    assertTrue(root.search(10));
    assertEquals(Integer.valueOf(10),
        root.getRight().getRight().getValue());
}

```

This test confirms that the inserted value 10 is in the tree, and that it is stored in the expected position.

Using object orientation, it is possible to simplify the methods using the Null Object Pattern, while also removing all of the null checks. This models the tree as having two different types of nodes: ones that have values (nodes) and ones that represent the bottom of the tree (leaves). Listing 5-9 shows how you can set this up, and how you would write the search method.

LISTING 5-9: Building trees with null objects

```

public interface Tree<E extends Comparable> {
    boolean search(E toFind);
    void insert(E toInsert);
}

public class Node<E extends Comparable> implements Tree<E> {

```

continues

LISTING 5-9 (continued)

```

...
@Override
public boolean search(E toFind) {
    if (toFind.equals(value)) {
        return true;
    }
    if (toFind.compareTo(value) < 0) {
        return left.search(toFind);
    }

    return right.search(toFind);
}
...
}

public class Leaf<E extends Comparable> implements Tree<E> {
    ...
    @Override
    public boolean search(E toFind) {
        return false;
    }
    ...
}

```

If a search traverses to the bottom of the tree, a `Leaf` object will return `false`: This object represents the absence of a value in the tree. You can write the `insert` method in a similar way. You traverse down the tree until you hit a `Leaf`, and then replace that leaf with a new `Node` of that value, itself having two leaves as children.

The `insert` method is similar to `search`, with each `Node` checking if the value to be inserted is less than or greater than the current node, and recursively traversing in the appropriate direction. The `Leaf` then sets its parent to have a new `Node` instance with that new value.

One concern of this approach is that it is possible to produce an *unbalanced tree*. Listing 5-10 shows a sequence of insertions.

LISTING 5-10: A possible set of insertions into a binary tree

```

final SimpleTree<Integer> root = new SimpleTree<>(1, null, null);
root.insert(2);
root.insert(3);
root.insert(4);
root.insert(5);

```

This would produce a tree such as the one displayed in Figure 5-4.

This definitely respects the properties of a binary search tree, but it is not as efficient as it could be: It is essentially a linked list. A specific implementation of a binary search tree, called an *AVL Tree*,

enforces that, for any node, the difference in depth for each child is at most one. After each insertion or deletion of a node, the tree checks if it is still balanced, and subsequently rotates the nodes of values where the property of the AVL Tree does not hold. Figure 5-5 shows the steps taken to keep the tree in Listing 5-10 balanced.

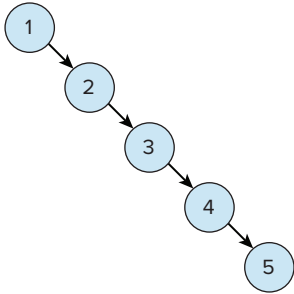


FIGURE 5-4

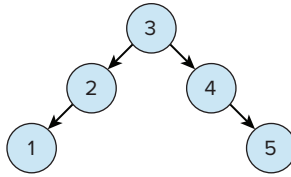


FIGURE 5-5

When a tree is balanced, searching, inserting, and deleting has the size $O(\log n)$.

Binary trees can be used for more than just searching. Another application of a binary tree is called a *Binary Heap*, which is a balanced tree with the property that children are “greater than” their parent, as demonstrated in Figure 5-6.

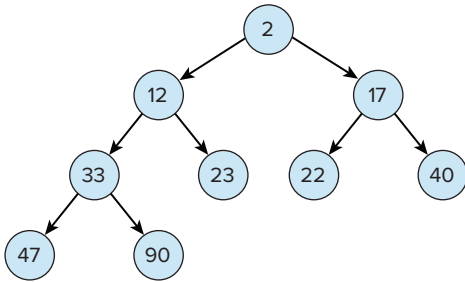


FIGURE 5-6

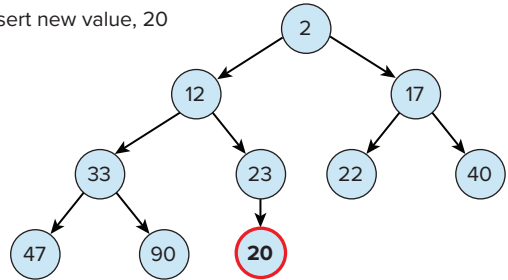
Again, the “greater than” comparison is on a per-implementation basis, either by a natural ordering, which would use a `Comparable` interface in Java, or a specified ordering, using a `Comparator` object.

The heap property defines that the smallest element in the tree is at the root. Heaps are especially useful for priority queues, or any time you require quick access to the smallest element of a collection. The remainder of the data structure is only partially sorted; that is, each node is smaller than its children, so it is not the most useful if you want the elements in order.

To insert elements into a heap, you start by adding the element at the next available position in the lowest level of the tree. You then compare the inserted value with its parent, and if the new value is

lower, the two values swap. The process continues, recursively, with the new value bubbling up until it is not necessary to swap any more elements. Figure 5-7 illustrates insertion.

Insert new value, 20



Switch with parent, heap property now holds

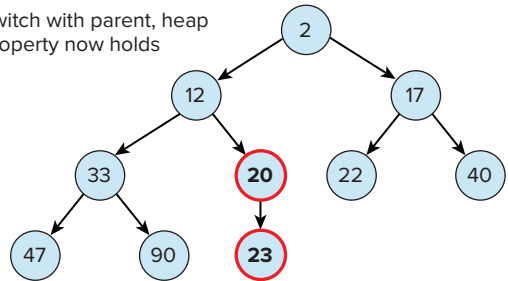


FIGURE 5-7

Deleting elements is a common use for heaps, especially when it is used as a priority queue. Deleting elements from the root or from within the tree requires the property that the tree remains balanced. When an element is removed, it is replaced with the last element in the heap. The value will be larger than its parent because it came from the bottom of the heap, but it may need to be swapped with one of its children to retain the heap property. The comparisons repeat until the value finds both children are larger, or the bottom row of the heap is found. Figure 5-8 illustrates deletion.

Although binary trees are the most common implementation of any kind of tree structure, you also can model trees with more than two children, often called *n-ary trees*. These trees often come up when defining probability outcomes, or scenarios in which more than two outcomes are possible at each choice. Figure 5-9 shows one such tree.

Note that the exact number of children for each parent can vary. Trees of this nature can still be modeled as a binary tree. The left child is the *first child* of the original parent tree, and the right child is the *next sibling*. Figure 5-10 shows the same tree defined in Figure 5-9, displayed as first-child, next-sibling.

Delete number 17

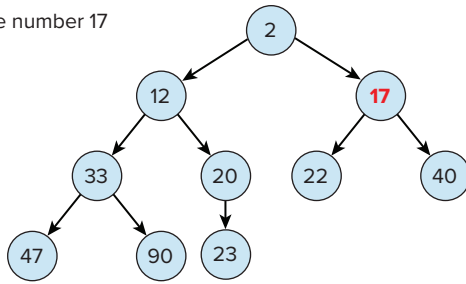
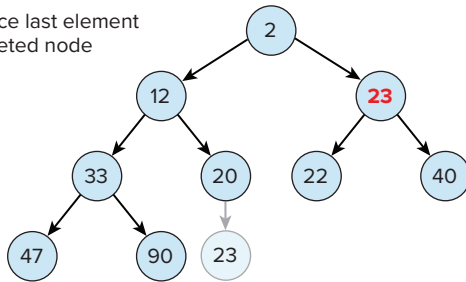
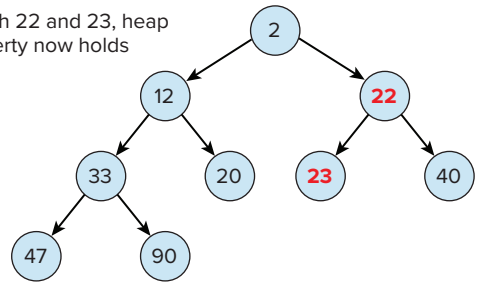
Replace last element
at deleted nodeSwitch 22 and 23, heap
property now holds

FIGURE 5-8

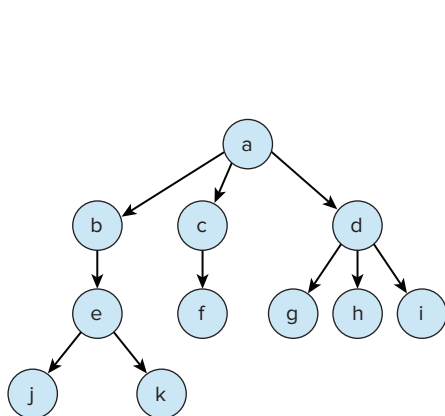


FIGURE 5-9

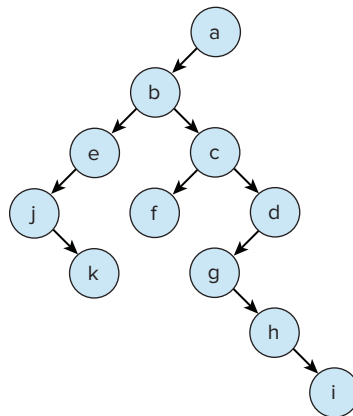


FIGURE 5-10

MAPS

A map, sometimes called a hash, associative array or dictionary, is a key-value store. Elements within the data structure can be queried by the key, which will return the associated value.

The Map interface is part of the Java Collections API, but, unlike `List`, it does not implement the `Collection` interface. Similar to the `List` interface, the Map interface specifies most common

operations for map implementations, such as the data structure size and the ability to read, insert, and delete key-value pairs.

One property of a map is that the value of a key can only appear once: Reinserting that key will result in the original value being overwritten, as shown in Listing 5-11.

LISTING 5-11: Overwriting keys in maps

```
@Test
public void overwriteKey() {
    final Map<String, String> preferences = new HashMap<>();
    preferences.put("like", "jacuzzi");
    preferences.put("dislike", "steam room");

    assertEquals("jacuzzi", preferences.get("like"));

    preferences.put("like", "sauna");

    assertEquals("sauna", preferences.get("like"));
}
```

Listing 5-11 displays use of a `HashMap`. This is the most common implementation of Java's `Map` interface, but other implementations exist, each with its own performance and storage implications, but still respecting the `Map` interface.

The `HashMap` class is Java's implementation of the hash table. The class implementation contains an inner class, called `Entry`, to represent the key-value pairs, and the elements are stored in an array of `Entry` objects. This is purely an implementation decision; it could have used a `List` of `Entry` objects instead.

The value of a particular key instance defines where the particular pair resides in the table. The `hashCode` method, defined on `Object`, returns an `int`, and that value is used to determine where exactly to place the key-value pair in the table. Chapter 8 defines the specific details for what exactly the `hashCode` value can return, but in a nutshell, two equal instances must return the same `hashCode` value. However, the inverse is not true: two equal `hashCode` values do not imply equality.

When you insert a value into the table for `HashMap`, the `hashCode` method is read, and because the value returned could be any valid value in the `int` range, it is then factored to a value between zero and one less than the size of the table; that is, a valid index for the table.

Considering unequal objects can return the same `hashCode`, unequal objects may be put into the same hash table entry, so handling collisions must be taken into account.

One approach would be to have a secondary hash function, and this second hash is calculated as an offset if you want to insert a value but an entry already exists in the given table position. This would just delay any collisions—they would still happen, and Java objects only have one `hashCode` method, so you should consider another approach.

One such alternative approach is to store a list of entries for each table element, and then all table keys that resolve to the same table index would be added to this value. Simply traversing the table would then perform key retrieval, checking equality on each element until a match is found.

If your class defines `hashCode` as a single value regardless of equality, then (although this meets the contract for `hashCode`), any `HashMap` with instances of this class will perform like a linked list, losing all benefits of a potential $O(1)$ lookup time. This is also true for classes with a small range of instance values such as `Boolean` or even `Short`.

Naturally, the size of the table will determine the frequency of possible collisions: A small table with many entries is more likely to have a collision on insertion than a large table with few entries.

When you construct a new `HashMap` object, you have the opportunity to specify a load factor: a value between 0 and 1 that represents a percentage. Once the underlying table has filled to this capacity, the table is doubled in size.

When a table is resized, all of the elements in the table need to be redistributed, because the `hashCode` value is factored to an index in the table. This also means that objects that collided in the smaller table do not collide any more. For a table containing a large number of elements, the recalculation of all key-value pair table entries can be expensive and time-consuming. If you know that you are going to have a large number of entries in a map, it may make sense to specify an initial size for the table when constructing the map to avoid resizing the table.

An alternative `Map` implementation is the `TreeMap`, which uses a binary tree data structure to conform to the `Map` interface. Each node in the tree is a key-value pair. Chapter 4 covered sorting, and explained the `Comparable` and `Comparator` interfaces. Those interfaces are used with a `TreeMap`. Keys can either be naturally ordered, in which case they must conform to the `Comparable` interface, or you can specify your own ordering with the `Comparator` interface.

Because the `TreeMap` stores the keys based on their natural ordering, the `hashCode` method is not used at all. Each element put into the `TreeMap` rebalances the tree, so that searching, deletion, and further insertion can always be performed as efficiently as possible: $O(\log n)$.

One main difference between a `TreeMap` and a `HashMap` is that, with a `TreeMap` the order of the keys is preserved when iterating over the whole collection, because the collection is stored in order. This isn't possible with a `HashMap`, because the keys are stored dependent on the object's `hashCode`. Listing 5-12 shows that the order is preserved when iterating on a `TreeMap`.

LISTING 5-12: Order-preserving iteration with a `TreeMap`

```
@Test
public void treeMapTraversal() {
    final Map<Integer, String> counts = new TreeMap<>();
    counts.put(4, "four");
    counts.put(1, "one");
    counts.put(3, "three");
    counts.put(2, "two");

    final Iterator<Integer> keys = counts.keySet().iterator();
    assertEquals(Integer.valueOf(1), keys.next());
    assertEquals(Integer.valueOf(2), keys.next());
    assertEquals(Integer.valueOf(3), keys.next());
    assertEquals(Integer.valueOf(4), keys.next());
    assertFalse(keys.hasNext());
}
```

A specialized `Map` implementation is the `LinkedHashMap`. This implementation works in the same way as a `HashMap`, so element retrieval will be $O(1)$, but it has the added property that iterating over the keys will be in the same order as insertion, as Listing 5-13 shows.

LISTING 5-13: A `LinkedHashMap`

```
@Test
public void linkedHashMapTraversal() {
    final Map<Integer, String> counts = new LinkedHashMap<>();
    counts.put(4, "four");
    counts.put(1, "one");
    counts.put(3, "three");
    counts.put(2, "two");

    final Iterator<Integer> keys = counts.keySet().iterator();
    assertEquals(Integer.valueOf(4), keys.next());
    assertEquals(Integer.valueOf(1), keys.next());
    assertEquals(Integer.valueOf(3), keys.next());
    assertEquals(Integer.valueOf(2), keys.next());
    assertFalse(keys.hasNext());
}
```

One final implementation to consider is the `ConcurrentHashMap`. You should use this implementation if you ever want to share the map instance with many threads. It is thread safe, and has been specifically designed to be able to return read values while the values are being written to the map. Only the specified row of the table is locked during a write, leaving the rest of the map available for reading.

The contract for `ConcurrentHashMap` has been slightly tweaked from the original `Map`: The `size` method returns an estimation of the map size, because it does not account for any writes currently being applied.

SETS

A set is an unordered collection of objects that does not contain any duplicates.

The Java Collections API contains a `Set` interface, extending from `Collection`, which provides many methods for inspecting and modifying a set. Listing 5-14 shows a small unit test that demonstrates the properties of a set.

LISTING 5-14: Using a set

```
@Test
public void setExample() {
    final Set<String> set = new HashSet<>();
```

```
        set.add("hello");
        set.add("welcome");
        set.add("goodbye");
        set.add("bye");
        set.add("hello");

        assertEquals(4, set.size());
    }
```

The string "hello" was added into the set twice, but because sets do not contain duplicates, only one instance is stored in the set.

The implementation used in Listing 5-14 is a `HashSet`. This implementation uses an underlying `HashMap`, storing the value as the key to the map, and the value is a marker object, signifying that a value is stored.

For each of the `Map` implementations visited earlier, there is an equivalent `Set` implementation—`HashSet`, `TreeSet`, and `LinkedHashSet`—although there is no `Set` backed by a `ConcurrentHashMap`. There is, however, a static method on the `Collections` class, `newSetFromMap`, which takes a `Map` and returns a `Set`, respecting the properties of the given map.

SUMMARY

Be prepared for questions on data structures. An interviewer will want to know what you understand about the different implementations and what implications your choice of a particular data structure will have on the rest of your application. You should also be able to explain *when* it is appropriate to use different data structures.

Try the examples in this chapter, and experiment with some of your own, too: Try to work out where any performance issues arise, what it takes to make your JVM crash with an `OutOfMemoryError`, or something similar.

Explore the Java Collections API. Take a look at the `Collections` class: It is a set of static methods that works to make your life a little easier in creating and manipulating standard Java data structures. The equivalent for arrays, the `Arrays` class, provides some interoperability between collections and arrays.

Chapter 18 covers some third party libraries, and introduces some new collection types, such as sets allowing duplicate elements (`MultiSet`) and maps allowing duplicate keys (`MultiMap`).

The next chapter covers object-oriented design patterns, and how to implement and notice some commonly used implementations.

6

Design Patterns

Software design patterns use one or more objects together to solve a common use case. These use cases often focus on code reuse, extensibility, or generally providing a solid foundation for future development.

Being an object-oriented language, Java makes use of many design patterns throughout the standard library APIs, and it is quite simple to create your own implementation of some common design patterns.

This chapter looks at several common design patterns, examining some of those found in the standard Java APIs, as well as implementing some patterns too.

Hundreds, if not thousands, of software design patterns exist, and you may create some new ones that work for you in your everyday job. The patterns described here are some of the more common and well-known patterns. Interviewers often like asking questions about design patterns, because they provide scope for a lot of discussion, and they have very clear definitions.

Try the examples in this chapter, think of your own scenarios, and see if you can apply the patterns to those, too. Try to justify if a given pattern is good for a given situation, and see if you can make other patterns fit that same situation, too. It won't always work, but sometimes it is possible for patterns to be interchanged in certain situations.

INVESTIGATING EXAMPLE PATTERNS

How is the Builder Pattern useful?

When you want to create an object that has many fields, using constructors can become unwieldy and confusing. Imagine the following class definition, which could possibly be used for a veterinarian's database:

```
public class Pet {  
    private final Animal animal;
```

```
private final String petName;  
private final String ownerName;  
private final String address;  
private final String telephone;  
private final Date dateOfBirth; // optional  
private final String emailAddress; //optional  
...
```

To allow for any combination of these fields to create a valid `Pet` object, you need at least four constructors with many parameters containing all of the required fields, and all combinations of the optional fields. If you were to add more optional fields, this would soon become hard to manage or even understand.

One approach would be to not use a constructor at all, remove the `final` modifier from the fields, and simply use setter methods on the `Pet` object. The main downside with this is that you can create objects that are not valid `Pet` objects:

```
final Pet p = new Pet();  
p.setEmailAddress("owners@address.com");
```

The definition of a `Pet` object is that the animal, `petName`, `ownerName`, `address`, and `telephone` fields are all set.

Using the Builder Pattern can mitigate this. With the Builder Pattern, you create a companion object, called a *builder*, which will construct domain-legal objects. It is also clearer than simply using constructors. Listing 6-1 shows the use of a builder to create `Pet` objects.

LISTING 6-1: Constructing objects with the Builder Pattern

```
@Test  
public void legalBuild() {  
    final Pet.Builder builder = new Pet.Builder();  
    final Pet pet = builder  
        .withAnimal(Animal.CAT)  
        .withPetName("Squidge")  
        .withOwnerName("Simon Smith")  
        .withAddress("123 High Street")  
        .withTelephone("07777777770")  
        .withEmailAddress("simon@email.com")  
        .build();  
    // test pass - no exception thrown  
}  
  
@Test(expected = IllegalStateException.class)  
public void illegalBuild() {  
    final Pet.Builder builder = new Pet.Builder();  
    final Pet pet = builder  
        .withAnimal(Animal.DOG)  
        .withPetName("Fido")  
        .withOwnerName("Simon Smith")  
        .build();  
}
```

The `Builder` class is part of the `Pet` class, and has the sole responsibility of creating `Pet` objects. With a constructor, the parameter ordering determines how each parameter is used. By having an explicit method for each parameter, you can understand exactly what each value is used for, and you can call them in any order you like. The `build` method calls the actual constructor of the `Pet` object and returns an actual `Pet` object. The constructor is now private.

Notice, too, that the convention with calls to builder methods is to *chain* them (`builder.withXXX().withYYY()...`), which allows for a more concise definition. Listing 6-2 shows how the `Builder` class for the `Pet` class is implemented.

LISTING 6-2: Implementing the Builder Pattern

```
public class Pet {
    public static class Builder {
        private Animal animal;
        private String petName;
        private String ownerName;
        private String address;
        private String telephone;
        private Date dateOfBirth;
        private String emailAddress;

        public Builder withAnimal(final Animal animal) {
            this.animal = animal;
            return this;
        }
        public Builder withPetName(final String petName) {
            this.petName = petName;
            return this;
        }
        public Builder withOwnerName(final String ownerName) {
            this.ownerName = ownerName;
            return this;
        }
        public Builder withAddress(final String address) {
            this.address = address;
            return this;
        }
        public Builder withTelephone(final String telephone) {
            this.telephone = telephone;
            return this;
        }
        public Builder withDateOfBirth(final Date dateOfBirth) {
            this.dateOfBirth = dateOfBirth;
            return this;
        }
        public Builder withEmailAddress(final String emailAddress) {
            this.emailAddress = emailAddress;
            return this;
        }
    }
    public Pet build() {
```

continues

LISTING 6-2 *(continued)*

```
        if (animal == null ||
            petName == null ||
            ownerName == null ||
            address == null ||
            telephone == null) {
            throw new IllegalStateException("Cannot create Pet");
        }

        return new Pet(
            animal,
            petName,
            ownerName,
            address,
            telephone,
            dateOfBirth,
            emailAddress);
    }

    private final Animal animal;
    private final String petName;
    private final String ownerName;
    private final String address;
    private final String telephone;
    private final Date dateOfBirth; // optional
    private final String emailAddress; //optional

    private Pet(final Animal animal,
                final String petName,
                final String ownerName,
                final String address,
                final String telephone,
                final Date dateOfBirth,
                final String emailAddress) {
        this.animal = animal;
        this.petName = petName;
        this.ownerName = ownerName;
        this.address = address;
        this.telephone = telephone;
        this.dateOfBirth = dateOfBirth;
        this.emailAddress = emailAddress;
    }
}
```

If your domain objects have default values, you can have these already set in the builder fields, and then you can override them where necessary. This allows for even more concise construction of objects. Listing 6-3 shows another builder, but with default values.

LISTING 6-3: Using default values in a builder

```

public class LibraryBook {
    public static class Builder {
        private BookType bookType = BookType.FICTION;
        private String bookName;

        public Builder withBookType(final BookType bookType) {
            this.bookType = bookType;
            return this;
        }

        public Builder withBookName(final String bookName) {
            this.bookName = bookName;
            return this;
        }

        public LibraryBook build() {
            return new LibraryBook(bookType, bookName);
        }
    }

    private final BookType bookType;
    private final String bookName;

    public LibraryBook(final BookType bookType, final String bookName) {
        this.bookType = bookType;
        this.bookName = bookName;
    }
}

```

You can then easily create fiction books without needing to specify the type:

```

@Test
public void fictionLibraryBook() {
    final LibraryBook.Builder builder = new LibraryBook.Builder();
    final LibraryBook book = builder
        .withBookName("War and Peace")
        .build();

    assertEquals(BookType.FICTION, book.getBookType());
}

```

Can you give an example of the Strategy Pattern?

The Strategy Pattern enables you to easily swap specific implementation details of an algorithm without requiring a complete rewrite. You can even swap implementations at run time. The Strategy Pattern is often used in conjunction with dependency injection to allow implementations to be swapped out for test-specific code, or to allow mocked implementations to be used instead.

Listing 6-4 shows how you could write a simple logger using the Strategy Pattern.

LISTING 6-4: Implementing a logger with the Strategy Pattern

```
public interface Logging {
    void write(String message);
}

public class ConsoleLogging implements Logging {
    @Override
    public void write(final String message) {
        System.out.println(message);
    }
}

public class FileLogging implements Logging {

    private final File toWrite;

    public FileLogging(final File toWrite) {
        this.toWrite = toWrite;
    }

    @Override
    public void write(final String message) {
        try {
            final FileWriter fos = new FileWriter(toWrite);
            fos.write(message);
            fos.close();
        } catch (IOException e) {
            // handle IOException
        }
    }
}
```

This enables you to write code using the `Logging` interface, and the code you want to log does not need to be concerned if the logging is being written to the console output or to a file. By programming to the interface rather than to a specific implementation, you can use the `ConsoleLogging` strategy when testing your code, and use `FileLogging` for production use. At a later date, you can add more implementations of `Logging`, and any code written against the interface will not change but will be able to use the new implementation.

Listing 6-5 shows how you can use the `Logging` interface in client code, along with three examples of tests: two using the `ConsoleLogging` and `FileLogging` strategies, and one using a Mockito mock. If you are not familiar with mocking, Chapter 9 shows how it works.

LISTING 6-5: Using the Logging strategy

```
public class Client {

    private final Logging logging;

    public Client(Logging logging) {
```

```

        this.logging = logging;
    }

    public void doWork(final int count) {
        if (count % 2 == 0) {
            logging.write("Even number: " + count);
        }
    }
}

public class ClientTest {

    @Test
    public void useConsoleLogging() {
        final Client c = new Client(new ConsoleLogging());
        c.doWork(32);
    }

    @Test
    public void useFileLogging() throws IOException {
        final File tempFile = File.createTempFile("test", "log");
        final Client c = new Client(new FileLogging(tempFile));
        c.doWork(41);
        c.doWork(42);
        c.doWork(43);

        final BufferedReader reader
            = new BufferedReader(new FileReader(tempFile));
        assertEquals("Even number: 42", reader.readLine());
        assertEquals(-1, reader.read());
    }

    @Test
    public void useMockLogging() {
        final Logging mockLogging = mock(Logging.class);
        final Client c = new Client(mockLogging);
        c.doWork(1);
        c.doWork(2);
        verify(mockLogging).write("Even number: 2");
    }
}

```

LOGGING IN JAVA APPLICATIONS

The code used in Listings 6-3 and 6-4 is purely for the example of how to use the Strategy Pattern; you should never implement logging this way, or ever write your own logger. Java has many libraries dedicated to logging output, such as Log4J and SLF4J. Look at these libraries and see if you can understand how the Strategy Pattern has been implemented, and how you can specify different behaviors.

Using the Strategy Pattern enables you to defer decisions about which implementation to use until run time. The Spring Framework uses an XML file to construct objects and their dependencies, which are read at run time, allowing a quick change between implementations without any need for recompilation. Chapter 16 has more information.

How can you use the Template Pattern?

The Template Pattern is used to defer or delegate some or all steps of an algorithm to a subclass. Common behavior can be defined in a superclass, and then specific variants are written in a subclass. Listing 6-6 shows a possible definition for a stack, backed by a `LinkedList` and an interface for filtering a `Stack` instance, called `StackPredicate`.

LISTING 6-6: A stack

```
public class Stack {

    private final LinkedList<Integer> stack;

    public Stack() {
        stack = new LinkedList<>();
    }

    public Stack(final LinkedList<Integer> initialState) {
        this.stack = initialState;
    }

    public void push(final int number) {
        stack.add(0, number);
    }

    public Integer pop() {
        return stack.remove(0);
    }

    public Stack filter(final StackPredicate filter) {
        final LinkedList<Integer> initialState = new LinkedList<>();
        for (Integer integer : stack) {
            if (filter.isValid(integer)) {
                initialState.add(integer);
            }
        }
        return new Stack(initialState);
    }
}

public interface StackPredicate {
    boolean isValid(int i);
}
```


The `StackPredicate` is an implementation of the Template Pattern that allows client code to define exactly how the filtering should work.

The `filter` method has specific logic: It iterates through each value in the stack, and asks the predicate if that value is to be included in the filtered stack to return. The logic in the `filter` method is completely isolated from the specific logic of the `StackPredicate` in deciding whether a given value is valid to be filtered.

Listing 6-7 shows two possible implementations of a `StackPredicate`: one that only allows even numbers, and one that has no effect:

LISTING 6-7: Implementing the `StackPredicate` template

```
public class StackPredicateTest {

    private Stack stack;

    @Before
    public void createStack() {
        stack = new Stack();
        for (int i = 1; i <= 10; i++) {
            stack.push(i);
        }
    }

    @Test
    public void evenPredicate() {
        final Stack filtered = stack.filter(new StackPredicate() {
            @Override
            public boolean isValid(int i) {
                return (i % 2 == 0);
            }
        });

        assertEquals(Integer.valueOf(10), filtered.pop());
        assertEquals(Integer.valueOf(8), filtered.pop());
        assertEquals(Integer.valueOf(6), filtered.pop());
    }

    @Test
    public void allPredicate() {
        final Stack filtered = stack.filter(new StackPredicate() {
            @Override
            public boolean isValid(int i) {
                return true;
            }
        });

        assertEquals(Integer.valueOf(10), filtered.pop());
        assertEquals(Integer.valueOf(9), filtered.pop());
        assertEquals(Integer.valueOf(8), filtered.pop());
    }
}
```

Separating the `StackPredicate` has numerous advantages. It allows clients of the `Stack` class to filter it to their specific needs, requiring no changes to the `Stack` class at all. Also, the `StackPredicate` implementations are single units of code, and therefore can be thoroughly tested in isolation without needing a `Stack` instance.

COMMONLY USED PATTERNS

Can you describe an implementation of the Decorator Pattern?

The Decorator Pattern enables you to change or configure the functionality of a specific object, such as adding buttons or functionality to a scrollbar, or defining exactly how to model sandwich orders from two different customers from the same set of ingredients.

Java's original IO classes for reading and writing to sources outside of the JVM use the Decorator Pattern extensively. The `InputStream` and `OutputStream` classes and their subclasses use the Decorator Pattern to read and write data in ways defined by the implementing classes, and these implementations can often be chained together to provide effective ways of reading sources and writing back.

Examining the abstract `OutputStream` class, you see it has one method that implementations must define, which is simply how to write a byte:

```
public abstract void write(int b) throws IOException;
```

You can use a couple of other methods to write bytes in bulk, which by default call the preceding method for each byte. Depending on the implementation, it may be more efficient to override these methods.

Many of the `OutputStream` implementations perform their necessary action, and then delegate to another `OutputStream`. These implementations will take another `OutputStream` as a constructor argument.

The `OutputStreams` that actually perform the writing of data, such as a `FileOutputStream` or a `SocketOutputStream`, will not delegate their `write` method to another stream, and thus will not need another `OutputStream` object in their constructor arguments.

Listing 6-8 shows the Decorator Pattern in use, chaining several operations on an array of bytes before writing them to disk.

LISTING 6-8: Using the Decorator Pattern to write to disk

```
@Test
public void decoratorPattern() throws IOException {
    final File f = new File("target", "out.bin");
    final FileOutputStream fos = new FileOutputStream(f);
    final BufferedOutputStream bos = new BufferedOutputStream(fos);
```

```

        final ObjectOutputStream oos = new ObjectOutputStream(fos);

        oos.writeBoolean(true);
        oos.writeInt(42);
        oos.writeObject(new ArrayList<Integer>());

        oos.flush();
        oos.close();
        bos.close();
        fos.close();

        assertTrue(f.exists());
    }

```

The first five lines of this test define the file to write to and how it will be written. The `FileOutputStream` writes files to disk. A `BufferedOutputStream` will cache any calls to write, and will write several bytes at once. This can be a huge efficiency gain when writing to disk. The `ObjectOutputStream` is Java's built-in serialization mechanism that writes objects and primitive types to a stream. Serialization is covered in more detail in Chapter 15.

Notice here that the `ObjectOutputStream` does not know where the files are being written; it is simply delegating to another `OutputStream`. The power of this pattern is that you can provide new `OutputStream` implementations, and these will work with those that already exist.

If, say, you wanted to write a log of every call to a `BufferedOutputStream`, you could write an implementation easily enough, and it would work with the chain defined earlier. Or if you wanted to encrypt data before writing, you could do that as well.

If you wanted to compress the data before writing to disk, you could use the `GZIPOutputStream` class provided by the Java standard library.

Remember that the order in which you define the chain of `OutputStreams` may be important. When reading back data that was zipped before saving; you would definitely want to use a `GZIPInputStream` to make sense of the data before using any other decorated `InputStream` objects.

Can you describe an implementation of the Flyweight Pattern?

The Flyweight Pattern can be a useful pattern in situations where you have several objects, and many may represent the same value. In these instances, it can be possible to share the values as long as the objects are immutable. Listing 6-9 shows an example from the Java standard library, taken from Sun's implementation of the `Integer` class.

LISTING 6-9: The Flyweight Pattern in action

```

public static Integer valueOf(int i) {
    assert IntegerCache.high >= 127;
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}

```


How can you use the Singleton Pattern?

A singleton is a class that allows only one instance to be created. It is often used to create a single point of entry to a third party, such as a database or web service, so that the number of connections can be easily managed and configured in one place.

Restricting clients of a singleton class to only one instance can prove challenging. Listing 6-11 shows a naïve first attempt.

LISTING 6-11: A problematic singleton

```
public class Singleton {

    private static Singleton INSTANCE;

    public static Singleton getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new Singleton();
        }

        return INSTANCE;
    }

    public void singletonMethod() {
        // operations here
    }
}
```

The approach here is called *lazy initialization*: The `Singleton` instance is only created when it is first needed. At first glance, it looks like any other calls to `getInstance()` will return that same instance. If, however, `INSTANCE` is `null`, and a thread is switched after the `if` statement but before `INSTANCE` is initialized, then a second (or more) thread calling `getInstance()` will also have the `if` statement return `true` and create a new object. This can result in strange, erratic behavior or worse, memory leaks resulting in the JVM eventually crashing.

Java 5 introduced the `Enum` type. If you create the `Singleton` as a single-element `Enum`, the JVM *guarantees* that only one instance will ever be created, as shown in Listing 6-12.

LISTING 6-12: Using Enum for a Singleton

```
public enum SingletonEnum {

    INSTANCE;

    public void singletonMethod() {
        // operations here
    }
}
```

Be aware that using the Singleton Pattern can lead to other issues: It can be very difficult to test in isolation, especially if the `Singleton` performs heavyweight operations, such as writing to a database. It is often better to simply manage any “singleton-like” object as a dependency on a class, and use a dependency injection framework to construct only one object.

Singletons work best in specialized applications, such as a GUI on a desktop or mobile application, or when you know you will not have many concurrent users. If you are building large, scalable server applications, then singleton objects are often the source of any performance bottlenecks.

SUMMARY

This chapter has been a non-exhaustive look at some common object-oriented software design patterns. Make sure you understand *how* they work and *why* they are used. Other patterns within the Java language and libraries have not been discussed here, such as the Iterator Pattern, Factory Pattern, and many more. Once you are familiar with these patterns, you should be able to easily see them throughout the language and libraries.

Take the time to read and investigate more design patterns. Many free online resources describe more design patterns with example implementations. The original design patterns book, *Design Patterns: Elements of Reusable Object-Oriented Software*, (Ralph Johnson, John Vlissides, Richard Helm, and Erich Gamma: Addison-Wesley, October 1994) is a fantastic resource for this topic, although the examples given are not in Java.

Once you are comfortable with object-oriented patterns, you should examine where else you see patterns in software. Another, more modern, book on software patterns, covering the patterns found across full systems is *Enterprise Integration Patterns* (Gregor Hohpe and Bobby Woolf: Addison-Wesley Signature, October 2003).

The more experience you get with using design patterns, the more you will spot them when reviewing code or even API documentation. This allows you to get up to speed quicker when understanding and learning new APIs and functionality.

The next chapter looks at a different kind of pattern: some commonly occurring questions that are often favorites with interviewers.

7

Implementing Popular Interview Algorithms

This chapter looks at some popular questions used in interviews. Although these questions will probably not come up in the exact form presented here, they serve as a good introduction to the level and approach that interviewers take.

Many of the questions here have been taken from the website Interview Zen, (www.interviewzen.com) where they have been used in real interviews for real developers.

When you are asked in an interview to write or develop an algorithm, you should be focusing on how *efficient* the algorithm is. Efficiency can take several forms: The running time of an algorithm is usually the main point of focus, but it is also important to consider the space an algorithm uses. Quite often, you will need to strike a balance between these two depending on your requirements.

A common method for describing the efficiency of an algorithm is called *Big O Notation*. This is used to describe how an algorithm reacts to a change to its input.

An algorithm with $O(n)$ is said to have *linear complexity*: As the size of the input, n , increases, the performance of the algorithm increases by a proportional amount.

An $O(n^2)$ complexity is said to be *quadratic*: if you tripled the size of the input, the algorithm would take nine times longer to run. As you would expect, quadratic, or worse, exponential ($O(n^n)$) algorithms should be avoided where possible.

IMPLEMENTING FIZZBUZZ

Write an algorithm that prints all numbers between 1 and n , replacing multiples of 3 with the String `Fizz`, multiples of 5 with `Buzz`, and multiples of 15 with `FizzBuzz`.

This question is extremely popular, and definitely one of the most used on Interview Zen. The implementation should not be too hard, and is shown in Listing 7-1.

LISTING 7-1: A simple implementation of the “FizzBuzz” interview question

```

public static List<String> fizzBuzz(final int n) {
    final List<String> toReturn = new ArrayList<>(n);
    for (int i = 1; i <= n; i++) {
        if (i % 15 == 0) {
            toReturn.add("FizzBuzz");
        } else if (i % 3 == 0) {
            toReturn.add("Fizz");
        } else if (i % 5 == 0) {
            toReturn.add("Buzz");
        } else {
            toReturn.add(Integer.toString(i));
        }
    }

    return toReturn;
}

```

The one catch is to check for the FizzBuzz case before the others, because values divisible by 15 are also divisible by 3 and 5. You would miss this if you checked the 3 and 5 cases first.

Interviewers often look for examples of code reuse, and logical abstraction. Listing 7-1 does not do this particularly well.

It should be possible to provide an abstraction over the different cases for Fizz, Buzz, and FizzBuzz. This would allow for the different cases to be easily tested in isolation, and even changed to return different words for different values, should you wish. This is shown in Listing 7-2.

LISTING 7-2: Abstraction with FizzBuzz

```

public static List<String> alternativeFizzBuzz(final int n) {
    final List<String> toReturn = new ArrayList<>(n);
    for (int i = 1; i <= n; i++) {
        final String word =
            toWord(3, i, "Fizz") + toWord(5, i, "Buzz");

        if (StringUtils.isEmpty(word)) {
            toReturn.add(Integer.toString(i));
        }
        else {
            toReturn.add(word);
        }
    }
    return toReturn;
}

private static String toWord(final int divisor,
                             final int value,
                             final String word) {
    return value % divisor == 0 ? word : "";
}

```


When the `toWord` method does not return a match, it returns an empty `String`. This allows you to check for 3 and 5 together: if a number is divisible by 3 but not 5, then the 5 case will return an empty `String`, and so `Fizz` will be concatenated with an empty `String`. When a number is divisible by 15, that is, divisible by both 3 and 5, then `Fizz` will be concatenated with `Buzz`, giving `FizzBuzz`.

A check is still required to see if either check was successful: An empty `String` denotes neither check was divisible.

Write a method that returns a Fibonacci sequence from 1 to *n*.

DEMONSTRATING THE FIBONACCI SEQUENCE

The Fibonacci sequence is a list of numbers, where the next value in the sequence is the sum of the previous two. The sequence defines that the first number is zero, and the next is one.

Such an algorithm will have a method signature like:

```
public static List<Integer> fibonacci(int n)
```

That is, given a value *n*, the method will return a list, in order, of the Fibonacci sequence to *n*. The method can be static, because it does not rely on any object state or an instance to be constructed.

The parameter has one main constraint: *n* must be greater than or equal to zero. Due to the definition of the sequence, there is also some efficiency that can be applied when the parameter is 0, 1, or 2. If the parameter is zero, return an empty list. If the parameter is 1, return a list containing a solitary zero, and if the parameter is 2, you can return a list containing 0 and 1 in that order.

For any other value of *n*, you can build the list one value at a time, simply adding the last two values in the list, starting with a list containing 0 and 1. Listing 7-3 shows a working, iterative approach.

LISTING 7-3: An iterative Fibonacci sequence

```
public static List<Integer> fibonacci(int n) {
    if (n < 0) {
        throw new IllegalArgumentException(
            "n must not be less than zero");
    }

    if (n == 0) {
        return new ArrayList<>();
    }

    if (n == 1) {
        return Arrays.asList(0);
    }

    if (n == 2) {
        return Arrays.asList(0, 1);
    }
}
```

continues

LISTING 7-3 *(continued)*

```

    final List<Integer> seq = new ArrayList<>(n);
    seq.add(0);
    n = n - 1;
    seq.add(1);
    n = n - 1;

    while (n > 0) {
        int a = seq.get(seq.size() - 1);
        int b = seq.get(seq.size() - 2);
        seq.add(a + b);
        n = n - 1;
    }

    return seq;
}

```

By decreasing the value of *n* each time a value is added, once *n* has reached zero, you have added enough values to the list. Retrieval of the previous two values in the array is guaranteed not to throw an `ArrayIndexOutOfBoundsException`, because you always start with a list containing at least two values.

You can write a simple unit test to verify that the algorithm works, as shown in Listing 7-4.

LISTING 7-4: Testing the Fibonacci sequence

```

@Test
public void fibonacciList() {
    assertEquals(0, fibonacci(0).size());
    assertEquals(Arrays.asList(0), fibonacci(1));
    assertEquals(Arrays.asList(0, 1), fibonacci(2));
    assertEquals(Arrays.asList(0, 1, 1), fibonacci(3));
    assertEquals(Arrays.asList(0, 1, 1, 2), fibonacci(4));
    assertEquals(Arrays.asList(0, 1, 1, 2, 3), fibonacci(5));
    assertEquals(Arrays.asList(0, 1, 1, 2, 3, 5), fibonacci(6));
    assertEquals(Arrays.asList(0, 1, 1, 2, 3, 5, 8), fibonacci(7));
    assertEquals(Arrays.asList(0, 1, 1, 2, 3, 5, 8, 13), fibonacci(8));
}

```

You should test that a negative parameter would throw an exception, too.

Write a method that returns the *n*th value of the Fibonacci sequence.

This is a variation on the previous question. Rather than returning the whole Fibonacci sequence, this method only returns the value at position *n*. This algorithm would likely have a method signature like this:

```
public static int fibN(int n)
```

Passing the value 0 would return 0; 1 would return 1; 2 would return 1; and so on.

A particularly naïve implementation could call the method developed in the previous question, and return the last value in the list. This, however, consumes a large amount of memory in creating the list, only to throw all but one value away. If you were to look for the millionth entry, this would take an unacceptable amount of time to run, and would need a lot of memory.

One better approach could be a recursive solution. From the algorithm definition:

$$\text{Fibonacci}(n) = \text{Fibonacci}(n - 1) + \text{Fibonacci}(n - 2)$$

there are two base cases:

$$\begin{aligned}\text{Fibonacci}(1) &= 1 \\ \text{Fibonacci}(0) &= 0\end{aligned}$$

This recursive definition will make an attractive recursive algorithm, as defined in Listing 7-5.

LISTING 7-5: A recursive algorithm for finding the nth Fibonacci number

```
public static int fibN(int n) {
    if (n < 0) {
        throw new IllegalArgumentException(
            "n must not be less than zero");
    }
    if (n == 1) return 1;
    if (n == 0) return 0;
    return (fibN(n - 1) + fibN(n - 2));
}
```

As elegant as this solution appears, it is extremely inefficient. If you were calculating, for instance, the 45th value, this is made from the sum of `Fibonacci(43)` and `Fibonacci(44)`. `Fibonacci(44)` itself is made from the sum of `Fibonacci(42)` and `Fibonacci(43)`. This algorithm would calculate `Fibonacci(43)` on both occasions here, and this would happen many times throughout the whole calculation. This is more apparent in Figure 7-1.

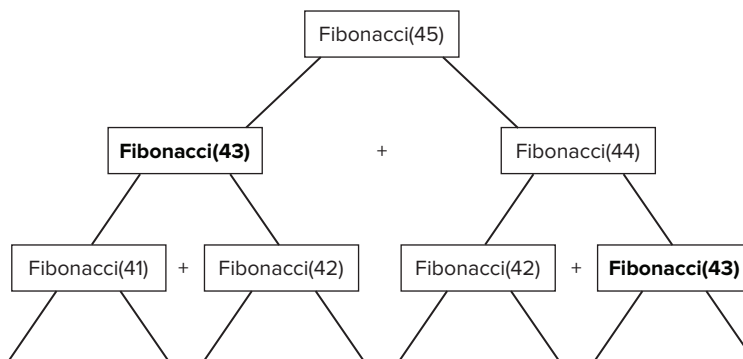


FIGURE 7-1

USING RETURN WITH IF STATEMENTS

Whenever you have a `return` statement inside an `if` statement, you do not need to provide an `else` clause. If an `if` statement condition is false, the code execution continues after the block. If the statement is true, the method execution will finish with the `return` statement.

This has the advantage that your code does not need an extra level of indentation due to the redundant `else` block:

```
if (condition) {
    // code for condition == true
    return valueIfTrue;
}

// more code for condition == false
return valueIfFalse;
```

By caching any calculated results, only unknown Fibonacci numbers would be calculated on a particular run. The results can be stored locally in a map. Listing 7-6 shows an improved approach to Listing 7-5.

LISTING 7-6: Caching previously computed Fibonacci numbers

```
private Map<Integer, Integer> fibCache = new HashMap<>();

public int cachedFibN(int n) {
    if (n < 0) {
        throw new IllegalArgumentException(
            "n must not be less than zero");
    }
    fibCache.put(0, 0);
    fibCache.put(1, 1);
    return recursiveCachedFibN(n);
}

private int recursiveCachedFibN(int n) {
    if (fibCache.containsKey(n)) {
        return fibCache.get(n);
    }

    int value = recursiveCachedFibN(n - 1) + recursiveCachedFibN(n - 2);
    fibCache.put(n, value);
    return value;
}
```

Note here that the recursive method was split into a separate method, to detach the calculation from the setup of the map. Listing 7-7 shows a small unit test to measure the decrease in calculation time due to this optimization.

LISTING 7-7: Measuring the performance increase

```
@Test
public void largeFib() {
    final long nonCachedStart = System.nanoTime();
    assertEquals(1134903170, fibN(45));
    final long nonCachedFinish = System.nanoTime();
    assertEquals(1134903170, cachedFibN(45));
    final long cachedFinish = System.nanoTime();

    System.out.printf(
        "Non cached time: %d nanoseconds%n",
        nonCachedFinish - nonCachedStart);
    System.out.printf(
        "Cached time: %d nanoseconds%n",
        cachedFinish - nonCachedFinish);
}
```

At the time of writing, it produced the following output:

```
Non cached time: 19211293000 nanoseconds
Cached time: 311000 nanoseconds
```

Quite an improvement: from 19 seconds to 0.000311 seconds.

When the result of a method call is cached to save recalculating the result again later, this is called *memoization*.

The Big O Notation for the original implementation to calculate the n th Fibonacci number, is $O(n^n)$, and thus is extremely inefficient; too inefficient for anything useful than very low values of n . The memoized implementation has $O(n)$, with the additional stipulation that it must store each value between 0 and n .

DEMONSTRATING FACTORIALS

Write a factorial implementation that does not use recursion.

A factorial method would be another good candidate for using recursion. The general algorithm for a factorial is:

$$\text{Factorial}(n) = n \times \text{Factorial}(n - 1)$$

If you decide not to implement this in a recursive fashion, you can simply loop up to n , keeping track of the value to return, as shown in Listing 7-8.

LISTING 7-8: An iterative implementation of a factorial

```
public static long factorial(int n) {
    if (n < 1) {
        throw new IllegalArgumentException(
            "n must be greater than zero");
    }

    long toReturn = 1;
    for (int i = 1; i <= n; i++) {
        toReturn *= i;
    }

    return toReturn;
}
```

Be aware that with factorials the calculated value grows large extremely quickly, which is why this method returns a long. You could even make a case for returning a `BigInteger` instance here. `BigInteger` objects have no upper bound; they allow for numerical integer values of any size.

IMPLEMENTING LIBRARY FUNCTIONALITY

Given a list of words, produce an algorithm that will return a list of all anagrams for a specific word.

An anagram is a word that, when the letters are rearranged, produces another word. No letters can be removed or taken away.

A first attempt at writing an algorithm for this question could be something like the following pseudocode:

```
given a word a, and a list of words ws
given an empty list anagrams
for each word w in ws:
    if (isAnagram(w, a)) add w to anagrams
return anagrams

method isAnagram(word1, word2):
    for each letter letter in word1:
        if (word2 contains letter):
            remove letter from word2
            continue with the next letter
        else return false

    if (word2 is empty) return true
    else return false
```

This pseudocode would definitely work. It examines each word in the given list and compares that word to the one to check. If all the letters match and there are no more, or fewer letters, it's a match.

This algorithm is very inefficient. For a list of length m , and an average word length of n , the algorithm scans each word n times, one for each letter to check. This is measured as $O(m \times n!)$, which is simply $O(n!)$, because as n grows large, it will dominate the performance over m . *Each* letter in *each* word must be scanned for *each* letter in the word to check against. With a small amount of preparation, it is possible to slash the running time of this algorithm.

Every word can be thought to have an “algorithm signature.” That is, for all anagrams of a given word, they will all have the same signature. That signature is simply the letters of the word rearranged into alphabetical order.

If you store each word in a map against its signature, when you are presented with a word you calculate its signature and return the map entry of the list of words indexed at that signature. Listing 7-9 shows this implementation.

LISTING 7-9: An efficient anagram finder

```
public class Anagrams {

    final Map<String, List<String>> lookup = new HashMap<>();

    public Anagrams(final List<String> words) {

        for (final String word : words) {
            final String signature = alphabetize(word);
            if (lookup.containsKey(signature)) {
                lookup.get(signature).add(word);
            } else {
                final List<String> anagramList = new ArrayList<>();
                anagramList.add(word);
                lookup.put(signature, anagramList);
            }
        }
    }

    private String alphabetize(final String word) {
        final byte[] bytes = word.getBytes();
        Arrays.sort(bytes);
        return new String(bytes);
    }

    public List<String> getAnagrams(final String word) {
        final String signature = alphabetize(word);
        final List<String> anagrams = lookup.get(signature);
        return anagrams == null ? new ArrayList<String>() : anagrams;
    }
}
```

The `alphabetize` method creates the signature for use in storing the word, and also for lookup when querying for any anagrams. This method makes use of the sorting algorithm in Java’s standard library, which was covered in Chapter 4 if you want to write a sort yourself.

The trade-off here is that you need to take time to preprocess the list into the different buckets of anagrams, and you will also need extra storage space. However, lookup times will drop from a potential $O(n!)$ to $O(1)$ for a list of length m with an average word length of n . It is $O(1)$ because the hash function is able to directly locate the list of anagrams stored in the map. Don't forget, there is the overhead to create the lookup in the first place, so the whole lookup algorithm has a Big O Notation of $O(m)$.

Write a method to reverse a String.

In Java, a `String` is immutable, so once it has been constructed it is not possible to change the contents. So when you are asked to reverse a `String`, you are actually being asked to produce a new `String` object, with the contents reversed:

```
public static String reverse(final String s)
```

One approach to reversing a `String` is to iterate over the contents in reverse, populating a new container such as a `StringBuilder` as you go, as shown in Listing 7-10.

LISTING 7-10: Reversing a String using a StringBuilder

```
public static String reverse(final String s) {  
    final StringBuilder builder = new StringBuilder(s.length());  
    for (int i = s.length() - 1; i >= 0; i--) {  
        builder.append(s.charAt(i));  
    }  
    return builder.toString();  
}
```

USING BUILT-IN REVERSE METHODS

The `StringBuilder` class has a `reverse` method, which you should use if necessary when programming in your day job.

If you are ever asked to implement an algorithm that is also provided by the Java standard libraries, such as the `String` reversal, sorting, or searching, then you should not use these methods: The interviewer is assessing how *you* attempt these problems.

If the algorithm you need to use is secondary, or otherwise auxiliary in whatever you are writing, such as the `String` sorting in Listing 7-9, then do use the built-in methods so you can concentrate on the details of your application.

This approach, while fine, does require a large amount of memory. It needs to hold the original `String` and the `StringBuilder` in memory. This could be problematic if you were reversing some data of several gigabytes in size.

It is possible to reverse the `String` in place, by loading the whole `String` instance into a `StringBuilder`, traversing the characters from one end, and swapping with the character the same length away from the other end of the `String`. This only requires one extra character of memory to facilitate the swapping of characters. Listing 7-11 shows this.

LISTING 7-11: Reversing a `String` in place

```
public static String inplaceReverse(final String s) {
    final StringBuilder builder = new StringBuilder(s);
    for (int i = 0; i < builder.length() / 2; i++) {
        final char tmp = builder.charAt(i);
        final int otherEnd = builder.length() - i - 1;
        builder.setCharAt(i, builder.charAt(otherEnd));
        builder.setCharAt(otherEnd, tmp);
    }

    return builder.toString();
}
```

A common mistake with this approach is to traverse the whole `String`, rather than stopping halfway. If you traverse the whole `String`, you switch each character twice, and it returns to its original position! Simple errors like this can be easily mitigated with a small number of unit tests.

How do you reverse a linked list in place?

Following from `String` reversal, it is also possible to reverse a generic linked list in place. Given a linked list class definition:

```
public class LinkedList<T> {
    private T element;
    private LinkedList<T> next;

    public LinkedList(T element, LinkedList<T> next) {
        this.element = element;
        this.next = next;
    }

    public T getElement() {
        return element;
    }

    public LinkedList<T> getNext() {
        return next;
    }
}
```

This *recursive class definition* takes other `LinkedList` instances, and assumes `null` to represent the end of the list.

This being a recursive definition, it makes sense to use a recursive algorithm to reverse the list. You would expect a method to perform the reversal to have a signature like the following:

```
public static <T> LinkedList<T> reverse(final LinkedList<T> original)
```

This may look like it is returning a new copy of the elements in the original list, but actually, it is returning a reference to the last element in the structure, with `next` pointing to the `LinkedList` instance that was originally immediately before that.

The base case for this algorithm is when the `next` reference is `null`. If you have a one-element list, reversing that list is simply the same list.

The recursive step for this algorithm is to remove the link to the `next` element, reverse that `next` element, and then set the `next` element's own `next` to be the current element—the original parameter. Listing 7-12 shows a possible implementation.

LISTING 7-12: Reversing a linked list recursively

```
public static <T> LinkedList<T> reverse(final LinkedList<T> original) {
    if (original == null) {
        throw new NullPointerException("Cannot reverse a null list");
    }

    if (original.getNext() == null) {
        return original;
    }
    final LinkedList<T> next = original.next;
    original.next = null;

    final LinkedList<T> othersReversed = reverse(next);

    next.next = original;

    return othersReversed;
}
```

Figure 7-2 shows a linked list constructed in the unit test in Listing 7-13.



FIGURE 7-2

LISTING 7-13: Testing the linked list reversal

```
@Test
public void reverseLinkedList() {
    final LinkedList<String> three = new LinkedList<>("3", null);
    final LinkedList<String> two = new LinkedList<>("2", three);
}
```

```

final LinkedList<String> one = new LinkedList<>("1", two);

final LinkedList<String> reversed = LinkedList.reverse(one);

assertEquals("3", reversed.getElement());
assertEquals("2", reversed.getNext().getElement());
assertEquals("1", reversed.getNext().getNext().getElement());
}

```

How do you test if a word is a palindrome?

A palindrome is a word, or phrase, that when the letters are reversed, the spelling is the same. Some examples include, “eve,” “level,” or, ignoring the space, “top spot.”

An algorithm to check for palindromes is similar to that for reversing a String, although you need to take non-letter characters into account. Listing 7-14 is an example.

LISTING 7-14: A palindrome checker

```

public static boolean isPalindrome(final String s) {
    final String toCheck = s.toLowerCase();
    int left = 0;
    int right = toCheck.length() - 1;

    while (left <= right) {
        while(left < toCheck.length() &&
            !Character.isLetter(toCheck.charAt(left))) {
            left++;
        }
        while(right > 0 && !Character.isLetter(toCheck.charAt(right))) {
            right--;
        }
        if (left > toCheck.length() || right < 0) {
            return false;
        }

        if (toCheck.charAt(left) != toCheck.charAt(right)) {
            return false;
        }

        left++;
        right--;
    }

    return true;
}

```

This implementation comprises two pointers: one looking at the left half of the String, and one looking at the right. As soon as the pointers cross, the check is complete.

By ignoring any characters that are not letters, each pointer needs to be individually checked until it is pointing to the next letter in the `String`. A successful comparison between two characters does not automatically imply the word is a palindrome, because another pair of letters may not match. As soon as one comparison fails, you know that the `String` does not read the same in reversed form.

If you *do* care about `Strings` being exactly the same in reverse, taking all punctuation into account, Listing 7-15 shows a much simpler approach using the `reverse` method from Listing 7-10.

LISTING 7-15: A strict palindrome checker

```
public static boolean strictPalindrome(final String s) {
    return s.equals(reverse(s));
}
```

You can reuse your `reverse` method, leading to a one-line method: The fewer lines of code you need to write, the fewer places you can introduce bugs. Assuming the `reverse` method is fully tested, writing unit tests for this method will be very, very simple.

Write an algorithm that collapses a list of `Iterators` into a single `Iterator`.

Such an algorithm would have a signature like the following:

```
public static <T> Iterator<T> singleIterator(
    final List<Iterator<T>> iteratorList)
```

The `Iterator` interface has the following definition:

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

This algorithm will implement the `Iterator` interface, creating a new implementation that will treat the underlying list of `Iterators` as a single `Iterator`.

You will use the `Iterators` one at a time, and once one has been exhausted, you move on to the next `Iterator`.

The `hasNext` method can be a particular red herring, because when your current `Iterator` has been exhausted, it will return `false`, but you may still have `Iterators` to consume. Once all `Iterators` have been consumed, this method will return `false`.

Listing 7-16 shows the constructor for a `ListIterator` class, along with the `hasNext` method.

LISTING 7-16: Creating an Iterator of Iterators

```

public static class ListIterator<T> implements Iterator<T> {

    private final Iterator<Iterator<T>> listIterator;
    private Iterator<T> currentIterator;

    public ListIterator(List<Iterator<T>> iterators) {
        this.listIterator = iterators.iterator();
        this.currentIterator = listIterator.next();
    }

    @Override
    public boolean hasNext() {
        if(!currentIterator.hasNext()) {
            if (!listIterator.hasNext()) {
                return false;
            }

            currentIterator = listIterator.next();
            hasNext();
        }

        return true;
    }
    ... // see below for the remaining implementation

```

The class holds two fields: the complete `Iterator of Iterators`, and the current `Iterator`.

When the `hasNext` method is called, it checks to see if the current `Iterator` has any more elements. If so, the method simply returns `true`. If it is false, the method checks to see if the `Iterator of Iterators` has any more elements. If that, too, is exhausted, there are no more elements at all.

If the `Iterator of Iterators` does have more elements, then the `currentIterator` is set to the next iterator, and the `hasNext` method is called recursively. This means that the `currentIterator` reference could potentially advance several iterators before finding one where the `hasNext` method returns `true`, but the recursion will take care of that.

For the `next` and `remove` methods on the `Iterator` interface, these too pose problems. It is possible for the current `Iterator` to be exhausted, and therefore calling `next` would return a `NoSuchElementException`, but one of the following `Iterators` does have elements. Before calling `next`, the `currentIterator` reference needs to be updated. The code to update the `currentIterator` reference to the correct reference was written in the `hasNext` method, so you can simply call `hasNext`, and ignore the result. It is exactly the same for the `remove` method: Simply update the `currentIterator` reference before actually calling `remove`. Listing 7-17 shows the implementation.

LISTING 7-17: The next and remove methods for the Iterator of Iterators

```

@Override
public T next() {

```

continues

LISTING 7-17 *(continued)*

```

        hasNext();
        return currentIterator.next();
    }

    @Override
    public void remove() {
        hasNext();
        currentIterator.remove();
    }

```

Listing 7-18 shows a sample unit test to verify the implementation.

LISTING 7-18: Testing the Iterator of Iterators

```

@Test
public void multipleIterators() {
    final Iterator<Integer> a = Arrays.asList(1, 2, 3, 4, 5).iterator();
    final Iterator<Integer> b = Arrays.asList(6).iterator();
    final Iterator<Integer> c = new ArrayList<Integer>().iterator();
    final Iterator<Integer> d = new ArrayList<Integer>().iterator();
    final Iterator<Integer> e = Arrays.asList(7, 8, 9).iterator();

    final Iterator<Integer> singleIterator =
        Iterators.singleIterator(Arrays.asList(a, b, c, d, e));

    assertTrue(singleIterator.hasNext());
    for (Integer i = 1; i <= 9; i++) {
        assertEquals(i, singleIterator.next());
    }
    assertFalse(singleIterator.hasNext());
}

```

USING GENERICS

Write a method that replicates a list of `Integers`, with 1 added to each element.

On examination, this question appears trivially easy, as shown in Listing 7-19.

LISTING 7-19: Adding one to a list of numbers

```

public static List<Integer> addOne(final List<Integer> numbers) {
    final ArrayList<Integer> toReturn = new ArrayList<>(numbers.size());

    for (final Integer number : numbers) {
        toReturn.add(number + 1);
    }
}

```

```

    }

    return toReturn;
}

```

Even adding a parameter to the method to allow you to add a different value rather than 1 is just as simple. But what if the question is modified to allow you to perform any operation on each element in the list?

Providing any client code with the facility to perform any operation can be particularly powerful. To allow any Integer operation, you can define a new interface, `IntegerOperation`, which has one method, taking an `Integer`, and returning an `Integer`:

```

public interface IntegerOperation {
    Integer performOperation(Integer value);
}

```

An implementation of this to add one would simply be as follows:

```

public class AddOneOperation implements IntegerOperation {
    @Override
    public Integer performOperation(Integer value) {
        return value + 1;
    }
}

```

Listing 7-20 modifies the original `addOne` method to accept the `IntegerOperation` interface, and gives it a more appropriate name.

LISTING 7-20: Performing any Integer operation on a list

```

public static List<Integer> updateList(final List<Integer> numbers,
                                       final IntegerOperation op) {
    final ArrayList<Integer> toReturn = new ArrayList<>(numbers.size());
    for (final Integer number : numbers) {
        toReturn.add(op.performOperation(number));
    }

    return toReturn;
}

```

Listing 7-21 shows a unit test, with an anonymous implementation of `IntegerOperation`.

LISTING 7-21: Testing the `updateList` method

```

@Test
public void positiveList() {
    final List<Integer> numbers = Arrays.asList(4, 7, 2, -2, 8, -5, -7);
    final List<Integer> expected = Arrays.asList(4, 7, 2, 2, 8, 5, 7);

    final List<Integer> actual =

```

continues

LISTING 7-21 *(continued)*

```

        Lists.updateList(numbers, new IntegerOperation() {
            @Override
            public Integer performOperation(Integer value) {
                return Math.abs(value);
            }
        });

        assertEquals(expected, actual);
    }

```

This `IntegerOperation` implementation makes all values in the list positive.

The power of abstracting this operation should be clear. The `updateList` method has the ability to do *anything* to a list of `Integers`, such as ignoring whatever the value in the list is and replacing it with a constant, or even something more exotic such as printing the value to the console, or making a call to a URL with that number.

It would be even better if the `updateList` could be even more abstract; currently it only works with `Integers`. What if it could work with a list of *any* type, and return a list of a *different* type?

Take the following interface:

```

public interface GenericOperation<A, B> {
    B performOperation(A value);
}

```

This interface uses Java generics on the method `performOperation` to *map* the value passed into a different type, such as turning a `String` parameter into an `Integer`:

```

public class StringLengthOperation
    implements GenericOperation<String, Integer> {
    @Override
    public Integer performOperation(String value) {
        return value.length();
    }
}

```

Listing 7-22 shows an implementation *mapping* the list from one of a certain type into a list of another.

LISTING 7-22: Mapping a list into a different type

```

public static <A, B> List<B> mapList(final List<A> values,
                                    final GenericOperation<A, B> op) {
    final ArrayList<B> toReturn = new ArrayList<>(values.size());
    for (final A a : values) {
        toReturn.add(op.performOperation(a));
    }

    return toReturn;
}

```


Listing 7-23 shows how you can use this in conjunction with the `StringLengthOperation` class.

LISTING 7-23: Testing the list mapping functionality

```
@Test
public void stringLengths() {
    final List<String> strings = Arrays.asList(
        "acing", "the", "java", "interview"
    );
    final List<Integer> expected = Arrays.asList(5, 3, 4, 9);
    final List<Integer> actual =
        Lists.mapList(strings, new StringLengthOperation());
    assertEquals(expected, actual);
}
```

Providing a generic abstraction such as this is something that functional languages, such as Scala provide within the language itself. Treating a function as a value, such as a `String` or an `Integer` allows them to be passed directly to functions. For more information about this, the appendix looks at Scala, and how to use functions as values.

SUMMARY

The final question in this chapter shows that even remarkably simple questions can often expand into longer discussions, and this is often an approach taken in face-to-face interviews. Interviewers will start with simple questions, and will probe deeper and deeper to try to find the extent of your knowledge.

To prepare for answering questions in interviews, try to find some programming exercises to do at home. The questions in this chapter are some of the more popular questions taken from Interview Zen, but other common programming questions are some of the core computer science topics, such as sorting and searching, and algorithms around data structures, such as lists and maps.

This chapter has attempted to strike a balance between recursive and iterative implementations. Often, a recursive approach can appear simpler, and perhaps more elegant, although this comes at a price. Particular algorithms that perform a high number of recursive calls can throw a `StackOverflowException`, and may even be slower in performance over their iterative counterparts due to the extra setup in stack allocation for calling a method.

Tail recursive calls can sometime be optimized into iterative approaches. Tail recursion is when the recursive step is the final step of the method: The compiler can replace the stack variables for the current method with those for the next method call, rather than creating a new stack frame. The JVM itself cannot handle this directly, but some compilers can do this for Java and other JVM-supported languages.

Regardless of a recursive or iterative implementation, always consider the performance of your algorithm. Many algorithms asked in interviews may have a simple, yet inefficient answer, and a more effective approach may become clearer with a little thought.

The reason for interviewers asking questions like the ones in this chapter is to examine your ability to write legible, reusable, clean code. There are plenty of other sources around for refactoring and writing code that is testable. Speaking with colleagues, and performing active code reviews is a great way to take feedback from anything you have written and to lean on the experience of others.

The next section of the book looks at core Java topics, covering the basics of the language and its libraries, as well as focusing strongly on unit testing.

PART II

Core Java

This part concentrates on fundamentals of Java, which are relevant for any Java interview, independent of the specifics of any job application.

Chapter 8 covers essentials of the language, and how the Java language interacts with the JVM. It highlights techniques and corner-cases that all Java developers should know and make use of regularly.

Chapter 9 is about unit testing, using JUnit. It discusses why unit testing is important, and how to use JUnit to demonstrate an upper hand in any assessment.

Chapter 10 complements Chapter 8, concentrating on the JVM, rather than the language, illustrating how you can demonstrate a thorough understanding of the platform.

Chapter 11 is about concurrency, which is often a favorite topic for interview questions: Most server-side development roles need to deal with concerns around scaling and parallel processing. This chapter looks at the tools available in the Java standard library, and also at some third-party tools.

8

Java Basics

This chapter covers the main elements of the Java ecosystem, including the core parts of the language, specific conventions to follow, and the use of core libraries. In any interview, as a bare minimum, an interviewer would expect you to thoroughly understand these topics. Any questions similar to the ones here should be seen as introductory, and would usually be used early in an interview session, or perhaps even used as pre-interview screening questions.

A NOTE ON THE CODE SAMPLES

All Java code is written as small, independent JUnit tests.

This is a good discipline to carry through for any small coding interview questions and is generally a good technique for understanding how code works. Rather than polluting classes with a `main()` method to execute test code, writing JUnit tests provides isolated, independent snippets of code. On a project managed by Maven, or something similar, this code would not be loaded into a live production system either.

While attempting to understand the code, rather than scattering plenty of `System.out.println()` or `logger.debug()` lines, you can make assertions, and use the JUnit framework to test these assertions. A passing test means your assertions are correct.

These small exploratory snippets of code become tests in their own right: As long as the tests are consistently passing, you have created the beginnings of a test suite.

The more tests you write, the better you become at reasoning about existing code, and it also helps to break down code into small, atomic, isolated chunks.

Any time you are asked to do a small programming assignment as part of an interview, you should always write tests, whether you were asked to or not. It demonstrates that you are disciplined; have mature reasoning for how to understand and test code; and also provides a level of documentation, which is an aid to any interviewer to how you were thinking at the time.

continues

(continued)

If you are unfamiliar with JUnit, or unit testing in general, feel free to look ahead to Chapter 9, but do make sure you return to this chapter. Alternatively, try to understand what JUnit is doing with the simple code samples in this chapter.

Only the test method is included each listing. The class definition, package definition, and any imports are omitted, unless it is relevant to the listing.

THE PRIMITIVE TYPES

Name some of Java’s primitive types. How are these handled by the JVM?

The basic types in Java such as `boolean`, `int`, and `double` are known as *primitive types*. The JVM treats these differently from *reference types*, otherwise known as *objects*. Primitives always have a value; they can never be `null`.

When you are defining a variable for `ints` and `longs`, the compiler needs to be able to differentiate between the two types. You do this by using the suffix `L` after `long` values. The absence of `L` implies an `int`. The `L` can be either in upper- or lowercase, but it is recommended to always use uppercase. This way, anyone reading your code can easily tell the difference between the `L` and the number `1`! This is also used with `floats` and `doubles`: `floats` can be defined with `F`, and `doubles` with `D`. The suffix is optional for `doubles`; its omission implies a `double`. Table 8-1 shows the primitive types and their sizes.

TABLE 8-1: Primitive Types and Their Sizes

PRIMITIVE	SIZE (BITS)
<code>boolean</code>	1
<code>short</code>	16
<code>int</code>	32
<code>long</code>	64
<code>float</code>	32
<code>double</code>	64
<code>char</code>	16

Be aware that `char` is unsigned. That is, the range of values a `char` can take is from 0 to 65,535, because chars represent Unicode values.

If a primitive type is not set to a value when a variable is defined, it takes a default value. For `boolean` values, this is `false`. For the others, this is a representation of zero, such as 0 for `ints` or 0.0f for `floats`.

The compiler can promote values to their appropriate type when necessary, as shown in Listing 8-1.

LISTING 8-1: Implicit upcasting to wider types

```
int value = Integer.MAX_VALUE;
long biggerValue = value + 1;
```

With the exception of `chars`, the compiler can automatically use a wider type for a variable, because it is not possible to lose any precision, for instance, when stepping up from an `int` to a `long`, or a `float` to a `double`. The reverse is not true. When trying to assign a `long` value to an `int`, you must cast that value. This tells the compiler this is actually what you intend, and you are happy to lose precision, as shown in Listing 8-2.

LISTING 8-2: Explicit downcasting to narrower types

```
long veryLargeNumber = Long.MAX_VALUE;
int fromLargeNumber = (int) veryLargeNumber;
```

Only in the rarest of cases would you actually want to downcast. If you find yourself downcasting variables regularly, you are probably not using the correct type.

Why is there no positive equivalent for `Integer.MIN_VALUE`?

The storage of the binary values of `short`, `int`, and `long` use a representation in memory called *Two's Complement*. For zero and positive binary numbers, Table 8-2 shows that the notation is as you would expect.

TABLE 8-2: Binary Representation of Positive Numbers

DECIMAL REPRESENTATION	BINARY REPRESENTATION
0	0000 0000
1	0000 0001
2	0000 0010

In Two’s Complement notation, the negative equivalent of a positive value is calculated by applying a binary NOT and then adding 1.

If Two’s Complement is an unfamiliar or new concept, it may seem a little alien. It may even appear counter-intuitive, because negation now requires two instructions. The one main advantage to a Two’s Complement representation is that there is only one value for zero: There is no concept of a “negative zero.” This, in turn, means the system can store one extra negative value.

The most negative number available to be stored, according to Table 8-3, has a most significant bit of 1, and all remaining bits are zeros. Negating this by flipping all bits (0111 1111), and then adding one (1000 0000) brings you back to the original value; there is no positive equivalent! If it were necessary to make this value positive, you would have to use a wider type, such as a long, or possibly the more complex reference type of BigInteger, which is effectively unbounded.

If you try to perform a calculation that will result in a value larger than Integer.MAX_VALUE, or lower than Integer.MIN_VALUE, then only the least significant 32 bits will be used, resulting in an erroneous value. This is called *overflow*. Try this yourself—create a small unit test that adds one to Integer.MAX_VALUE, and see what the result is.

Listing 8-3 is a useful investigation, showing you cannot use an int to represent the absolute value of Integer.MIN_VALUE.

TABLE 8-3: Binary Representation of Some Negative Numbers

DECIMAL REPRESENTATION	BINARY REPRESENTATION
-1	1111 1111
-2	1111 1110
...	
-127	1000 0001
-128	1000 0000

LISTING 8-3: Attempting to find the absolute value of the most negative int

```
@Test
public void absoluteOfMostNegativeValue() {
    final int mostNegative = Integer.MIN_VALUE;
    final int negated = Math.abs(mostNegative);
    assertFalse("No positive equivalent of Integer.MIN_VALUE", negated > 0);
}
```

This particular quirk is something to bear in mind if you are ever asked to implement a method for calculating the absolute value of an int, or indeed anything involving negative numbers.

USING OBJECTS

What is a Java object?

An *object* can be defined as a collection of variables, which when viewed collectively represents a single, complex entity, and a collection of methods providing operations relevant to that entity. Concisely, objects encapsulate state and behavior.

After primitive types, all other variables in the Java programming language are reference types, better known as objects. Objects differ from primitives in many ways, but one of the most important distinctions is that there is a notation for the absence of an object, `null`. Variables can be set to `null`, and methods can return `null` too. Attempting to call a method on a `null` reference throws a `NullPointerException`. Listing 8-4 illustrates a `NullPointerException` being thrown. Exceptions are covered in more depth in the “Handling Exceptions” section later in this chapter.

LISTING 8-4: Dealing with `NullPointerException`s

```
@Test(expected = NullPointerException.class)
public void expectNullPointerExceptionToBeThrown() {
    final String s = null;
    final int stringLength = s.length();
}
```

Objects are reference types. What exactly does this mean? With primitive types, when declaring a variable, `int i = 42`, this is being assigned to a location in memory, with the value 42. If, later in the program, another variable is assigned to hold the value currently represented by `i`, say, `int j = i`, then another location in memory has been assigned to that same value. Subsequent changes to `i` do not affect the value of `j`, and vice versa.

In Java, a statement such as `new ArrayList(20)`, requests an area in memory to store that data. When that created object is assigned to a variable, `List myList = new ArrayList(20)`, then it can be said that `myList` points to that memory location. On the surface, this appears to act the same as a primitive type assignment, but it is not. If several variables are assigned to that same created object, known as an *instance*, then they are pointing to the same memory location. Making any changes to one instance will be reflected when accessing the other. Listing 8-5 shows this in action.

LISTING 8-5: Several variables referencing the same instance in memory

```
@Test
public void objectMemoryAssignment() {
    List<String> list1 = new ArrayList<>(20);

    list1.add("entry in list1");
```

continues

LISTING 8-5 *(continued)*

```

        assertTrue(list1.size() == 1);

        List list2 = list1;
        list2.add("entry in list2");
        assertTrue(list1.size() == 2);
    }

```

What effect does the `final` keyword have on object references?

The `final` keyword works in exactly the same way for objects as it does for primitive types. The value is set on variable definition, and after that, *the value stored in that memory location cannot change*. As previously discussed, variable definition and memory allocation are quite different between primitives and objects. Although the object reference cannot change, the values held within that object *can* change, unless they themselves are `final`. Listing 8-6 explores the `final` keyword.

LISTING 8-6: The final keyword on object references

```

@Test
public void finalReferenceChanges() {
    final int i = 42;
    // i = 43; <- uncommenting this line would result in a compiler error

    final List<String> list = new ArrayList<>(20);
    // list = new ArrayList(50); <- uncommenting this line will result in an error
    assertEquals(0, list.size());

    list.add("adding a new value into my list");
    assertEquals(1, list.size());

    list.clear();
    assertEquals(0, list.size());
}

```

How do the visibility modifiers work for objects?

The visibility modifiers control access to the encapsulated state of the class and the methods controlling the instance's behavior. A variable encapsulated within an object, or a method on that object, can have its visibility restricted by one of four definitions as shown in Table 8-4.

Remember that private member variables are available only to that class, not even to any subclasses: Private variables are deemed to be necessary for only that type and no other.

One common misconception about the `private` modifier is that `private` means a variable is only accessible to the instance. In reality, any other instance of that same type can access private member variables.

TABLE 8-4: Visibility Modifiers

VISIBILITY	MODIFIER	SCOPE
Least	private	Visible to any instance of that same class, not to subtypes
	<none>	Visible to any class in the same package
	protected	Visible to any subclasses
Most	public	Visible anywhere

Listing 8-7 shows two cases of accessing private member variables of the same type, but a different instance. Most reputable integrated development environments (IDEs) will provide assistance in generating correct `hashCode` and `equals` methods: Usually these will be accessing the private member variables of the other instance when determining equality.

LISTING 8-7: Accessing private member variables

```

public class Complex {
    private final double real;
    private final double imaginary;

    public Complex(final double r, final double i) {
        this.real = r;
        this.imaginary = i;
    }

    public Complex add(final Complex other) {
        return new Complex(this.real + other.real,
            this.imaginary + other.imaginary);
    }

    // hashCode omitted for brevity

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Complex complex = (Complex) o;

        if (Double.compare(complex.imaginary, imaginary) != 0) return false;
        if (Double.compare(complex.real, real) != 0) return false;

        return true;
    }
}

@Test
public void complexNumberAddition() {

```

continues

LISTING 8-7 (continued)

```

    final Complex expected = new Complex(6, 2);

    final Complex a = new Complex(8, 0);
    final Complex b = new Complex(-2, 2);

    assertEquals(a.add(b), expected);
}

```

For methods and variables, what does `static` mean?

Static methods and variables are defined as belonging to the class and not specifically to an instance. They are common to all instances, and are usually accessed through the class name rather than a specific instance.

Listing 8-8 shows that class member variables can be manipulated either from an instance or by referencing the class itself. It is only recommended to access static methods and variables through the class name, because access through an instance can lead to confusion: Other programmers unfamiliar with the class definition may expect it to be a member of the instance, and will experience strange behavior in other instances after modifying the value.

LISTING 8-8: Accessing class member variables

```

public class ExampleClass {
    public static int EXAMPLE_VALUE = 6;
}

@Test
public void staticVariableAccess() {
    assertEquals(ExampleClass.EXAMPLE_VALUE, 6);

    ExampleClass c1 = new ExampleClass();
    ExampleClass c2 = new ExampleClass();
    c1.EXAMPLE_VALUE = 22; // permitted, but not recommended
    assertEquals(ExampleClass.EXAMPLE_VALUE, 22);
    assertEquals(c2.EXAMPLE_VALUE, 22);
}

```

What are polymorphism and inheritance?

Polymorphism and *inheritance* are two of the core idioms of object-oriented development.

Polymorphism allows you to make a definition for a certain type of behavior, and have many different classes implement that behavior. Inheritance allows you to derive the behavior and definition of a class from a superclass.

When defining a new class, it is possible to inherit definitions and state from a previously defined class, and then add new specific behavior, or override behavior for the new type.

From Listing 8-9, the `Square` class inherits from `Rectangle` (you can say a square *is-a* rectangle). For the `Square` definition, the specific variables storing the length of the sides have been reused, and the `Square` type is enforcing that the width and height are the same.

LISTING 8-9: Creating shapes using inheritance

```
public class Rectangle {

    private final int width;
    private final int height;

    public Rectangle(final int width, final int height) {
        this.width = width;
        this.height = height;
    }

    public int area() {
        return width * height;
    }
}

public class Square extends Rectangle {

    public Square(final int sideLength) {
        super(sideLength, sideLength);
    }
}
```

Considering the *is-a* relationship defined previously, polymorphism can be thought of as using a subclass when a superclass has been asked for. The behavior of the subclass will remain, but the user of the polymorphic type is none the wiser.

Listing 8-10 explores the use of the `Square` and `Rectangle` classes with polymorphism. As far as the `ArrayList` is concerned, it is only dealing with `Rectangles`; it does not understand, nor does it need to understand, the difference between a `Rectangle` and a `Square`. Considering a `Square is-a Rectangle`, the code works fine. Had any specific methods been defined on the `Square` class, these would not have been available to any users of the `rectangles` list, because that method is not available on the `Rectangle` class.

LISTING 8-10: Use of polymorphism with the Java collections

```
@Test
public void polymorphicList() {
    List<Rectangle> rectangles = new ArrayList<>(3);
    rectangles.add(new Rectangle(5, 1));
    rectangles.add(new Rectangle(2, 10));
    rectangles.add(new Square(9));

    assertEquals(rectangles.get(0).area(), 5);
    assertEquals(rectangles.get(1).area(), 20);
    assertEquals(rectangles.get(2).area(), 81);
}
```

Explain how some of the methods on the `Object` class are used when overridden.

Every class running on a JVM inherits from `java.lang.Object`, and thus, any non-final public or protected method can be overridden.

The method `equals(Object other)` is used for testing that two references are logically equal. For collection classes, such as `java.util.TreeSet` or `java.util.HashMap`, these classes make use of an object's `equals` method to determine if an object already exists in the collection. The implementation of `equals` on `Object` compares the memory location of the objects, meaning that if two objects have the same memory location, they are actually the same object, so they must be equal. This is usually not much use, nor is it intended for testing equality, so overriding the method is a must for any need for testing equality.

The rule for `hashCode` is that two equal objects must return the same value. Note that the reverse is not required: that if two objects return the same `hashCode`, they are not necessarily equal. This is a nice feature to have, but not required: The more variance in the `hashCode` between different instances, the more distributed those instances will be when stored in a `HashMap`. Note that `hashCode` returns an `int`—this means that if unequal `hashCode`s meant unequal objects, there would only be a maximum of 2^{32} unique variations of a specific instance. This would be quite a limitation, especially for objects like `Strings`!

The reason for the relationship between `hashCode` and `equals` is in the way collection classes such as `java.util.HashMap` are implemented. As discussed in Chapter 5, the data structure backing a `HashMap` is some sort of table, like an array or a `List`. The `hashCode` value is used to determine which index in the table to use. Because the `hashCode` returns an `int`, this means that the value could be negative, or perhaps a value larger than the size of the table. Any `HashMap` implementation will have to manipulate this value to a meaningful index for the table.

Like `equals`, `hashCode` also uses the memory location to generate the `hashCode` value on the `Object` class. This means that two separate instances that are logically equal will be in different memory locations, and therefore will not return the same `hashCode`.

The behavior of `Set` in Listing 8-11 is not performing as expected. The set does not know how to compare the `Person` objects properly, because `Person` does not implement `equals`. This can be fixed by implementing an `equals` method, which compares the name string and the age integer.

LISTING 8-11: Breaking Sets due to poor implementation

```
public class Person {

    private final String name;
    private final int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```

    }

    @Override
    public int hashCode() {
        return name.hashCode() * age;
    }
}

@Test
public void wrongSetBehavior() {
    final Set<Person> people = new HashSet<>();

    final Person person1 = new Person("Alice", 28);
    final Person person2 = new Person("Bob", 30);
    final Person person3 = new Person("Charlie", 22);

    final boolean person1Added = people.add(person1);
    final boolean person2Added = people.add(person2);
    final boolean person3Added = people.add(person3);

    assertTrue(person1Added && person2Added && person3Added);

    // logically equal to person1
    final Person person1Again = new Person("Alice", 28);
    // should return false, as Alice is already in the set
    final boolean person1AgainAdded = people.add(person1Again);

    // But will return true as the equals method has not been implemented
    assertTrue(person1AgainAdded); assertEquals(4, people.size());
}

```

Given the relationship between `hashCode` and `equals`, the golden rule is: *Whenever overriding `hashCode` or `equals`, you MUST override both methods.* In fact, IDEs such as IntelliJ or Eclipse do not allow you to auto-generate one method without the other.

Even though equal objects must return the same `hashCode`, the reverse is not true. Listing 8-12 is a legal implementation of `hashCode`.

LISTING 8-12: Overriding `hashCode()`

```

@Override
public int hashCode() {
    return 42;
}

```

This approach, however, is not advisable. As you now know, `HashMap`s (and therefore `HashSet`s, which are backed by `HashMap`s) use the `hashCode` method as an index to the table storing the references. Should unequal objects have the same `hashCode`, then both are stored at that index, using a `LinkedList` data structure.

If you were to have every object return the same `hashCode`, and those are stored in a `HashMap`, then the performance of your `HashMap` has degraded to that of a linked list: All values will be stored in

the same row of the `HashMap` implementation. Any search for a particular value will traverse all the values, testing each one by one for equality.

JAVA'S ARRAYS

How are arrays represented in Java?

The most important thing to remember about arrays in Java is that they are objects. They can be treated as objects. It is possible (though not very useful) to call `toString()` on arrays, or use them in a polymorphic sense, such as holding arrays in a container of `Objects`, illustrated in Listing 8-13.

LISTING 8-13: Treating arrays as objects (I)

```
@Test
public void arraysAsObjects() {
    Map<String, Object> mapping = new HashMap<>();
    mapping.put("key", new int[] {0, 1, 2, 3, 4, 5});

    assertTrue(mapping.get("key") instanceof (int[]));
}
```

Because arrays are objects, this means arrays are passed by reference, so anything holding a reference to that array can mutate it in some way. Listing 8-14 demonstrates this. This can be a cause of great confusion, especially when the references are being mutated in completely separate areas of the code, or even different threads.

LISTING 8-14: Treating arrays as objects (II)

```
@Test
public void arrayReferences() {
    final int[] myArray = new int[] {0, 1, 2, 3, 4, 5};
    int[] arrayReference2 = myArray

    arrayReference2[5] = 99;

    assertFalse(myArray[5] == 5);
}
```

WORKING WITH STRINGS

How are strings held in memory?

One of the great advantages to working with Java is that all of the library implementations are available to view and inspect. Looking at the implementation of `String` in Java 1.7 in Listing 8-15 is not a great surprise. The value represented by the `String` is held as an array of `chars`.

LISTING 8-15: String definition

```
public final class String implements
    java.io.Serializable,
    Comparable<String>,
    CharSequence {

    private final char[] value;
    ...
}
```

The `String` class is so core to the Java language, and is so widely used, that although it is simply a defined class in a Java library, the JVM and the compiler treats `Strings` in special ways under certain circumstances. `String` can almost be treated as a primitive type.

When creating a `String` literal, it is not necessary, or even advisable, to call `new`. At compile time, `String` literals—that is, any characters between a pair of quotations—will be created as a `String`.

The two `Strings` created in Listing 8-16 are the same, and can be treated as the same value in any running program. Looking at the construction of `helloString2` first, when the compiler sees the sequence of characters `" , H , e , . . . , ! , "`, it knows that this is to create a `String` literal of the value enclosed in the quotes.

LISTING 8-16: String creation

```
@Test
public void stringCreation() {
    String helloString1 = new String("Hello World!");
    String helloString2 = "Hello World!";

    assertEquals(helloString1, helloString2);
}
```

When `helloString1` sees the characters in quotes, it also makes a `String` object for that value. This `String` literal is enclosed in a constructor, so that is passed to whatever the constructor does. As you will see, `Strings` are immutable, so this constructor can take a copy of the value passed. But the `String(char[])` constructor will make a full copy of that array. Most IDEs will generate a warning when attempting to pass a `String` literal to a `String` constructor—it is just not necessary.

Is it possible to change the value of a `string`?

On inspection of the methods on `String`, all the methods that appear to mutate the `String` actually return a `String` instance themselves.

Although the code in Listing 8-17 isn't exactly surprising, it does highlight the important behavior of `Strings`: The value of the `String` can never change. It is immutable.

LISTING 8-17: Changing String values

```
@Test
public void stringChanges() {
    final String greeting = "Good Morning, Dave";
    final String substring = greeting.substring(4);

    assertTrue(substring.equals("Good"));
    assertFalse(greeting.equals(substring));
    assertTrue(greeting.equals("Good Morning, Dave"));
}
```

The listing shows that the “mutable” methods, such as `substring` (used in the listing), or others like `replace` or `split`, will always return a new copy of the string, changed appropriately.

Safe in the knowledge that the value represented by an instance can never change, immutability gives many advantages. Immutable objects are thread-safe: They can be used in many concurrent threads, and each thread is confident that the value will never change. No locking or complex thread coordination is necessary.

`String` is not the only immutable class in the standard Java library. All of the number classes, such as `Integer`, `Double`, `Character`, and `BigInteger`, are immutable.

What is interning?

Following the `String` literal and immutable discussions, literals get even more help and special treatment from the JVM at run time. When the class is loaded by the JVM, it holds all the literals in a constants pool. Any repetition of a `String` literal can be referenced from the same constant in the pool. This is known as `String` interning.

Naturally, because the constants in this pool can be referenced from any running class in the JVM, which could easily run into the thousands, the immutable nature of `Strings` is an absolute necessity.

The `String` intern pool is not just open to compile-time `String` literals; any `String` instance can be added to this pool with the `intern()` method. One valid use for `String` interning could be when parsing a large amount of data from a file or a network connection, and that data could contain many duplicates. Imagine something like a large banking statement, which could contain many credits or debits from a single party, or many similar actions such as multiple purchases of the same product over time. If these entities are interned when they are read, there will only be one unique instance taking up memory inside the JVM. The `equals` method on the `String` class also checks if the two compared instances are the same reference in memory, which makes the comparison much faster if they are the same.

Be aware that the `intern()` method does not come for free: Although the `Strings` are not stored on the heap, they do need to be stored somewhere, and that's the PermGen space (covered in Chapter 10).

Also, liberal use of `intern()`, if the constant pool had, say, several million entries, can affect the run-time performance of the application due to the cost of lookups for each entry.

The `String` constant pool is an implementation of the Flyweight Pattern, and similar occurrences of this pattern exist within the standard Java libraries. For instance, Listing 8-18 illustrates that the method `Integer.valueOf(String)` will return the same instance of the `Integer` object for values between `-128` and `127`.

LISTING 8-18: The Flyweight Pattern in the Java libraries

```
@Test
public void intEquality() {
    // Explicitly calling new String so the instances
    // are in different memory locations
    final Integer int1 = Integer.valueOf(new String("100"));
    final Integer int2 = Integer.valueOf(new String("100"));

    assertTrue(int1 == int2);
}
```

By using an equality check, this code makes sure these objects refer to the same instance in memory. (JUnit's `Assert.assertSame()` could do that here, but `assertTrue()` with `==` makes this absolutely explicit.)

UNDERSTANDING GENERICS

Explain how to use generics with the Collections API.

Generics are also known as *parameterized types*. When you use generics with the collections classes, the compiler is being informed to restrict the collection to allow only certain types to be contained.

The code in Listing 8-19 is perfectly legal: It's adding `Strings` to a `List` instance, and accessing `Strings` from that same instance. When calling `get`, the object needs to be cast to a `String`. The `get` method returns an `Object`, because `List` is polymorphic; the `List` class needs to be able to deal with any type possible.

LISTING 8-19: Use of a list without generics

```
private List authors;

private class Author {
    private final String name;
    private Author(final String name) { this.name = name; }
    public String getName() { return name; }
}

@Before
```

continues

LISTING 8-19 *(continued)*

```
public void createAuthors() {
    authors = new ArrayList();

    authors.add(new Author("Stephen Hawking"));
    authors.add(new Author("Edgar Allan Poe"));
    authors.add(new Author("William Shakespeare"));
}

@Test
public void authorListAccess() {
    final Author author = (Author) authors.get(2);
    assertEquals("William Shakespeare", author.getName());
}
```

Because the list is not constrained to allow only instances of the `Author` class, the code in Listing 8-20 is allowed, possibly a simple mistake for a developer to make.

LISTING 8-20: Erroneously using a list without generics

```
@Test
public void useStrings() {
    authors.add("J. K. Rowling");

    final String authorAsString = (String) authors.get(authors.size() - 1);
    assertEquals("J. K. Rowling", authorAsString);
}
```

There is no constraint on the type of instance used in the list. You can probably spot several issues with the code. Is it an absolute certainty that only these types will be in the list? If not, then a `ClassCastException` will be thrown at run time. The cast itself is messy, boilerplate code.

By using generics, any potential exceptions at run time are moved to actual compiler errors. These are spotted much earlier in the development life cycle, which, in turn, leads to faster fixes and cleaner code. Listing 8-21 shows this.

LISTING 8-21: Using generics

```
private List<Author> authors;

private class Author {
    private final String name;
    private Author(final String name) { this.name = name; }
    public String getName() { return name; }
}

@Before
public void createAuthors() {
```

```

    authors = new ArrayList<>();

    authors.add(new Author("Stephen Hawking"));
    authors.add(new Author("Edgar Allan Poe"));
    authors.add(new Author("William Shakespeare"));
}

@Test
public void authorListAccess() {
    final Author author = authors.get(2);
    assertEquals("William Shakespeare", author.getName());
}

```

The cast for access has disappeared, and reading this code feels more natural: “Get the third author from the list of authors.”

The `authors` instance is constrained to take only objects of type `Author`. The other test, `useStrings`, now doesn’t even compile. The compiler has noticed the mistake. Making this test use an `Author` class rather than a `String` is an easy fix.

All of the classes in the collections API make use of generics. As you have seen, the `List` interface and its implementations take one type parameter. The same goes for `Sets`. As you should expect, `Maps` take two type parameters: one for the key and one for the value.

Generic types can be nested, too. It is legal for a `Map` to be defined as `HashMap<Integer, List<String>>`: a `HashMap` that has an `Integer` as its key, mapped to `Lists` of type `String`.

Amend the given Stack API to use generics.

Imagine Listing 8-22 was required to use generics.

LISTING 8-22: An API for stack manipulation

```

public class Stack {

    private final List values;

    public Stack() {
        values = new LinkedList();
    }

    public void push(final Object object) {
        values.add(0, object);
    }

    public Object pop() {
        if (values.size() == 0) {
            return null;
        }
    }
}

```

continues

LISTING 8-22 *(continued)*

```

    }

    return values.remove(0);
}

```

Implementations of a stack are very popular in interviews. They have a relatively small set of operations, usually `push`, `pop`, and perhaps `peek`. Stacks are easily implemented using the Java collections API; no extra libraries are necessary. Take a moment to make sure you understand why a `LinkedList` is used here—would an `ArrayList` have worked too? What is the difference?

For the given implementation, this is a fully functioning stack, although it has not been developed for use with generics. It suffers from all the issues discussed in the previous question.

To migrate this class to use generics, you can use the compiler as your guide. First, the `Stack` class must be declared to take a parameterized type:

```

public class GenericStack<E> {
    ...

```

`E` is the parameterized type variable. Any symbol could have been used. `E` here stands for “element,” mirroring its use in the collections API.

The contained `List` can now use this generic type:

```

    private final List<E> values;

```

This immediately throws up a compiler error. The `push` method takes `Objects`, but now it needs to take objects of type `E`:

```

    public void push(final E element) {
        values.add(0, element);
    }

```

Changing `values` to `List<E>` also threw up a compiler warning in the `Stack` constructor. When creating the `values` `LinkedList`, this should be parameterized too:

```

    public GenericStack() {
        values = new LinkedList<E>();
    }

```

One change the compiler was not aware of is a change to the `pop` method. The last line of the `pop` method, `values.remove(0)`, now returns a value of type `E`. Currently, this method returns `Object`. The compiler has no reason to error or even to throw up a warning, because `Object` is a supertype of all classes, regardless of what `E` is. This can be changed to return the type `E`:

```

    public E pop() {
        if (values.size() == 0) {
            return null;
        }
    }

```

```

    }

    return values.remove(0);
}

```

How does type variance affect generics?

Given the following class hierarchy:

```

class A {}
class B extends A {}

```

B is a subtype of A. But `List` is not a subtype of `List<A>`. Known as *covariance*, Java's generics system has no way to model this.

When dealing with generic types, at certain times you may want to accept subtypes of a class. Using the `GenericStack` class from the previous question, imagine a utility method, which creates a new `GenericStack` from a `List` of A:

```

public static GenericStack<A> pushAllA(final List<A> listOfA) {
    final GenericStack<A> stack = new GenericStack<>();
    for (A a : listOfA) {
        stack.push(a);
    }

    return stack;
}

```

This compiles and works exactly as expected for `Lists` of A:

```

@Test
public void usePushAllA() {
    final ArrayList<A> list = new ArrayList<>();
    for (int i = 0; i < 10; i++) {
        list.add(new A());
    }

    final GenericStack<A> genericStack = pushAllA(list);

    assertNotNull(genericStack.pop());
}

```

But when trying to add a `List` of B, it doesn't compile:

```

@Test
public void usePushAllAWithBs() {
    final ArrayList<B> listOfBs = new ArrayList<>();
    for(int i = 0; i < 10; i++) {
        listOfBs.add(new B());
    }
}

```

```

        final GenericStack<A> genericStack = pushAllA(listOfBs);

        assertNotNull(genericStack.pop());
    }

```

Although `B` is a subclass of `A`, `List` is not a subtype of `List<A>`. The method signature of `pushAllA` needs to change to explicitly allow `A`, *and any subclass of A*:

```

public static GenericStack<A> pushAllA(final List<? extends A> listOfA) {

```

The question mark is called a *wildcard*, and it is telling the compiler to allow any instance of a class extending `A`.

What does reified mean?

Essentially, being *reified* means being available at run time. Java's generic types are not reified. This means all the type information the compiler uses to check that any implementing code is using generic parameters correctly is not part of the `.class` file definition.

Listing 8-23 makes use of generics.

LISTING 8-23: A simple use of generics

```

import java.util.ArrayList;
import java.util.List;

public class SimpleRefiedExample {

    public void genericTypesCheck() {
        List<String> strings = new ArrayList<>();
        strings.add("Die Hard 4.0");
        strings.add("Terminator 3");
        strings.add("Under Siege 2");

        System.out.println(strings.get(2) instanceof String);
    }
}

```

Decompiling the class file using JAD (a Java decompiler) produces the code in Listing 8-24.

LISTING 8-24: Decompiled code from Listing 8-23

```

import java.io.PrintStream;
import java.util.ArrayList;
import java.util.List;

public class SimpleRefiedExample

```



```

{
    public void genericTypesCheck() {
        ArrayList arraylist = new ArrayList();
        arraylist.add("Die Hard 4.0");
        arraylist.add("Terminator 3");
        arraylist.add("Under Siege 2");
        System.out.println(arraylist.get(2) instanceof String);
    }
}

```

All type information has been lost. This means that if you were to pull in this list from a class file that had already been compiled, you must assume that it respects the generic parameter information it was compiled with.

You can see from this example that it would be quite simple to construct a small use case where an instance of a `List` with a generic type is used, but the dynamically loaded instances are of a different type.

AN INTERESTING POINT ABOUT LISTING 8-24

It is also worth noting that, as expected, the compiler has produced a default constructor because the code did not provide one. All classes must have at least one constructor.

AUTOBOXING AND UNBOXING

Is it possible for access to a primitive type to throw a `NullPointerException`?

The correct answer to this question is no, but it can certainly appear so. Take Listing 8-25; there are several points of note here. When assigning `intObject` to `42`, you are attempting to assign the primitive value `42` to an object. This shouldn't be legal, and before Java 5, this certainly would have produced a compiler error. However, the compiler is using a technique called *autoboxing*. The compiler knows that the reference type equivalent of `int` is `Integer`, and so this is allowed. If the compiler was not aware that these two types were related (as was the case before Java 5), and if a reference to an `Integer` object was needed, then you would need to manage this manually, either using `new Integer(42)` or, more efficiently, `Integer.valueOf(42)`.

The reason `Integer.valueOf(42)` is more efficient is because small values are cached. This was discussed in Chapter 6 on Design Patterns and is known as the Flyweight Pattern.

LISTING 8-25: Demonstration of autoboxing and unboxing

```

@Test
public void primitiveNullPointer() {
    final Integer intObject = 42;
    assert(intObject == 42);

    try {
        final int newIntValue = methodWhichMayReturnNull(intObject);
        fail("Assignment of null to primitive should throw NPE");
    } catch (NullPointerException e) {
        // do nothing, test passed
    }
}

private Integer methodWhichMayReturnNull(Integer intValue) {
    return null;
}

```

THE DIFFERENCE BETWEEN AUTOBOXING AND BOXING

Java 5 introduced the notion of autoboxing. That is, the automatic transformation from a primitive type to its reference type, such as `boolean` to `Boolean`, or `int` to `Integer`.

Before Java 5, you had to perform this operation manually, known as *boxing*. To turn an `int` into an `Integer`, you would have had to construct one, (`new Integer(42)`), or use the factory method (`Integer.valueOf(42)`).

The process of converting from a boxed reference type, such as `Float`, `Integer`, or `Boolean` to their primitive counterparts, `float`, `int`, or `boolean` is called *unboxing*. Again, this is an operation the compiler provides. However, as you will be very aware by now, whenever working with reference types, you must always be prepared for a reference being `null`, and this is true for the boxed types. When the compiler converts from `Integer` to `int`, it assumes that the value is not `null`, so should it be `null`, this will immediately throw a `NullPointerException`. The initial line of code in the `try` block assigns the return type of the method, `Integer` to an `int` primitive type. In this case, the method always returns `null`, so when this line is run, a `NullPointerException` is thrown, because it is not possible to assign `null` to a primitive type.

This can cause frustration and confusion when trying to debug or fix issues, especially when it is not explicitly clear that boxing and unboxing is happening.

Primitive types cannot be used in generic type definitions—you cannot have a `List<int>`. In this case, the reference type should be used.

USING ANNOTATIONS

Give an example of a use of annotations.

Annotations were introduced in Java 5, and the JUnit library made full use of these annotations following the JUnit 4 release.

Listing 8-26 shows how JUnit tests worked prior to JUnit 4. Test suites were written using specific conventions for naming tests, and also steps to run before and after the tests were run. The JUnit runner would inspect any subclass of `TestCase` compiled tests, using reflection. If it found the method signature `public void setUp()`, this would be run before each test, and `public void tearDown()` would be run after. Any public methods with a void return type, and the method name starting with `test`, would be treated as a test.

LISTING 8-26: Anatomy of a JUnit 3 test

```
public class JUnit3Example extends TestCase {

    private int myInt;

    public void setUp() {
        myInt = 42;
    }

    public void testMyIntValue() {
        assertEquals(42, myInt);
    }

    public void tearDown() {
        myInt = -1;
    }
}
```

Mistyping the prefix `test`, or `setUp`, or `tearDown` would result in the tests not running, without any indication why; or worse, the tests would fail.

With the introduction of annotations, this brittle naming convention could be tossed aside, and instead, use annotations to mark the appropriate methods, allowing tests to be more expressive in their definition. Listing 8-27 shows the same test using annotations.

LISTING 8-27: Anatomy of a JUnit 4 test

```
public class Junit4Example {

    private int myInt;

    @Before
```

continues

LISTING 8-27 *(continued)*

```

    public void assignIntValue() {
        myInt = 42;
    }

    @Test
    public void checkIntValueIsCorrect() {
        Assert.assertEquals(42, myInt);
    }

    @After
    public void unsetIntValue() {
        myInt = -1;
    }
}

```

Even if you have no experience with JUnit, it should be quite apparent that using annotations makes for better tests: Methods can have better names, and methods can have multiple annotations. Feasibly, one method could have a `@Before` and `@After` annotation on it, eliminating code duplication. For more details on JUnit, see Chapter 9.

What does the `@Override` annotation give us?

The `@Override` annotation is an extremely useful compile-time check. It is an instruction to the compiler that a superclass method is being overridden. If there is no matching method signature on any superclass, this is an error and should not compile.

In general, it is a great way to make sure you don't make mistakes when overriding methods.

```

public class Vehicle {

    public String getName() {
        return "GENERIC VEHICLE";
    }
}

public class Car extends Vehicle {

    public String getname() {
        return "CAR";
    }
}

```

It appears that `getName()` has been overridden to return something slightly more descriptive, but the code:

```

Car c = new Car();
System.out.println(c.getName());

```

will actually print `GENERIC VEHICLE`. The method in class `Car` has a typo: It is `getname()` rather than `getName()`. All identifiers in Java are case-sensitive. Trying to track down bugs like this can be extremely painful.

Annotating the attempt at overriding `getName()` with `@Override` would have flagged a compiler error. This could have been fixed immediately.

As always, you should be using a professional-grade IDE, and leaning on it for as much help as possible. Both IntelliJ and Eclipse provide wizard dialog boxes for overriding methods, allowing you to pick exactly which method you want to override. These wizards will add the `@Override` annotation for you, making it very hard to get this wrong.

NAMING CONVENTIONS

When interviewers look at sample code, whether from a test or a candidate's sample code, perhaps from Github or a similar site, they often look for the quality of the code, from the use of the `final` keyword to the conventions used in variable naming. IDE support for naming conventions is excellent, and there are few, if any, reasons not to follow these conventions. They make the code much more legible to a colleague or interviewer.

Classes

Classes always start with a capital letter, and are CamelCase:

- `Boolean`
- `AbstractJUnit4SpringContextTests`
- `StringBuilder`

Variables and Methods

Variables and methods always start with a lowercase letter, and are camelCase too:

- `int myInteger = 56;`
- `public String toString();`

Constants

`static final` instance variables can never change, and are therefore constant. The convention for constants is to be all uppercase, with any breaks between words represented by an underscore:

- `public static final int HTTP_OK = 200;`
- `private static final String EXPECTED_TEST_RESPONSE = "YES";`

One element of debate is how to represent acronyms properly with camel casing. The most accepted variant is to capitalize the first letter, and lowercase the rest, such as `HttpClient` or `getXmlStream()`.

This isn't even consistent within the Java's own standard libraries. For example, `URLConnection` uses both styles in one class name!

It is this author's opinion that the capitalized first letter approach is favorable: It is much easier to see where the acronym ends and the next word begins. With many IDEs nowadays, it is possible to search for classes merely by entering the first letter of each word in the camel case.

HANDLING EXCEPTIONS

Describe the core classes in Java's exception hierarchy.

Figure 8-1 illustrates the exception hierarchy, with examples of each type.

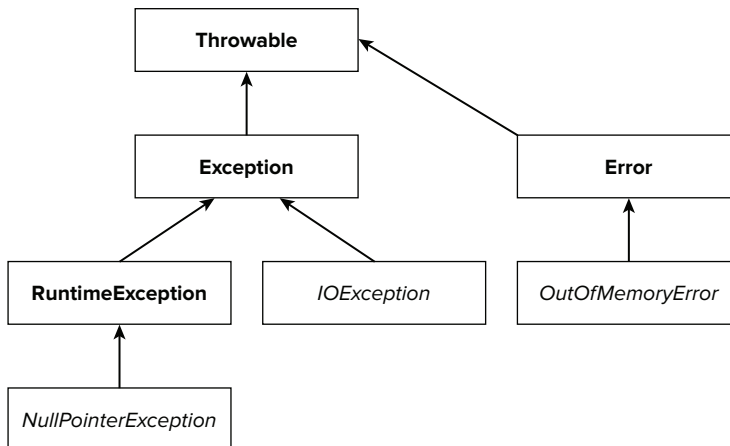


FIGURE 8-1

Any class that can be thrown extends from `Throwable`. `Throwable` has two direct subclasses: `Error` and `Exception`. As a rule, it is usually the responsibility of the programmer to address and fix `Exceptions` where necessary. `Errors` are not something you can recover from, such as an `OutOfMemoryError` or a `NoClassDefFoundError`.

Exceptions themselves are split into two definitions: An exception is either a runtime exception or a checked exception. A runtime exception is any subclass of `RuntimeException`. A checked exception is any other exception.

When using a method (or constructor) that may throw a checked exception, that exception must be explicitly defined on the method definition. Any caller to the code must be prepared to handle that

exception, by either throwing it back to any caller of that method, or surrounding the method call in a `try/catch/finally` block, and deal with the exceptional case appropriately.

Are runtime exceptions or checked exceptions better?

Another area of great debate is whether checked or runtime exceptions are better. Arguments certainly exist for both sides.

With checked exceptions, you are being explicit to any user of your code about what could possibly go wrong. When writing a method that may throw a checked exception, be as explicit as possible. A method definition such as `public String getHostName() throws Exception` does not give much clue to what may go wrong. If it were defined as `public String getHostName() throws UnknownHostException`, the client code would know what may go wrong, and it also would give some more insight as to how exactly the method works.

With runtime exceptions, defining the exception on the method, rethrowing the exception, or using a `try/catch/finally` block are all optional. As a general rule, `RuntimeExceptions` are exceptions that a diligent programmer should already have mitigated: problems such as accessing an array index value larger than the size of the array—`ArrayIndexOutOfBoundsException`—or trying to call methods on a null reference—`NullPointerException`.

When defining an API and deciding whether to use runtime or checked exceptions, this author's opinion is to favor runtime exceptions and be explicit in any documentation as to what exactly may be thrown to any clients calling that method. Using `try/catch/finally` blocks adds a significant amount of boilerplate code to even the simplest of method calls, and makes life very difficult for any future maintainer of the code. Just examine any non-trivial use of JDBC, and you will often see `try/finally` blocks inside `try/catch/finally` blocks. Modern languages such as Scala have stepped away from checked exceptions, and have only runtime exceptions.

What is exception chaining?

When catching an exception to deal with an error case, it is entirely possible you would want to rethrow that exception, or even throw an exception of a different type.

Reasons to do this include “laundering” an exception from a checked exception to a runtime one, or perhaps performing some logging of the exception and then rethrowing it. When throwing a previously caught exception, it is advisable to throw a new exception and add a reference on that new exception. This technique is called *exception chaining*.

This mindset is the same for throwing a new exception within a catch block. Add a reference to the old exception in the new exception's constructor.

The reason for doing this is that these exception chains are extremely valuable for debugging should the exception not be handled at all, and the stack trace make it all the way to the application console.

The chains are referenced in the stack trace by a “caused by” line. This refers to the original exception prior to being wrapped in a new exception or rethrown.

Listing 8-28 shows how to create a chained exception. The exception instance in the catch block is chained to the new `IllegalStateException` instance, by passing that reference to the constructor of the `IllegalStateException`.

LISTING 8-28: Handling chained exceptions

```
public class Exceptions {

    private int addNumbers(int first, int second) {
        if(first > 42) {
            throw new IllegalArgumentException("First parameter must be small");
        }

        return first + second;
    }

    @Test
    public void exceptionChaining() {
        int total = 0;

        try {
            total = addNumbers(100, 25);
            System.out.println("total = " + total);
        } catch (IllegalArgumentException e) {
            throw new IllegalStateException("Unable to add numbers together", e);
        }
    }
}
```

All standard Java library exceptions can take a `throwable` as a constructor parameter, and should you ever create any new `Exception` classes, make sure you adhere to this.

When this (admittedly contrived) test is run, it demonstrates that the `IllegalStateException` was indeed caused by the first parameter to `addNumbers` being too high:

```
java.lang.IllegalStateException: Unable to add numbers together
    at com.wiley.acinginterview.chapter08.Exceptions.exceptionChaining(Exceptions.java:23)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    ...
<Exeption truncated>
...
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    at com.intellij.rt.execution.application.AppMain.main(AppMain.java:120)
Caused by: java.lang.IllegalArgumentException: First parameter must be small
    at com.wiley.acinginterview.chapter08.Exceptions.addNumbers(Exceptions.java:9)
    at com.wiley.acinginterview.chapter08.Exceptions.exceptionChaining(Exceptions.java:20)
    ... 26 more
```


Had the exceptions not been chained, and the code inside the `try` block much larger, trying to work out where the original exception had been thrown from could have been a time-wasting chore.

If you ever see an exception being thrown in a catch block, and it does not chain the original exception, add a reference there and to that original exception. You'll thank yourself later!

What is a try-with-resources statement?

Java 7 introduced some syntactic sugar for the `try/catch/finally` statement. If a class implements the `AutoCloseable` interface, you do not need to worry about closing any resources, which is shown in Listing 8-29.

LISTING 8-29: try-with-resources

```
@Test
public void demonstrateResourceHandling() {
    try(final FileReader reader = new FileReader("/tmp/dataFile")) {
        final char[] buffer = new char[128];
        reader.read(buffer);
    } catch (IOException e) {
        // .. deal with exception
    }
}
```

Before try-with-resources was introduced, the reader instance would have had to be explicitly closed, which itself could have caused an exception, or unspecified behavior if the close failed, or even if the close was forgotten altogether.

The `AutoCloseable` interface specifies one method, `close()`, and this is called after the `try` block, as if it were in the `finally` part of the block.

USING THE STANDARD JAVA LIBRARY

The APIs available as part of the standard Java library cover a wide range of domains, from database access to optimized sorting and searching algorithms, from concurrency APIs to two user interface frameworks.

Why do fields that are private also need to be marked as final to make them immutable?

If you had a final class with no accessible setters, and all fields were private, you could be inclined to think that the class is immutable, such as that in Listing 8-30.

LISTING 8-30: An almost-immutable class

```
public final class BookRecord {  
  
    private String author;  
    private String bookTitle;  
  
    public BookRecord(String author, String bookTitle) {  
        this.author = author;  
        this.bookTitle = bookTitle;  
    }  
  
    public String getAuthor() {  
        return author;  
    }  
  
    public String getBookTitle() {  
        return bookTitle;  
    }  
}
```

Unfortunately this is not the case. The fields can still be manipulated using the Reflection API. Reflection has the ability to access and mutate all fields, regardless of their visibility. The `final` modifier instructs the JVM that no modifications are allowed on that field at all. Though it may seem slightly underhanded to allow access to state that has genuinely been marked as inaccessible to the outside world, there actually are a few legitimate cases. With Spring's Inversion of Control container, private fields with an `@Autowired` annotation have these fields set when the container is initializing at run time.

Listing 8-31 shows how you can mutate private fields from outside of a class definition.

LISTING 8-31: Mutating private fields with the Reflection API

```
@Test  
public void mutateBookRecordState() throws NoSuchFieldException,  
                                           IllegalAccessException {  
    final BookRecord record = new BookRecord("Suzanne Collins",  
                                              "The Hunger Games");  
  
    final Field author = record.getClass().getDeclaredField("author");  
    author.setAccessible(true);  
    author.set(record, "Catching Fire");  
  
    assertEquals("Catching Fire", record.getAuthor());  
}
```

Unless you are writing your own framework, very few reasons exist to change the value of a private field, especially in classes you own or that are under your control.

Which classes do all other collections API classes inherit from?

The framework can be broadly split into three main definitions: `Sets`, `Lists`, and `Maps`. Take another look at Chapter 5 if you need a refresher on how these collections work. There is also the specialized interface of `Queue`, which provides simple operations for adding, removing and examining queued elements. The `ArrayList` class is both a `List` and a `Queue`. The Java Collections framework is contained in the package `java.util`. All single-element collections implement the `Collection` interface, which specifies some very broad methods, such as `clear()` for removing all elements and `size()` for determining how many entries are in the collection. `Map` implementations do not implement `Collection`. Instead, Java prefers a distinction between mappings and collections. Of course, `Collections` and `Maps` are intricately linked: the `Map` interface contains the methods `entrySet()`, `keySet()`, and `values`, giving access to the `Map` components as `Collections`.

Figure 8-2 shows the collections hierarchy, and where some common implementations fit in.

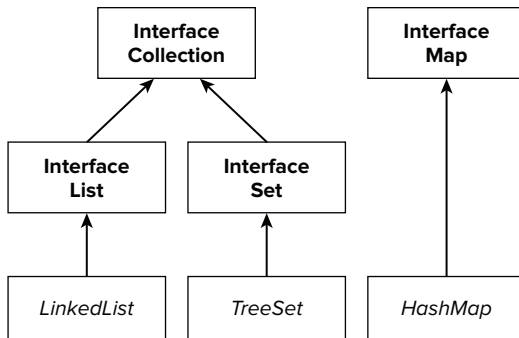


FIGURE 8-2

What is a `LinkedHashMap`?

A `LinkedHashMap` has a confusing name—is it a `HashMap`? Linked in what way?

A `LinkedHashMap` has all the properties of a `HashMap`—that is, quick lookup on a key index—but it also preserves the order of entry into the map. Listing 8-32 demonstrates how to use a `LinkedHashMap`.

LISTING 8-32: Usage of a `LinkedHashMap`

```

@Test
public void showLinkedHashMapProperties() {
    final LinkedHashMap<Integer, String> linkedHashMap = new LinkedHashMap<>();

    linkedHashMap.put(10, "ten");
  
```

continues

LISTING 8-32 *(continued)*

```

    linkedHashMap.put(20, "twenty");
    linkedHashMap.put(30, "thirty");
    linkedHashMap.put(40, "forty");
    linkedHashMap.put(50, "fifty");

    // works like a map
    assertEquals("fifty", linkedHashMap.get(50));

    // Respects insertion order
    final Iterator<Integer> keyIterator = linkedHashMap.keySet().iterator();

    assertEquals("ten",    linkedHashMap.get(keyIterator.next()));
    assertEquals("twenty", linkedHashMap.get(keyIterator.next()));
    assertEquals("thirty", linkedHashMap.get(keyIterator.next()));
    assertEquals("forty",  linkedHashMap.get(keyIterator.next()));
    assertEquals("fifty",  linkedHashMap.get(keyIterator.next()));

    // The same is not true for HashMap
    final HashMap<Integer, String> regularHashMap = new HashMap<>();
    regularHashMap.put(10, "ten");
    regularHashMap.put(20, "twenty");
    regularHashMap.put(30, "thirty");
    regularHashMap.put(40, "forty");
    regularHashMap.put(50, "fifty");

    final ArrayList hashMapValues = new ArrayList<>(regularHashMap.values());
    final ArrayList linkedHashMapValues = new ArrayList<>(linkedHashMap.values());

    // the lists will have the same values, but in a different order
    assertFalse(linkedHashMapValues.equals(hashMapValues));
}

```

Why was `HashMap` introduced when `Hashtable` already existed?

The Collections framework and the collections interfaces (`Collection`, `Map`, `List`, and so on) first appeared in the 1.2 release, but the `Hashtable` class, and the list-equivalent `Vector`, has been part of Java since its initial release. These classes were not written with a framework in mind. When the Collections framework was included, `Hashtable` was retrofitted to match the `Collection` interface.

The `Hashtable` class is synchronized, which, though effective for parallel work, provides a significant performance hit due to this overhead for any single-threaded work. `HashMap` is not synchronized, so the developer has the power to tailor the use of this class to any particular concurrent needs. It is advisable, for any serious use of a `Map` interface in a parallel environment, to make use of `ConcurrentHashMap` (introduced in Java 5). See Chapter 11 for more details.

LOOKING FORWARD TO JAVA 8

What new features are to be expected in Java 8?

At the time of writing, two key new features in Java 8 are default implementations on interfaces, and lambdas.

It is now possible to define a default implementation on interfaces. Any implementer of an interface does not need to provide an implementation if the default is sufficient, such as in Listing 8-33.

LISTING 8-33: Java 8 interfaces

```
public interface Java8Interface {
    void alpha();
    int beta default { return 6; };
    String omega final { return "Cannot override"; };
}
```

As well as a default implementation, interfaces can have a final implementation too. This now works in a very similar way to abstract classes.

The past few years have seen an explosion of new languages that are capable of being compiled into code that is runnable on the JVM. Some of these languages, such as Scala and Groovy, have *lambdas*—that is, functions that are treated as first-class citizens; they can be passed into other functions and methods as variables. Lambdas are anonymous functions, such as the example in Listing 8-34. A lambda can be used in any situation where an interface with only one method is required. They are written inline, in place of a full interface implementation. Note the new syntax: parameters, an arrow, and then the method body.

LISTING 8-34: Lambdas in Java 8

```
public void run() {
    List<Integer> ints = new ArrayList<>();
    ints.add(1);
    ints.add(2);
    ints.add(3);

    List<Integer> newInts = new ArrayList<>();
    ints.forEach(i -> { newInts.add(i+10); });

    for(int i = 0; i < ints.size(); i++) {
        assert ints.get(i) + 10 == newInts.get(i);
    }

    assert ints.size() == newInts.size();
    System.out.println("Validated");
}
```

The `List` interface, or more specifically, the `Iterable` interface, has had a new method added: `forEach`. Had a default implementation not been allowed on interfaces, this would have needed to be added to every implementation of `Iterable`—not only in the standard Java library, but all third-party code too!

The parameter to `forEach` is an instance of the new interface `Consumer`. This interface has one method, `accept`, which has a `void` return type, and it takes one parameter. Because there is only one method, a lambda can be used here. When `forEach` is called, the parameter to the method is an instance of `Consumer`, and the parameter to that method is the same type as that of the list. The list is iterated over, and each element is passed to the implementation of `Consumer`. `Consumer` returns `void`, so the lambda cannot return anything to the calling method. Here, the lambda takes each element, adds ten, and puts that value in a new `List` instance. The Java 4 `assert` keyword is being used here to check that, first, the original `List` has not been modified in any way; second, that the new `List` has the correct values in it; and third, that there are no extra elements in either `List`.

For another take on lambdas, see the appendix.

The most important thing to take away from this is that the Java landscape is always changing, and these implementation details may change before Java 8 is released. It is extremely important to keep on top of what is happening in the industry, not just with Java releases.

SUMMARY

Many of the questions and discussions covered in this chapter should be familiar to you, especially if you have a few years' experience working with Java and the JVM. Hopefully while reading this chapter you have experimented and run the code. If not, try it now. Change the variables, and see what effect it has on the tests. Make the tests pass again based on your changes.

It really is important that you understand this chapter; everything else in this book and in an interview situation will assume this is well understood, and will build on the concepts here.

The topics covered here are not exhaustive. While running these examples, if you have areas you don't understand, use the unit tests to try out hypotheses, and take advantage of many of the useful resources on the Internet to see if others have asked similar questions. Have discussions with your peers and colleagues—learning from those around you is often the most productive and useful.

If you are not using an IDE, now is the time to start. More and more interviewers today expect you to write real code on a real computer using real tools. If you already know how to use an IDE presented to you in an interview, this can help you with a lot of the heavy lifting: filling in getters and setters, providing practical `hashCode` and `equals`, to name but a few. But IDEs also provide great refactoring tools, which are extremely helpful for making large-scale changes in a time-pressured situation like a coding test. If IDEs are unfamiliar to you in an interview situation, you will lose a lot of time fighting against the tool to write your code.

Make sure you are familiar with the Collections framework. Open up their implementations in an IDE (most have the Java library source automatically loaded). Look at how common code is implemented—`hashCode` on the primitives' reference type equivalents (`Boolean.hashCode()` is

interesting); look at how `HashSet` is implemented in terms of `HashMap`. See how many object-oriented patterns you can find.

But most of all, do not treat this chapter, or this book, as a quick fix, how-to guide for beating a Java interview. By implementing best practices, and receiving peer review of your code and practices, you will learn and improve. There will be less chance of you falling foul to a tricky corner-case in an interview. Experience is key.

Chapter 9 focuses on JUnit, the unit test library. Writing unit tests can often be a way to stand out and impress an interviewer, as not every candidate, surprisingly, will write unit tests.



Testing with JUnit

Testing is the cornerstone of any professional software product. Testing itself comes in many forms, at many different levels. Software is tested as a whole, complete artifact, but when broken down, each of the individual components should be tested, and therefore be *testable*, too.

By thinking of testing at the start of the development process, the software as a whole becomes easier to test, both for any full testing teams, if you are lucky enough to have one, and also for any *automated* tests. Plenty of rich, full-featured, complete testing solutions are available for testing code written for Java. One of the most ubiquitous and well-understood libraries for this is JUnit. JUnit provides a lightweight and simple interface for creating tests on any level, including unit tests, integration tests, system tests, or something more exotic like user interface tests. JUnit itself is properly integrated into many build tools such as Ant and Maven, and the more up-to-date ones such as Gradle and SBT. These tools automatically stop builds and flag errors should any test fail.

By writing tests first and making sure the tests are fully integrated into a build means that the tests will run any time a build is created, such as before code is checked in, or when a release artifact is built. This, in turn, builds confidence that the system is working correctly. Any time a test fails for a seemingly unknown reason; this can often be due to new code introducing bugs elsewhere in the system. This is known as a *regression*.

Development using JUnit is helped by the fact that the library is well integrated in most IDEs, allowing single tests or even multiple suites to be run within a couple of clicks. This gives extremely quick feedback from small changes as to whether the test passes. The smaller the change made between test runs, the more confidence you can have that your change made a test pass, rather than some environmental change or even a change in a seemingly unrelated piece of code.

This chapter covers how to use JUnit for several areas of the development life cycle, from unit testing small, isolated pieces, to combining these pieces with integration tests, to testing full running systems using unit tests.

Unit tests are a great way to show intent in any kind of assessment. As you will see, the `JUnit Assert` class provides a great way to open a dialogue with anyone else reading the code, showing exactly the intent for writing a test and any expectation or assumption made.

What value do JUnit tests give?

JUnit tests are often used with the development approach known as *Test-Driven Development* (TDD). The process of TDD is to perform short, iterative loops: You write a test based on expectations and assertions about what your code should do. Considering you would not yet have written the code for these tests, they should not pass. You then write the code to make the tests pass. Once the test passes, you repeat the process for a new piece of functionality. If your tests completely cover a specification of what you are trying to achieve, then once all the tests pass, you have the confidence that your code works and is correct.

The tests do not need to be confined to making sure functionality is correct: You can specify non-functional requirements, and only pass tests once those requirements are met. Some examples could be making sure a server under load responds to requests within a certain time, or that certain security parameters are met.

For any reasonable-sized project, you will probably be using a build system, such as Maven or Ant. The tests you have written can, and should, be integrated into your build, so that any changes to the underlying code that break the tests means your build halts, and you must fix the code before your build works again. This saves you from introducing bugs once the code has been released. For Maven, the tests are automatically run as part of the build. For Ant, tests are integrated with a single command.

Confidence in your code being correct means you can put increased reliance in automated tools. For developing server-side architectures, you can move toward a continuous delivery model, where fully tested code is automatically released to a production server with no human intervention after check-in.

How are JUnit tests run?

As previously mentioned, JUnit is well integrated into many build tools, but it is still possible to invoke it manually on the command line.

In the JUnit library, the class `JUnitCore` contains the main method used for starting the tests from the command line. The arguments are a list of classes to test:

```
$ /path/to/java -cp /path/to/junit.jar:. [classes to test]
```

For Maven, which works with project layouts following a specific convention, simply running `mvn test` in the root directory of the project will find and run all tests in the project. Maven itself is highly configurable, and specifying only a dependency on the JUnit library will specifically use JUnit for testing. Chapter 19 describes how to set up Maven and declare project dependencies.

You can run Maven with a declaration of which tests to run. If you set the system property `test`, only the tests set in that property will be run:

```
mvn test -Dtest=SystemTest
```

The test parameter can take wildcards, so setting the system property to a value such as `-Dtest=*IntegrationTest` would run any test suffixed with `IntegrationTest`.

THE JUNIT TEST LIFE CYCLE

When you run a test suite, each test follows a prescribed set of steps. These steps can help you modularize your tests and reuse as much code as possible.

What happens when running a JUnit test?

A test suite is usually confined to a class. Before annotations arrived in JUnit 4, you would need your class to extend `TestSuite`.

You can define a method, or set of methods, to run once, before the whole suite is run. These may do some long-running computations, such as preparing the filesystem for the upcoming tests, or some kind of pretesting notification to a build server or similar.

To run some code once as the test suite is started, you specify a public static method that returns `void`, and annotate it with `@BeforeClass`. The method is static, so you do not have access to a fully constructed instance of the test suite class, such as the instance variables or instance methods.

Mirroring this annotation is `@AfterClass`. Methods with this annotation are run after all tests have completed.

As soon as the `@BeforeClass` annotated methods have completed successfully, the test runner performs the following steps for each test in the suite:

1. A new instance of the suite is constructed. As with all Java classes, any code in the constructor is run. Test suite classes may only declare a single constructor with no arguments.
2. Immediately following the object construction, any public methods with a `@Before` annotation and with a `void` return type are run. These usually set up anything common across all tests, such as mock objects or objects with state. Because this is run before each test, you can use this to return stateful objects to their correct state, or perhaps set the filesystem to a state expected for your test. Because both the constructor and `@Before` annotated methods are run before each test, you can do any test setup in either of these positions. The convention is to perform the setup in the `@Before` methods, to keep symmetry with the equivalent `@After` method.
3. The test is then run. Tests that are defined with the `@Test` annotation, are public, and again, have a `void` return type.
4. Following a successful, or unsuccessful, run of the test, the `@After` annotated (again, `public void`) methods are called. This will tidy up anything the test may have dirtied, such as a database or filesystem, or perhaps perform some post-test logging.

The order in which the `@Before`, `@After`, and `@Test` methods run is not guaranteed, so you cannot do some partial setup in one `@Before` method, and expect another `@Before` method written later in the source file to finish that setup. This is the heart of JUnit: Your tests should be independent and atomic.

Listing 9-1 shows all of the steps for a suite with two tests, using a counter to verify the order in which all the components are run.

LISTING 9-1: The life cycle of a JUnit test

```
public class JUnitLifecycle {

    private static int counter = 0;

    @BeforeClass
    public static void suiteSetup() {
        assertEquals(0, counter);
        counter++;
    }

    public JUnitLifecycle() {
        assertTrue(Arrays.asList(1, 5).contains(counter));
        counter++;
    }

    @Before
    public void prepareTest() {
        assertTrue(Arrays.asList(2, 6).contains(counter));
        counter++;
    }

    @Test
    public void performFirstTest() {
        assertTrue(Arrays.asList(3, 7).contains(counter));
        counter++;
    }

    @Test
    public void performSecondTest() {
        assertTrue(Arrays.asList(3, 7).contains(counter));
        counter++;
    }

    @After
    public void cleanupTest() {
        assertTrue(Arrays.asList(4, 8).contains(counter));
        counter++;
    }

    @AfterClass
    public static void suiteFinished() {
        assertEquals(9, counter);
    }
}
```

The counter used in the code is a static variable, because for this test suite to run, the test runner instantiates `JUnitLifecycle` twice, one for each `@Test`.

Any test methods annotated with `@Ignore` are ignored. The `@Ignore` annotation is often used for tests that are known to fail and would break a continuous integration build. This is often a sign of *code smell*: The code backed by the tests has changed, but the tests have not. Code smell is a symptom that usually points to a deeper problem. This can lead to code not covered by tests, and reduces the confidence of a correct codebase. Any `@Ignored` tests in code submitted for an interview would be treated with concern.

Should you ever find yourself with a justifiable reason for annotating some tests with `@Ignore`, be sure to include a comment with a date as to why the test is ignored, describing how and when this will be rectified.

You can also use the `@Ignore` annotation at the class level, instructing the JUnit runner to skip a whole suite of tests.

BEST PRACTICES FOR USING JUNIT

The near-ubiquity of JUnit nowadays and the expressiveness of the library give you great power to show your potential in an interview.

How do you verify that your tests are successful?

One of the core classes in the JUnit library is the `Assert` class. It contains many static methods for expressing an assumption, which then verifies that assumption is true. Some of the key methods and their function are:

- `assertEquals`—Two objects are equal according to their `equals` method.
- `assertTrue` and `assertFalse`—The given statement matches the Boolean expectation.
- `assertNotNull`—An object is not null.
- `assertArrayEquals`—The two arrays contain the same values, checking equality by `equals` if comparing `Object` arrays.

If the assertion does not hold, an exception is thrown. Unless that exception is expected, or caught, that exception will fail the JUnit test.

There is also the `fail` method, which you can use if your test has reached a failing state. Most of the `assertXXX` methods call `fail` when necessary.

Take a look at the `Assert` class, and see what other assertion methods exist and how you can make use of them.

Each of the `assertXXX` methods are overloaded in pairs, with an extra `String` parameter available:

```
public static void assertTrue(String message, boolean condition)
public static void assertTrue(boolean condition)
```

This `String` parameter is a message that is displayed when assertion fails. Listing 9-2 shows a simple example of its usage.

LISTING 9-2: Assertion with a failure message

```
@Test
public void assertionWithMessage() {
    final List<Integer> numbers = new ArrayList<>();
    numbers.add(1);

    assertTrue("The list is not empty", numbers.isEmpty());
}

junit.framework.AssertionFailedError: The list is not empty
```

Listing 9-3 shows what happens if you use the `assertTrue` method without a message string.

LISTING 9-3: Assertion without a failure message

```
@Test
public void assertionWithoutMessage() {
    final List<Integer> numbers = new ArrayList<>();
    numbers.add(1);

    assertTrue(numbers.isEmpty());
}

junit.framework.AssertionFailedError: null
```

The message is merely the statement `null`. In a larger test with several `assertTrue` or `assertFalse` assertions, this can often lead to confusion as to why exactly a certain assertion is failing. If you use `assertEquals` without a message, then you are notified as to the difference when comparing two unequal values, but the error message has no explanation.

There are very few situations in which you would not provide this message parameter; especially when your code is being assessed for an interview.

However, it is possible to go one better. Although these messages are only ever printed for failing situations, when casually reading the code, these can be read as *expectations* rather than failure explanations.

This confusion, admittedly wrong, can be mitigated with the language used for writing the test. If you use that error message to say what *should* happen, then the code reads well, and when it is displayed as an error message, it still makes sense. Take the assertion from Listing 9-2:

```
assertTrue("The list is not empty", numbers.isEmpty());
```

This can be casually seen as a contradiction. The first parameter says "The list is not empty", but the second parameter says `numbers.isEmpty`—these are opposites. Had this been written instead as

```
assertTrue("The numbers list should not be empty", numbers.isEmpty());
```

the message on the top of a failed assertion stack trace would still make sense, and the code would be clearer for anyone, whether it's a peer reviewing your code or an interviewer assessing a coding test.

For brevity, the convention of this book is to not include the failure message string parameter in any JUnit tests, because the intention should be clear from the discussion.

How can you expect certain exceptions?

If you are testing a failing situation in your code, and you expect an exception to occur, you can notify the test of your expected exception type. If that exception is thrown in the test, it passes. Completion of the test without that exception being thrown is a failure. Listing 9-4 shows this in action.

LISTING 9-4: Expecting exceptions

```
@Test(expected = NoSuchFileException.class)
public void expectException() throws IOException {
    Files.size(Paths.get("/tmp/non-existent-file.txt"));
}
```

The parameter to the `@Test` annotation provides an indication to the test runner that this test should throw an exception. The method still needs to declare `throws IOException` here because `IOException` is a checked exception.

Had the test in Listing 9-4 been longer than one line, and the expected exception had been something more general, such as `RuntimeException` or even `Throwable`, it would be very hard to differentiate between the expected exception and a problem elsewhere, such as the test environment, as Listing 9-5 shows.

LISTING 9-5: A poorly defined test for expecting exceptions

```
@Test(expected = Exception.class)
public void runTest() throws IOException {
    final Path fileSystemFile = Paths.get("/tmp/existent-file.txt");

    // incorrect usage of Paths.get
    final Path wrongFile = Paths.get("http://example.com/wrong-file");

    final long fileSize = Files.size(fileSystemFile);
    final long networkFileSize = Files.size(wrongFile);

    assertEquals(fileSize, networkFileSize);
}
```

Every single line in this test has the possibility for throwing an exception, including the assertion itself. In fact, if the assertion fails, it throws an exception, and therefore the test incorrectly passes!

It is advisable to use the `expected` parameter on the `@Test` annotation sparingly. The most reliable tests using this parameter have only one line in the method body: the line that should throw the exception. Therefore, it is crystal clear to see how and why a test could start failing, such as shown in Listing 9-4 earlier.

Of course, methods can throw exceptions, and you should still be able to test for those exceptions. Listing 9-6 details a clearer way to handle this. Imagine a utility method, `checkString`, which throws an exception if the `String` passed to the method is `null`.

LISTING 9-6: Explicitly expecting exceptions

```
@Test
public void testExceptionThrowingMethod() {
    final String validString = "ValidString";
    final String emptyValidString = "";
    final String invalidString = null;

    ParameterVerifications.checkString(validString);
    ParameterVerifications.checkString(emptyValidString);
    try {
        ParameterVerifications.checkString(invalidString);
        fail("Validation should throw exception for null String");
    } catch (ParameterVerificationException e) {
        // test passed
    }
}
```

This is explicitly testing which line should throw an exception. If the exception is not thrown, the next line is run, which is the `fail` method. If a different type of exception is thrown than the one expected by the `catch` block, this is propagated to the calling method. Because there is no expectation of an exception leaving the method, the test fails. The comment in the `catch` block is necessary because it informs anyone reading the code that you intended to leave it blank. This is advisable for any time you leave a code block empty.

How can a test fail if it does not complete quickly enough?

The `@Test` annotation can take two parameters. One is `expected` (which you have already seen), which allows tests to pass when a certain type of exception is thrown. The other parameter is a `timeout`, which takes a value of type `long`. The number represents a number of milliseconds, and if the test is running for longer than that time, the test fails.

This test condition can help meet certain non-functional conditions. For example, if you were writing a service that had a requirement to respond within a second, you would write an integration test to make sure this requirement was met. Listing 9-7 demonstrates this.

LISTING 9-7: Failing long-running tests

```

@Test(timeout = 1000L)
public void serviceResponseTime() {
    // constructed against a real service
    final HighScoreService realHighScoreService = ...
    final Game gameUnderTest = new Game(realHighScoreService);
    final String highScoreDisplay = gameUnderTest.displayHighScores();
    assertNotNull(highScoreDisplay);
}

```

This integration test calls out to a real-world high score service, and if the test does not complete within a second, an exception is thrown with the notification that the test timed out.

Of course, this timeout is for the *whole test* to complete, not the specific long-running method call. Similar to the exception expectation, if you want to explicitly check that a method call took less than a certain amount of time, you can either have that single method call in the test, or run the method call in a separate thread, as shown in Listing 9-8.

LISTING 9-8: Explicitly timing tests

```

@Test
public void manualResponseTimeCheck() throws InterruptedException {
    final HighScoreService realHighScoreService =
        new StubHighScoreService();

    final Game gameUnderTest = new Game(realHighScoreService);

    final CountDownLatch latch = new CountDownLatch(1);
    final List<Throwable> exceptions = new ArrayList<>();

    final Runnable highScoreRunnable = new Runnable() {
        @Override
        public void run() {
            final String highScoreDisplay =
                gameUnderTest.displayHighScores();
            try {
                assertNotNull(highScoreDisplay);
            } catch (Throwable e) {
                exceptions.add(e);
            }
            latch.countDown();
        }
    };

    new Thread(highScoreRunnable).start();
    assertTrue(latch.await(1, TimeUnit.SECONDS));

    if(!exceptions.isEmpty()) {
        fail("Exceptions thrown in different thread: " + exceptions);
    }
}

```

Managing the timing has added quite a lot of code. If you are not familiar with threads or `CountDownLatch`s, these are covered in Chapter 11. What happens here is that the main executing thread waits for the thread running the test to complete, and if that takes longer than a second, the assertion surrounding `latch.await` returns `false`, so the test fails.

Also, JUnit will only fail a test for any failed assertions on the thread running the test, so you need to capture and collate any exceptions from assertions (or otherwise) from the spawned thread, and fail the test if any exceptions were thrown.

How does the `@RunWith` annotation work?

The `@RunWith` annotation is a class-level annotation, and it provides a mechanism for changing the default behavior of the test runner. The parameter to the annotation is a subclass of `Runner`. JUnit itself comes with several runners, the default being `JUnit4`, and a common alternative is the `Parameterized` class.

When a JUnit test is annotated with `@RunWith(Parameterized.class)`, several changes are made to the life cycle of the test and the way the test is run. A class-level method providing the test data is expected, and this returns an array of data to use for testing. This data could be hard-coded in the test, or for more sophisticated tests, this could be dynamically produced, or even pulled in from a filesystem, database, or another relevant storage mechanism.

However this data is generated, each element in the array from this method is passed into the constructor for the test suite, and all tests run with that data.

Listing 9-9 shows a test suite run with the `Parameterized` runner. It provides a layer of abstraction over the tests; all of the tests are run against each of the data sets.

LISTING 9-9: Using the `Parameterized` test runner

```
@RunWith(Parameterized.class)
public class TestWithParameters {

    private final int a;
    private final int b;
    private final int expectedAddition;
    private final int expectedSubtraction;
    private final int expectedMultiplication;
    private final int expectedDivision;

    public TestWithParameters(final int a,
                             final int b,
                             final int expectedAddition,
                             final int expectedSubtraction,
                             final int expectedMultiplication,
                             final int expectedDivision) {

        this.a = a;
        this.b = b;
        this.expectedAddition = expectedAddition;
    }
}
```

```

        this.expectedSubtraction = expectedSubtraction;
        this.expectedMultiplication = expectedMultiplication;
        this.expectedDivision = expectedDivision;
    }

    @Parameterized.Parameters
    public static List<Integer[]> parameters() {
        return new ArrayList<Integer[]>(3) {{
            add(new Integer[] {1, 2, 3, -1, 2, 0});
            add(new Integer[] {0, 1, 1, -1, 0, 0});
            add(new Integer[] {-11, 2, -9, -13, -22, -5});
        }};
    }

    @Test
    public void addNumbers() {
        assertEquals(expectedAddition, a + b);
    }

    @Test
    public void subtractNumbers() {
        assertEquals(expectedSubtraction, a - b);
    }

    @Test
    public void multiplyNumbers() {
        assertEquals(expectedMultiplication, a * b);
    }

    @Test
    public void divideNumbers() {
        assertEquals(expectedDivision, a / b);
    }
}

```

This listing introduces some new concepts. First, a new annotation, `@Parameterized.Parameters`, needs to be placed on a public class method, and it returns a list of arrays. The objects in each array are passed to the constructor of the test, with the same ordering in the array as the ordering to the constructor.

One thing to bear in mind is that for test suites that require many parameters, it can be unwieldy or unclear as to which position in the provided array matches which constructor argument.

For this listing, the `parameters()` class method returns a readily constructed instance of an `ArrayList`.

What can you do if you want to customize the running of your tests?

At times it may be appropriate to create your own test runner.

One property of a good suite of unit tests is that they should be atomic. This means that they should be able to be run in any sequence, with no dependency on any other test in the suite.

USING DOUBLE BRACES

The parameters for the test in Listing 9-9 use *syntactic sugar*, with double braces following the constructor. Inside the double braces, instance methods are called. What is happening here is that an anonymous subclass of `ArrayList` has been created, and inside that implementation, a block of code is defined to be run after the object has been constructed. Because the anonymous block is not defined as static, it is at the instance level, and so instance methods and variables can be called. The following code shows the logical, but more verbose, equivalent of that code, which should highlight exactly what is happening if it is not yet clear:

```
public class ListOfIntegerArraysForTest
    extends ArrayList<Integer[]> {
    {
        this.add(new Integer[]{1, 2, 3, -1, 2, 0});
        this.add(new Integer[]{0, 1, 1, -1, 0, 0});
        this.add(new Integer[]{-11, 2, -9, -13, -22, -5});
    }

    public ListOfIntegerArraysForTest() {
        super(3);
    }
}

@Parameterized.Parameters
public static List<Integer[]> parameters() {
    return new ListOfIntegerArraysForTest();
}
```

If you are still unsure of how this works, try running the code with some logging statements, or step through it in a debugger in your favorite IDE.

It is not advisable to use this double-curly-bracket technique for production code, because it creates a new anonymous class for each declaration, which will take up space in the PermGen area of the JVM's memory for each new anonymous class. There will also be the overhead actually loading the class, such as verification and initializing the class. But for testing, this provides a neat and concise way of creating immutable objects, with the object declaration and construction all held together. Be wary that this can be confusing to people who are not familiar with the construct, so perhaps include a comment, or talk directly with your team to make them aware of this.

Though this is generally an implied rule, there is nothing within the JUnit library to enforce this. However, it should be possible to write your own test runner that runs the test methods in a random order. Should any test be dependent on any other, running the tests in a different order could expose this.

The JUnit runners are concrete implementations of the `Runner` abstract class. Listing 9-10 shows the class structure, along with the methods, that must be implemented for a custom runner.

LISTING 9-10: The Runner class

```

public abstract class Runner implements Describable {
    public abstract Description getDescription();
    public abstract void run(RunNotifier notifier);

    public int testCount() {
        return getDescription().testCount();
    }
}

```

In your implementation of the `Runner`, there is just one convention to follow: You must provide a single argument constructor, which takes a `Class` object. JUnit will call this constructor once, passing in the class of the test suite. You are then free to do whatever you require with that class.

The `Runner` abstract class shows two methods that must be implemented: `getDescription`, which informs which methods to run and in which order, and `run`, which actually runs the test.

Given a class of the test suite and these two methods, it should be clear that this gives total flexibility as to what is run, how it is run, and how to report the outcome.

The `Description` class is a relatively straightforward class that defines what exactly is to be run, and the order of the run. To create a `Description` instance for running a set of JUnit tests, you call the static method `createSuiteDescription`. To add actual tests, you add child `Description` objects with the `addChild` method. You create the actual test description instances with another static method, `createTestDescription`. Considering that `Description` objects themselves contain `Descriptions`, you can create a tree of `Descriptions` several instances deep. In fact, this is what the `Parameterized` runner does: For a test suite, it creates a child for each of the parameters, and for each of those children, it creates a child for each test.

Given that the `getDescription` method on the `Runner` interface has no parameters, you need to create the instance to return before the method is called: on construction of the runner. Listing 9-11 shows the construction and the `getDescription` method of the `RandomizedRunner`.

LISTING 9-11: The RandomizedRunner description generation

```

public class RandomizedRunner extends Runner {

    private final Class<?> testClass;
    private final Description description;

    private final Map<String, Method> methodMap;

    public RandomizedRunner(Class<?> testClass)
        throws InitializationError {

        this.testClass = testClass;
        this.description =

```

continues

LISTING 9-11 *(continued)*

```

        Description.createSuiteDescription(testClass);

        final List<Method> methodsUnderTest = new ArrayList<>();

        for (Method method : testClass.getMethods()) {
            if (method.isAnnotationPresent(Test.class)) {
                methodsUnderTest.add(method);
            }
        }

        Collections.shuffle(methodsUnderTest);

        methodMap = new HashMap<>();

        for (Method method : methodsUnderTest) {
            description.addChild(Description.createTestDescription(
                testClass,
                method.getName()
            ));

            System.out.println(method.getName());
            methodMap.put(method.getName(), method);
        }
    }

    @Override
    public Description getDescription() {
        return description;
    }

    @Override
    public void run(RunNotifier runNotifier) {
        // not yet implemented
    }

    public static void main(String[] args) {
        JUnitCore.main(RandomizedRunner.class.getName());
    }
}

```

Reflection is used to find all of the methods from the given `testClass` suite; each is tested for the presence of the `@Test` annotation, and if so, is captured for running as a test later.

Notice that there is no constraint on using the `@Test` annotation for marking the methods to be run as tests. You could have used any annotation, or indeed any kind of representation, such as a prefix of `test` on any relevant method. (This was the standard before JUnit 4, before annotations were part of the Java language.) In fact, as you will see in the section “Creating System Tests with Behavior-Driven Development” later in this chapter, these tests are run without using the `@Test` annotation.

For brevity, this `Runner` implementation is more limited than the default implementation in JUnit. It assumes that all methods annotated with `@Test` will be runnable as a test. There is no understanding of any other annotation such as `@Before`, `@After`, `@BeforeClass`, and `@AfterClass`. Also note that `@Ignore` is not checked either: Merely the presence of `@Test` will mark that method for testing.

As each method on the class is checked for the `@Test` annotation, it is placed into a holding collection, and is then randomly sorted using the utility method `Collections.shuffle`.

The methods are added to the `Description` object in that random order. The description uses the method name to keep track of the test to run. So, for convenience, the method object is added to a mapping against its name for later lookup by the description's method name.

The `Description` object itself informs the JUnit framework of what is going to be run; it's still up to the runner to actually run the tests and decide if the test suite has succeeded, or if not, what has failed. When the framework is ready to run the tests, it calls the `Runner`'s `run` method. Listing 9-12 is the `RandomizedRunner` class's implementation.

LISTING 9-12: The `RandomizedRunner`'s `run` method

```
@Override
public void run(RunNotifier runNotifier) {
    runNotifier.fireTestStarted(description);
    for (Description descUnderTest : description.getChildren()) {
        runNotifier.fireTestStarted(descUnderTest);
        try {
            methodMap.get(descUnderTest.getMethodName())
                .invoke(testClass.newInstance());

            runNotifier.fireTestFinished(descUnderTest);
        } catch (Throwable e) {
            runNotifier.fireTestFailure(
                new Failure(descUnderTest, e.getCause()));
        }
    }
    runNotifier.fireTestFinished(description);
}
```

The `run` method has one parameter: a `RunNotifier` object. This is the hook back to the JUnit framework to say whether each test has succeeded or failed, and if it has failed, what exactly happened.

Remember that for the `RandomizedRunner`, the `Description` object had two levels: a top level for the suite, and then a child for each test. The `run` method needs to notify the framework that the suite has started running. It then calls each of the children, in turn. These children were added to the description in the random order.

For each of these children, a new instance of the test suite class is created, and the test method is invoked using reflection. Before the invocation, the framework is notified that the test is about to be run.

JUNIT AND IDES

You may recognize that IDEs such as IntelliJ and Eclipse have very tight integration with JUnit: during a test run, their interfaces show exactly what test is running, what has already run, what has passed, and what has failed. The IDEs are using the calls to the `RunNotifier` to update their own test runner display.

If and when the test passes, the `RunNotifier` is told that the test has completed. But if the invocation throws any kind of exception, the `catch` block updates the `RunNotifier` of the failure. One point to note here is that the `Failure` object takes two parameters: the description object that has failed and the exception instance causing the failure. Considering this method was invoked using reflection, any exception caught will be a `java.lang.reflect.InvocationTargetException`, with the causing exception chained to that exception. It is therefore much clearer to any user of the `RandomizedRunner` to display the cause, ignoring the exception from reflection.

Now, any test suites marked with `@RunWith(RandomizedRunner.class)` can take advantage of having the test methods run in a non-specific order. Should multiple runs of the suite result in flickering tests, this will show an issue with one or more of the tests: They may depend on being run in a certain order, or the tests may not be cleaning their environment before or after their run.

A couple of possible extensions to this runner could be to try and figure out which test is causing a problem. When creating the `Description` object, you could add each method under test several times before the list of all methods is shuffled. Another approach could be to perform a binary search on the tests. Should any test fail, you could split the test list in two and rerun each list run. If all the tests in that sublist pass, the problematic test is most likely in the other half. You could further split the list in a similar manner until the test causing problems has been isolated.

ELIMINATING DEPENDENCIES WITH MOCKS

What is the difference between a unit test and an integration test?

JUnit has something of an unfortunate name, in that it can be used for both unit testing and integration testing, among other things. The usual definition for a unit test is a test that purely tests a single *unit* of functionality. It will have no side effects. A well-defined unit test, run several times with no other changes, will produce exactly the same result each time.

Integration tests are a more complex beast. As their name suggests, integration tests examine several parts of a system to make sure that when integrated, these parts behave as expected. These tests often cover things like calling and reading from a database, or even involving calls to another server. There can often be dummy data inserted into a database, which will be expected for the test. This can lead to brittle tests, should database schemas or URL endpoints change. It is possible to use in-memory databases to test database interaction within integration tests. This is covered in Chapter 12.

For any non-complex system, testing single classes in isolation can be difficult, because they will have dependency on external factors, or other classes, which in turn can have dependency on external factors, and so on. After all, any running application that has no external dependencies cannot affect the outside world—it is essentially useless!

To break down the dependencies between classes and the outside world, you have two main things to follow: the use of dependency injection and mocking. The two go hand in hand. Dependency injection and specifically how to use that with integration testing is covered in more detail in Chapter 16, but Listing 9-13 highlights a small example.

LISTING 9-13: A game with a dependency on a high score service

```
public interface HighScoreService {
    List<String> getTopFivePlayers();
    boolean saveHighScore(int score, String playerName);
}

public class Game {

    private final HighScoreService highScoreService;

    public Game(HighScoreService highScoreService) {
        this.highScoreService = highScoreService;
    }

    public String displayHighScores() {
        final List<String> topFivePlayers =
            highScoreService.getTopFivePlayers();
        final StringBuilder sb = new StringBuilder();

        for (int i = 0; i < topFivePlayers.size(); i++) {
            String player = topFivePlayers.get(i);
            sb.append(String.format("%d. %s\n", i+1, player));
        }

        return sb.toString();
    }
}
```

You could imagine an implementation of the `HighScoreService` class calling a database, which would sort results before returning the list for the top players, or perhaps the class calling a web service to get the data.

The `Game` class *depends* on the `HighScoreService`—it should not care how and where the high scores come from.

To test the algorithm in the `displayHighScores()` method, an instance of the `HighScoreService` is needed. If the implementation used in testing is configured to call an external web service, and that service is down for one of any number of reasons, then the test will fail, and you can do little to rectify it.

However, if a special instance is created, especially for testing, that reacts exactly as expected, this instance could be passed to the object when it is constructed, and it would allow testing of the `displayHighScores()` method. Listing 9-14 is one such implementation.

LISTING 9-14: Testing the Game class with a stub

```
public class GameTestWithStub {

    private static class StubHighScoreService
        implements HighScoreService {
        @Override
        public List<String> getTopFivePlayers() {
            return Arrays.asList(
                "Alice",
                "Bob",
                "Charlie",
                "Dave",
                "Elizabeth");
        }

        @Override
        public boolean saveHighScore(int score, String playerName) {
            throw new UnsupportedOperationException(
                "saveHighScore not implemented for this test");
        }
    }

    @Test
    public void highScoreDisplay() {
        final String expectedPlayerList =
            "1. Alice\n" +
            "2. Bob\n" +
            "3. Charlie\n" +
            "4. Dave\n" +
            "5. Elizabeth\n";

        final HighScoreService stubbedHighScoreService =
            new StubHighScoreService();
        final Game gameUnderTest = new Game(stubbedHighScoreService);

        assertEquals(
            expectedPlayerList,
            gameUnderTest.displayHighScores());
    }
}
```

Using a *stub* implementation of the `HighScoreService` means that the `Game` class will return the same list every time, with no latency from a network or database processing, and no issues around if a given high score service is running at the time of the test. This is not a silver bullet, though. You would want to verify other properties of this call to the high score service; for instance, that the service is called once *and only once* to generate this list, and not that the `Game` instance is doing

something inefficient, such as calling the service once for each of the five players, or that it's not called at all and somehow the `Game` is using a stale cached value.

Introducing a *mock object* can take care of these criteria. You can think of a mock as a smarter stub. It has the capability to respond differently to different method calls, regardless of the arguments passed, and it can record each call to assert that the calls were actually made as expected.

One such Java library is called Mockito. Listing 9-15 expands on Listing 9-14, testing the `Game` class more thoroughly.

LISTING 9-15: Testing the `Game` class with a mock

```
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;

public class GameTestWithMock {

    private final Game gameUnderTest;
    private final HighScoreService mockHighScoreService;

    public GameTestWithMock() {
        final List<String> firstHighScoreList = Arrays.asList(
            "Alice",
            "Bob",
            "Charlie",
            "Dave",
            "Elizabeth");
        final List<String> secondHighScoreList = Arrays.asList(
            "Fred",
            "Georgia",
            "Helen",
            "Ian",
            "Jane");

        this.mockHighScoreService = mock(HighScoreService.class);

        Mockito.when(mockHighScoreService.getTopFivePlayers())
            .thenReturn(firstHighScoreList)
            .thenReturn(secondHighScoreList);

        this.gameUnderTest = new Game(mockHighScoreService);
    }

    @Test
    public void highScoreDisplay() {
        final String firstExpectedPlayerList =
            "1. Alice\n" +
            "2. Bob\n" +
            "3. Charlie\n" +
            "4. Dave\n" +
```

continues

LISTING 9-15 *(continued)*

```
        "5. Elizabeth\n";

    final String secondExpectedPlayerList =
        "1. Fred\n" +
        "2. Georgia\n" +
        "3. Helen\n" +
        "4. Ian\n" +
        "5. Jane\n";

    final String firstCall = gameUnderTest.displayHighScores();
    final String secondCall = gameUnderTest.displayHighScores();

    assertEquals(firstExpectedPlayerList, firstCall);
    assertEquals(secondExpectedPlayerList, secondCall);

    verify(mockHighScoreService, times(2)).getTopFivePlayers();
}
}
```

This test can be broadly split into three parts: The test and mock are set up (which is done in the test suite constructor), the test itself is run, and then the assertions are checked.

In the constructor, a mock `HighScoreService` class is created, using Mockito's mocking framework. This creates the special instance, which is an implementation of the `HighScoreService` interface. Following this mock generation, the mock is instructed how to react to the `getTopFivePlayers()` call. It is told to give different responses to its first and second call, simulating the data from the service changing between calls. It is not relevant for this test, but for any subsequent calls to `getTopFivePlayers()`, the mock will return the second list repeatedly. Once the mock has been configured, this instance is passed to the constructor of the `Game` class.

The test is then executed. The `displayHighScores()` method is called twice, and an expectation has been made that for each call to `displayHighScores()`, a call is made to `getTopFivePlayers()` on the `HighScoreService` instance. The responses from each call are captured by the mock itself.

The responses are then asserted against the expectation of how the `Game` will format the top five players. Finally, the test checks with Mockito that the mock was used in the correct way. This test expected the `getTopFivePlayers()` method on the mock to be called twice. If this verification is not correct, an appropriate Mockito exception is thrown, failing the test.

Take a moment to make sure you fully understand how the mock interacts with the `Game` class. Note that, for the high scores, the `Game` class has been verified to work properly, at least for the test cases presented, without any thought given to the implementation of the `HighScoreService`. Any future change to a `HighScoreService` implementation that does not change the interface definition will have no effect on this test. A clear separation has been made between the `Game` and the `HighScoreService`.

The mocks that Mockito provides are extremely flexible. Other libraries can only mock interfaces, but Mockito mocks can mock concrete implementations too. This is extremely useful for working with legacy code, where perhaps dependency injection wasn't used properly, or interfaces were not used at all.

Mockito cannot mock final classes, due to the way they are stored and constructed in the JVM. The JVM takes precautionary measures for security to make sure a final class is not replaced.

How expressive can you make your tests?

You can use the Hamcrest matcher library in conjunction with JUnit to help articulate assertions and improve readability of tests. The construction of an assertion using Hamcrest matchers is a form of *Domain-Specific Language* (DSL). Listing 9-16 shows a simple example.

LISTING 9-16: Using the Hamcrest matchers

```
import static org.hamcrest.MatcherAssert.*;
import static org.hamcrest.Matchers.*;

public class HamcrestExample {

    @Test
    public void useHamcrest() {
        final Integer a = 400;
        final Integer b = 100;

        assertThat(a, is(notNullValue()));
        assertThat(a, is(equalTo(400)));
        assertThat(b - a, is(greaterThan(0)));
    }
}
```

These assertion lines attempt to read like English. The library has several methods that return an instance of `Matcher`. When the matcher is evaluated, if it is not true, the `assertThat` method throws an exception. Similar to the `Assert` class methods in JUnit, the `assertThat` method can take an extra `String` parameter, which is displayed on the assertion failing. Again, try to make the reason read well in the code, and also highlight an error by describing what should happen.

The `is` method is more syntactic sugar. Although it does not add any functional value, it provides help to the programmer, or anyone reading the code. Because the `is` method merely delegates to the matcher inside the method, it is optional, serving only to make the code more legible.

The use of Hamcrest matchers often divides opinion. If you like them, use them. If not, then unless you are specifically asked to use them, nobody will expect you to.

CREATING SYSTEM TESTS WITH BEHAVIOR-DRIVEN DEVELOPMENT

What is Behavior-Driven Development?

In relation to unit testing, *Behavior-Driven Development* (BDD) is a much newer concept. BDD tests are composed of two components: the test script, which is written in as close to natural language as possible, and the code backing the test script to run.

A general motivator behind these tests is to provide a very high-level test, independent of any implementing code, essentially treating a system as a black box. These tests are often called *system tests*. Listing 9-17 is a script, called a *feature*, for use in a BDD test.

LISTING 9-17: A feature for a BDD test

Feature: A simple mathematical BDD test

Scenario Outline: Two numbers can be added

Given the numbers <a> and

When the numbers are added together

Then the result is <result>

Examples:

a	b	result
1	10	11
0	0	0
10	-5	5
-1	1	0

This has been written to run with the Cucumber library. Cucumber originated as a BDD framework for the Ruby language, and was later rewritten to run natively on the JVM. Other BDD frameworks, such as JBehave, work in a similar way.

It is quite easy to imagine that a non-technical user of the system, perhaps the product owner or a major stakeholder, could write scripts like this, or perhaps even a customer, if that is relevant.

The script follows a prescribed set of instructions, called *steps*. The steps are split into three: Given, When, and Then.

- The Given steps prime the system into a known state, which is necessary for the following steps.
- The When steps perform the action under test. This will, or at least should, have a measurable effect on the system under test.
- The Then steps measure that the When steps were performed successfully.

The ordering of the steps is important. Given is followed by When, which is followed by Then. It is possible to have multiple steps performed for each of these keywords; but rather than having multiple Given/When/Then, you use And instead:

```
Given a running web server
And a new user is created with a random username
... etc
```

Of course, this script needs to be backed up by actual running code in order to verify that the test passes. Listing 9-18 is a possible implementation for the feature in Listing 9-17.

LISTING 9-18: Cucumber steps

```
public class CucumberSteps {

    private int a;
    private int b;
    private int calculation;

    @Given("^the numbers (.*?) and (.*?)$")
    public void captureNumbers(final int a, final int b) {
        this.a = a;
        this.b = b;
    }

    @When("^the numbers are added together$")
    public void addNumbers() {
        this.calculation = a + b;
    }

    @Then("^the result is (.*?)$")
    public void assertResult(final int expectedResult) {
        assertEquals(expectedResult, calculation);
    }
}

@RunWith(Cucumber.class)
public class CucumberRunner { }
```

Each of the steps is defined in its own method, with an annotation for that method. The annotation has a string parameter, which itself is a regular expression. When the test is run, the runner searches the methods with the correct annotation to find a regular expression match for that step.

Listing 9-17 parameterized the scenario, with what is called a *scenario outline*. This means that each step can be called with a variable input. This input can be captured with the regular expression grouping. A parameter of the appropriate type is then required on the method, giving the code access to the different test inputs.

Note that the implementation of Cucumber scenarios requires a specific JUnit runner, provided by the Cucumber jar. The `@RunWith` annotation must reside on a separate class definition than the steps. Even though the Cucumber tests are run through the JUnit library, using this runner means that no `@Test` annotations are necessary.

If a method can be used by more than one of steps classifications, then merely adding a new annotation to the method will allow that code to be run for that step.

Once you have written a few tests, a small vocabulary will be available, and you can reuse these `Given/When/Then` lines to make new tests, requiring minimal new coding effort (or perhaps none at all). Designing tests this way gives plenty of scope for those writing the test scripts to think up

their own scenarios, and writing tests from steps already implemented. Following on from the addition example, imagine a new requirement to be able to subtract numbers. You could write the new scripts, and you would need to add only one new method:

```
@When("^the first number is subtracted from the second$")
public void subtractAFromB() {
    this.calculation = b - a;
}
```

For multiple steps where `And` is used, the correct `Given/When/Then` annotation is used.

BDD tests are a useful vehicle for managing a test over several changes in state. This could work very well with database interaction. The `Given` step could prime the database ready for the test, checking for or inserting necessary data. The `When` step performs the test, and the `Then` step asserts that the test was successful. As with the regular JUnit runner, you can provide `@Before` and `@After` annotated methods that are run before each scenario.

SUMMARY

If you are not writing tests when you write code, now is the time to start. You will find that the more experience you get writing tests, the easier it is to write your production code. You will break down problems into manageable, testable chunks. Any future changes that break existing code will be spotted sooner rather than later. Eventually, writing tests will become second nature.

Try to break up any tests that you do write into logical units. Write unit tests for single pieces of functionality, which have no side effects. Any dependencies will be mocked. Write integration tests for any test code that has a side effect, such as writing to a filesystem or database, and clean up any side effects when the test completes, whether it passes or fails. One convention is to suffix any integration test class with `IntegrationTest`, and unit tests simply with `Test`.

When faced with a technical test for an interview, writing tests is an ideal way to help express why you wrote code in a certain way. Even in a time-limited test, writing even basic tests first will help structure your code, and help avoid errors in any last-minute refactoring.

The error message parameter on the `Assert` class methods is a perfect way to open a dialogue with anyone reading your code; it is a space to write exactly what you are thinking, and exactly what your code should do. Use this space to communicate your thoughts.

Importing the JUnit jar is very simple in IntelliJ and Eclipse. Both these IDEs recognize when you are writing test code, and will give you the option to import the library directly: no need for editing Maven POMs and reimporting whole projects.

The technical interview we use with my current employer does not advise writing tests at all, but whenever a completed interview is received, the first thing we do is check what tests have been written. If there are no tests at all, it is highly unlikely that candidate will be asked in for a face-to-face interview.

Chapter 10 looks at the Java Virtual Machine, how the Java language interacts with it, and what an interviewer may ask about how to optimize its performance and memory footprint.

10

Understanding the Java Virtual Machine

The *Java virtual machine* (JVM) is the platform upon which your programs run. It is built for that specific operating system and architecture, and sits between the operating system and any compiled programs or applications you write, making it platform agnostic. Java programs are compiled (using `javac`) into *bytecode*. This is interpreted by the JVM into the specific instructions for that architecture and operating system.

Other compiled languages such as C++ or Objective-C need to be compiled to run with a defined host architecture and operating system, but this is not the case with Java. Regardless of the platform on which your Java code was compiled, it can run on any JVM of the same or later version.

Considering this is where your code runs, it is important to understand how the JVM works, and how you can tweak certain parameters to obtain optimum performance for your code.

This chapter covers the particulars around the JVM's memory model, such as how that memory is allocated to your objects, and how the JVM reclaims that memory once you no longer need it. It then reviews the different memory areas and how the JVM uses them. Finally, this chapter looks at certain language hooks to allow executing code to interact with a running JVM.

GARBAGE COLLECTION

How is memory allocated?

The `new` keyword allocates memory on the Java *heap*. The heap is the main pool of memory, accessible to the whole of the application. If there is not enough memory available to allocate for that object, the JVM attempts to reclaim some memory from the heap with a garbage collection. If it still cannot obtain enough memory, an `OutOfMemoryError` is thrown, and the JVM exits.

The heap is split into several different sections, called *generations*. As objects survive more garbage collections, they are promoted into different generations. The older generations are not garbage collected as often. Because these objects have already proven to be longer lived, they are less likely to be garbage collected.

When objects are first constructed, they are allocated in the *Eden Space*. If they survive a garbage collection, they are promoted to *Survivor Space*, and should they live long enough there, they are allocated to the *Tenured Generation*. This generation is garbage collected much less frequently.

There is also a fourth generation, called the *Permanent Generation*, or *PermGen*. The objects that reside here are not eligible to be garbage collected, and usually contain an immutable state necessary for the JVM to run, such as class definitions and the `String` constant pool. Note that the PermGen space is planned to be removed from Java 8, and will be replaced with a new space called Metaspace, which will be held in native memory.

USING THE PERMGEN SPACE

For most applications, the PermGen area contains traditional class definitions, `String` constants, and not much else. Newer languages running on the JVM, such as Groovy, have the capability to create dynamic class definitions, and when used under load, this can fill up the PermGen space easily. You must be careful when creating many dynamic class definitions; and you may need to tweak the default memory allocation for PermGen space.

What is garbage collection?

Garbage collection is the mechanism of reclaiming previously allocated memory, so that it can be reused by future memory allocations. In most languages, garbage collection is automated; you do not need to free up the memory yourself. In Java, whenever a new object is constructed, usually by the `new` keyword, the JVM allocates an appropriate amount of memory for that object and the data it holds.

When that object is no longer needed, the JVM needs to reclaim that memory, so that other constructed objects can use it.

With languages such as C and C++, it is necessary to manage these memory allocations manually, usually through function calls to `malloc` and `free`. More modern languages, such as Java and C#, have an automatic system for this, taking the effort, and any potential mistakes, away from the programmer.

Several different algorithms for garbage collection exist, but they all have the same goal of finding allocated memory that is no longer referenced by any live code, and returning that to a pool of available memory for future allocations.

The traditional garbage collection algorithm in Java is called mark-and-sweep. Each object reference in running code is marked as live, and each reference within that object is traversed and also marked as live, and so on, until all routes from live objects have been traced.

Once this is complete, each object in the heap is visited, and those memory locations not marked as live are made available for allocation. During this process, all of the threads in the JVM are paused to allow the memory to be reclaimed, known as *stop-the-world*. Naturally, the garbage collector tries to minimize the amount of time this takes. There have been several iterations of the garbage collection algorithm since Java was first released, with as much of the work done in parallel as possible.

Java 6 introduced a new algorithm, called *Garbage First* (G1). It was approved for test use in Java 6, and production use in Java 7. G1 still concentrates on a mark-and-sweep algorithm, running in parallel, but it concentrates on areas of mainly empty memory first in an attempt to keep large areas of free space available.

Other operations are also performed during garbage collection, such as promotion to different generations, and grouping frequently accessed objects together by moving the objects around within memory to try to retain as much free space as possible. This is called *compaction*. Compaction takes place while the JVM is in its stop-the-world phase, as live objects are potentially moving to different physical memory locations.

MEMORY TUNING

What is the difference between the stack and the heap?

Memory is split into two major parts, the *stack* and the *heap*. Most discussion so far in this chapter has been about object allocation, which is what the heap is used for.

The stack is the place where any primitive values, references to objects, and methods are stored. The lifetime of variables on the stack is governed by the *scope* of the code. The scope is usually defined by an area of code in curly brackets, such as a method call, or a `for` or `while` loop. Once the execution has left that scope, those variables declared in the scope are removed from the stack.

When you call a method, those declared variables are placed on top of the stack. Calling another method within that stack pushes the new method's variables onto the stack.

A recursive method is a method that, directly or indirectly, calls itself again. If a method calls itself too many times, the stack memory fills up, and eventually any more method calls will not be able to allocate their necessary variables. This results in a `StackOverflowError`. The *stack trace* from the resultant exception usually has dozens of calls to the same method. If you are writing a recursive method, it is important that you have a state within the method known as the *base case*, where no more recursive calls will be made.

Recursive methods usually use much more stack space, and therefore more memory, than an iterative counterpart. Although a recursive method can look neat and elegant, be aware of possible

out-of-memory errors due to stack overflow. Listing 10-1 shows the same algorithm written in both a recursive and an iterative style.

LISTING 10-1: Using the stack for loops

```
@Test
public void listReversals() {
    final List<Integer> givenList = Arrays.asList(1, 2, 3, 4, 5);
    final List<Integer> expectedList = Arrays.asList(5, 4, 3, 2, 1);

    assertEquals(expectedList.size(), reverseRecursive(givenList).size());
    assertEquals(expectedList.size(), reverseIterative(givenList).size());
}

private List<Integer> reverseRecursive(List<Integer> list) {
    if (list.size() <= 1) { return list; }
    else {
        List<Integer> reversed = new ArrayList<>();
        reversed.add(list.get(list.size() - 1));
        reversed.addAll(reverseRecursive(list.subList(0, list.size() - 1)));
        return reversed;
    }
}

private List<Integer> reverseIterative(final List<Integer> list) {
    for (int i = 0; i < list.size() / 2; i++) {
        final int tmp = list.get(i);
        list.set(i, list.get(list.size() - i - 1));
        list.set(list.size() - i - 1, tmp);
    }

    return list;
}
```

Compare the two algorithms for reversing an array. How much space does each algorithm need? For the recursive definition, each time the method is recursively called, a new list is created. These lists from each method call must be held in memory until the list has been completely reversed. Although the actual lists will be held on the heap (because that is where objects are stored), each method call needs stack space.

For the iterative version, the only space needed is a variable to hold one value while it is being swapped with its counterpart at the other end of the list. There are no recursive calls, so the stack will not grow very deep. No new objects are allocated, so no extra space is taken on the heap.

Experiment with the number of elements in the list. Can you make the recursive method throw a `StackOverflowError`?

How can you define the size of the heap for the JVM?

The JVM provides several command-line arguments for defining the size of the memory allocated to the different memory areas.

When starting up the JVM, you can specify the maximum heap size with the command-line flag `-Xmx` and a size. For example, starting the JVM with `java -Xmx512M <classname>` creates a JVM with a maximum heap size of 512 megabytes. Suffixes for the memory size are `G` for gigabytes, `M` for megabytes, or `K` for kilobytes. It is important to note that the JVM will not allocate this memory in its entirety on startup; it will grow to a maximum of that size only if needed. Before the JVM expands its memory allocation, it will try to perform as much garbage collection as possible.

To specify the initial amount of memory allocated to the JVM, use the `-Xms` argument. It works in the same way as `-Xmx`. This argument is advisable if you know you are going to need a certain amount of memory for your function, as it will save your application from excessive slow garbage collections before expanding to your required size.

If both arguments are set to the same value, the JVM will ask the operating system for that full memory allocation on startup, and it will not grow any larger.

For the initial memory allocation, the default value is 1/64 of the memory on the computer, up to 1 GB. For the maximum default, it is the smaller of 1 GB and a quarter of the computer's physical memory. Considering that these can vary wildly from computer to computer, you should specify the values explicitly for any code running in a production environment, and make sure your code performs to satisfaction for these values.

Similar to setting initial and maximum heap sizes, you have JVM startup arguments for setting the size of the stack, too. For most running programs, you should avoid setting these. If you find yourself running into `StackOverflowExceptions` on a regular basis, you should examine your code and replace as many recursive methods with iterative counterparts as possible.

Other relevant JVM arguments include `-XX:PermSize` and `-XX:MaxPermSize` for the permanent generation. You may want to set this if you have a large number of classes or string constants, or if you are creating many dynamic class definitions using a non-Java language.

Is it possible to have memory leaks in Java?

A common misconception with the Java language is that, because the language has garbage collection, memory leaks are simply not possible. Admittedly, if you are coming from a C or C++ background you will be relieved that you do not need to explicitly free any allocated memory, but excessive memory usage can occur if you are not careful. Listing 10-2 is a simple implementation of a stack that can be exposed to memory leaks.

LISTING 10-2: A collection with a memory leak

```
public class MemoryLeakStack<E> {
    private final List<E> stackValues;
```

continues

LISTING 10-2 *(continued)*

```

    private int stackPointer;

    public MemoryLeakStack() {
        this.stackValues = new ArrayList<>();
        stackPointer = 0;
    }

    public void push(E element) {
        stackValues.add(stackPointer, element);
        stackPointer++;
    }

    public E pop() {
        stackPointer--;
        return stackValues.get(stackPointer);
    }
}

```

Ignoring any concurrency issues or elementary exceptions, such as calling `pop` on an empty stack, did you manage to spot the memory leak? When popping, the `stackValues` instance keeps a reference to the popped object, so it cannot be garbage collected. Of course, the object will be overwritten on the next call to `push`, so the previously popped object will never be seen again; but until that happens, it cannot be garbage collected.

If you are not sure why this is a problem, imagine that each element on this stack is an object holding the contents of a large file in memory; that memory cannot be used for anything else while it is referenced in the stack. Or imagine that the `push` method is called several million times, followed by the same number of pops.

A better implementation for the `pop` method would be to call the `remove` method on the `stackValues` list; `remove` still returns the object in the list, and also takes it out of the list completely, meaning that when any client code has removed all references to the popped object, it will be eligible for garbage collection.

INTEROPERABILITY BETWEEN THE JVM AND THE JAVA LANGUAGE

This section covers special methods and classes.

What is the lifecycle from writing a piece of Java code to it actually running on the JVM?

When you wish for some of your Java code to be run on a JVM, the first thing you do is *compile* it. The compiler has several roles, such as making sure you have written a legal program and making sure you have used valid types. It outputs *bytecode*, in a `.class` file. This is a binary format similar

to executable machine instructions for a specific architecture and operating system, but this is for the JVM.

The operation of bringing the bytecode for a class definition into the memory of a running JVM is called *classloading*. The JVM has classloaders, which are able to take binary `.class` files and load them into memory. The classloader is an abstraction; it is possible to load class files from disk, from a network interface, or even from an archived file, such as a JAR. Classes are only loaded on demand, when the running JVM needs the class definition.

It is possible to create your own classloader, and start an application that uses your loader to find classes from some location. Naturally, there are some security implications with this, so that an arbitrary application cannot be started with a malicious classloader. For instance, Java applets are not allowed to use custom classloaders at all.

Once a class has been loaded, the JVM itself verifies the bytecode to make sure it is valid. Some of the checks include making sure the bytecode does not branch to a memory location outside of its own class bytes, and all of the code instructions are complete instructions.

Once the code has been verified, the JVM can interpret the bytecode into the relevant instruction code for the architecture and operating system it is running on. However, this can be slow, and some of the early JVMs were notoriously slow, especially when working with GUIs for this exact reason. However, the *Just In Time* compiler (JIT) dynamically translates the running bytecode into native instructions so that interpreting the bytecode is not necessary. As this is performed dynamically, it is possible for the JIT to create highly optimized machine code based directly on what the state of the application is while it is running.

Can you explicitly tell the JVM to perform a garbage collection?

The static `gc` method on the `System` class is the method to call to tell the JVM to run a garbage collection, but calling this does not guarantee that a garbage collection will happen. The documentation for the `gc` method says:

Calling the `gc` method suggests that the Java Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse. When control returns from the method call, the Java Virtual Machine has made a best effort to reclaim space from all discarded objects.

It is not possible to enforce a garbage collection, but when you call `gc`, the JVM will take your request into account, and will perform a garbage collection if it can. Calling this method may make your code slower, putting your whole application in a stop-the-world state during garbage collection.

Explicit calls to `System.gc` are usually another form of code smell; if you are calling this in an effort to free memory, it is more than likely you have a memory leak, and you should try to address that, rather than attempting to sprinkle your code with explicit requests to the garbage collector.

What does the `finalize` method do?

The `finalize` method is a protected method, inherited from `Object`. When the JVM is about to garbage collect the object, this `finalize` method is called first. The rationale for this method is to tie off any loose ends, to close off any resources that the garbage-collected object was dependent upon.

One overriding concern with this method is that you have no control over when it is called: It is called if, and only if, the JVM decides to garbage collect the object. If you were to use this method for closing a database connection or a file handle, you could not be sure when exactly this event would happen. If you have many objects configured in this manner, you run the risk of exhausting a database connection pool, or perhaps even having too many files open at once.

If you are writing code that depends on an external resource that needs to be closed explicitly, like a database, filesystem, or network interface, you should aim to close your resources as soon as possible. When appropriate, use Java 7's new try-with-resources construct; and keep resources open for as little time as possible.

What is a `WeakReference`?

A `WeakReference` is a generic container class, and when the contained instance has no strong references, it is eligible for garbage collection.

An alternative approach to the stack implementation in Listing 10-1 would have been to hold a list of `WeakReferences` of the elements. Then, upon garbage collection, any elements that have no other references would be set to `null`. Listing 10-3 shows a possible declaration with an additional method, `peek`.

LISTING 10-3: A stack implementation with `WeakReference` objects

```
public class WeakReferenceStack<E> {

    private final List<WeakReference<E>> stackReferences;
    private int stackPointer = 0;

    public WeakReferenceStack() {
        this.stackReferences = new ArrayList<>();
    }

    public void push(E element) {
        this.stackReferences.add(
            stackPointer, new WeakReference<>(element));
        stackPointer++;
    }

    public E pop() {
        stackPointer--;
    }
}
```



```

        return this.stackReferences.get(stackPointer).get();
    }

    public E peek() {
        return this.stackReferences.get(stackPointer-1).get();
    }
}

```

When the new element is pushed in the stack, it is stored as a `WeakReference`, and when it is popped, the `WeakReference` is retrieved, and `get` is called to get that object. Now, when any client code has no more pointers to that object, it will be eligible for removal in the next garbage collection.

The `peek` method simply returns the top element on the stack, without removing it.

Listing 10-4 shows that the reference in the stack is set to `null` when all strong references to the value are removed. The `ValueContainer` class used merely holds a string, but its `finalize` method has been overridden to highlight this method being called by the garbage collector. If you remove the `System.gc()` line, the test will fail.

LISTING 10-4: Using a stack with WeakReferences

```

@Test
public void weakReferenceStackManipulation() {
    final WeakReferenceStack<ValueContainer> stack = new WeakReferenceStack<>();

    final ValueContainer expected = new ValueContainer("Value for the stack");
    stack.push(new ValueContainer("Value for the stack"));

    ValueContainer peekedValue = stack.peek();
    assertEquals(expected, peekedValue);
    assertEquals(expected, stack.peek());
    peekedValue = null;
    System.gc();
    assertNull(stack.peek());
}

public class ValueContainer {
    private final String value;

    public ValueContainer(final String value) {
        this.value = value;
    }

    @Override
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.printf("Finalizing for [%s]%n", toString());
    }

    /* equals, hashCode and toString omitted */
}

```

This test demonstrates some quite complex concepts. It is important to note that the reference expected and the reference passed to the stack is different. Had the push to the stack looked like `stack.push(expected)`, a strong reference would have been kept at all times during this test, meaning it would not have been garbage collected, causing the test to fail.

The test inspects the stack with the `peek` method, confirming that the value is on the stack as expected. The `peekedValue` reference is then set to `null`. There are no references to this value, other than inside the `WeakReference` in the stack, so at the next garbage collection, the memory should be reclaimed.

After instructing the JVM to perform a garbage collection, that reference is no longer available in the stack. You should also see a line printed to the standard out saying that the `finalize` method has been called.

USING THIS EXAMPLE STACK

If you were to need this kind of functionality from a stack, calling the `pop` method should decrement the stack pointer until it finds a non-`null` value. Any stack optimizations due to `WeakReferences` being garbage collected have been omitted for brevity.

What is a native method?

Regular Java class definitions are compiled to *bytecode*, held in class files. This bytecode is platform independent, and is translated into specific instructions for the architecture and operating system running the bytecode at run time.

At times, however, you need to run some platform-specific code, perhaps referencing a platform-specific library, or making some operating system-level calls, such as writing to disk or a network interface. Fortunately, the most common cases have been implemented for each platform on which the JVM is available.

For those other times, it is possible to write a native method; that is, a method with a well-defined header in C or C++, identifying the class name, the Java method name, as well as its parameters and return type. When your code is loaded into the JVM, you need to register your native code so that it knows exactly what needs to be run when your native method is called.

What are shutdown hooks?

When the JVM terminates, it is possible to have some code run before it exits, similar to the `finalize` method running before an object is garbage collected. *Shutdown hooks* are references to `Thread` objects; you can add a new reference by calling the `addShutdownHook` method on the current `Runtime` instance. Listing 10-5 shows a simple example.

LISTING 10-5: Adding a shutdown hook

```
@Test
public void addShutdownHook() {
    Runtime.getRuntime().addShutdownHook(new Thread() {
        @Override
        public void run() {
            System.err.println(
                "Shutting down JVM at time: " + new Date());
        }
    });
}
```

The shutdown hook is run on a successful or unsuccessful exit code. Although the code in Listing 10-5 merely logs what time the JVM terminated, you can imagine this facility could be used for notifying a support team of any JVM termination, particularly if that termination is not expected.

SUMMARY

The JVM is an extremely versatile piece of software. It is used in many, many different ways, from desktop applications to online real-time trading systems.

For any non-trivial application, you will need to examine how it interacts with the JVM, and whether it runs as expected under many different conditions. Optimization of a running JVM is often more art than science, and use of the relevant command-line arguments are rarely taught. Read the documentation for the JVM, the `java` command-line application on a Mac or UNIX, or `java.exe` on Windows. Experiment with the command-line arguments; see what effect they have on your program, and try to break your application.

Be aware that different JVM vendors provide different command-line arguments. As a rule, any argument prefixed with `-xx:` is usually a non-standard option, so their usage or even their presence will differ from vendor to vendor.

The next chapter is on concurrency, looking at how Java and the JVM helps to make this as simple as possible. The chapter also covers a new approach to asynchronous execution, using actors.

11

Concurrency

This chapter covers some of the basics of working with *concurrency*, from creating your own threads to asking the JVM to create them for you.

This chapter also introduces the Akka framework. It is a relatively new approach to working with concurrent programming that focuses on creating and processing messages, and allows the framework to manage the processing in parallel.

USING THREADS

How do you run code in parallel?

Java code is run in *threads*. When you start a simple application, such as a traditional Hello World application, the code is run in the *main thread*. As you would expect, an application needs at least one thread to run.

It is possible to create your own threads. Whenever you create a new thread, the code provided to run in that thread will run immediately. Considering that it is not physically possible to run more than one piece of code on a single CPU core at any one time, it is the job of the JVM to manage these threads and schedule which thread to run and when.

Listing 11-1 shows a very simple example of running code in a separate thread.

LISTING 11-1: Running two blocks of code simultaneously

```
public class Threads {  
  
    public static void main(String[] args) throws InterruptedException {  
  
        final Thread separateThread = new Thread(new ThreadPrinter());  
        separateThread.start();  
        for(int i = 0; i < 5; i++) {
```

continues

LISTING 11-1 (*continued*)

```

        System.out.println("From the main thread: "
            + Thread.currentThread().getName());
        Thread.sleep(1000);
    }
}

public class ThreadPrinter implements Runnable {
    @Override
    public void run() {
        for(int i = 0; i < 5; i++) {
            System.out.println("From the new thread: "
                + Thread.currentThread().getName());

            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

To create a new thread, you construct a new `Thread` object. This object takes an instance of the `Runnable` interface. The `Runnable` interface has one method, `public void run()`. Notice that to start the thread, you do not ever call this `run` method. You call `start` on the thread object, which in turn will call `run` once the JVM has created the new thread in order to run the code.

The `Thread` class is special, because it calls out to the JVM to create new threads to allow for parallel execution. If you look at the source code for the `start` method, you will see that it eventually calls a private method called `start0`, which itself is a *native* method: It is implemented in the *Java Native Interface*, specific to the host operating system.

A closer inspection of Listing 11-1 shows that it is doing three main operations: creating a new thread, starting that thread, and then in both threads it is printing the name of the thread every second for five seconds. If you run this code a few times, you will see that the ordering between the two threads may change: Sometimes the printed line from the main thread is first, and sometimes the printed line from the new thread is first. The JVM is deciding which thread to schedule to run.

THE TRY/CATCH BLOCK

Notice that you cannot add a `throws` declaration to the `run` method on the `ThreadPrinter` code in Listing 11-1, so the call to `Thread.sleep` must be surrounded in a `try/catch` block. The `Runnable` interface does not have a `throws` declaration, so neither can any implementing classes.

When the `Callable` interface was introduced in Java 5, the `call` method was defined to throw any exception, so this means that the code can throw an exception itself rather than managing it in a `try/catch` block.

What is the difference between a `Thread` and an `Executor`?

Introduced in Java 5, the concurrency framework provided a set of classes for working with concurrent code, helping you to work with Java's thread model.

Creating a running thread in Java is an expensive operation, and an operating system may limit the number of threads provided to a running application at any one time. By using a *thread pool*, you can create threads when needed, and, rather than creating new threads, you can reuse a thread after it has completed running previous code.

Thankfully, Java's concurrency framework provides a set of thread pools for common use cases, and it is possible to extend this for your own needs. Listing 11-2 shows an example, using the same `ThreadPrinter` class from Listing 11-1.

LISTING 11-2: Using Executors

```
public class UsingExecutor {  
  
    public static void main(String args[]) {  
        final Executor executor = Executors.newCachedThreadPool();  
        executor.execute(new ThreadPrinter());  
        executor.execute(new ThreadPrinter());  
        executor.execute(new ThreadPrinter());  
    }  
}
```

The boilerplate code around creating and starting threads has been removed; this is taken care of in the `Executor` instance for a cached thread pool.

The `Executor` class is an abstraction over parallel computation. It allows concurrent code to be run in a managed way. The interface has one method, `void execute(Runnable)`. Any implementation can manage which threads to run the particular `Runnable`.

You should see that the output of Listing 11-2 logs lines from three different threads. The cached thread pool manages the creation of these threads.

In actuality, `Executors.newCachedThreadPool` does not return an `Executor`, but an `ExecutorService`. The main advantage of the `ExecutorService` is that the parallel computation can return a result, unlike the `Executor`, where the `Runnable` has a `void` return type. An `ExecutorService` also has the ability to be shutdown unlike the `Executors` in Listing 11-2. In fact, the code from Listing 11-2 does not terminate. Listing 11-3 shows an example of how to use an `ExecutorService`.

LISTING 11-3: Calculating pi in a separate thread

```

public class UsingExecutorService {

    public static void main(String[] args) throws InterruptedException,
                                                ExecutionException,
                                                TimeoutException {

        final ExecutorService executorService =
            Executors.newCachedThreadPool();
        final long startTime = System.currentTimeMillis();
        final Future<Double> future =
            executorService.submit(new PiCalculator());

        final double pi = future.get(10, TimeUnit.SECONDS);

        final long stopTime = System.currentTimeMillis();
        System.out.printf("Calculated Pi in %d milliseconds: %10.9f%n",
            stopTime - startTime,
            pi);

        executorService.shutdown();
    }

    private static class PiCalculator implements Callable<Double> {

        public Double call() throws Exception {
            double currVal = 1.0;
            double nextVal = 0.0;
            double denominator = 1.0;

            for(int i = 0;
                Math.abs(nextVal - currVal) > 0.000000001d;
                denominator += 2.0, i++) {
                currVal = nextVal;
                if(i % 2 == 1) {
                    nextVal = currVal - (1 / denominator);
                } else {
                    nextVal = currVal + (1 / denominator);
                }
            }

            return currVal * 4;
        }
    }
}

```

This listing uses the inefficient Leibniz method for generating pi, and uses an `ExecutorService` to run the calculation in a separate thread. The `PiCalculator` class implements the parameterized `Callable` interface, which has one method, `call`, that returns the parameterized value.

Once you have submitted the value, you can perform any other tasks and call `get` on the `Future` as late as possible.

How do you test concurrent code?

Look at Listing 11-4.

LISTING 11-4: A test with a separate thread

```
@Test
public void executorExample() {
    final ExecutorService executor = Executors.newCachedThreadPool();
    final Runnable threadNamePrinter = new InfiniteThreadNamePrinter();

    System.out.println("Main thread: " +
        Thread.currentThread().getName());
    executor.execute(threadNamePrinter);
}

private static class InfiniteThreadNamePrinter implements Runnable {
    @Override
    public void run() {
        while (true) {
            System.out.println("Run from thread: " +
                Thread.currentThread().getName());
        }
    }
}
```

Examining this code, you would be forgiven for saying that this code never terminates. The code prints the thread name forever in a single thread.

However, on running this code, you should see it terminate rather quickly, after maybe 7–10 lines are printed to the console.

In JUnit, when the code in the main thread has completed, the test finishes. In this example, the final line of the `executorExample` method passes the `Runnable` to the `Executor`. When that operation has completed, JUnit assumes that the test is complete. It does not wait for any other threads to complete. This can be a surprise if you have more complicated tests and your expectation is computed in a separate thread.

Even using a `singleThreadedExecutor` rather than a `cachedThreadPool` does not fix this problem. The `singleThreadedExecutor` still runs in a separate thread, though that `Executor` can only have one thread.

Essentially you have two options: You need your `Executor` to run in the same thread, or you need to wait for your thread to complete. Listing 11-5 is an example of waiting for your supporting thread to complete.

LISTING 11-5: Waiting for a supporting thread to complete

```

@Test
public void waitToComplete() throws InterruptedException {
    final ExecutorService executor = Executors.newCachedThreadPool();
    final CountdownLatch latch = new CountdownLatch(1);
    executor.execute(new FiniteThreadNamePrinterLatch(latch));
    latch.await(5, TimeUnit.SECONDS);
}

private static class FiniteThreadNamePrinterLatch implements Runnable {
    final CountdownLatch latch;

    private FiniteThreadNamePrinterLatch(final CountdownLatch latch) {
        this.latch = latch;
    }

    @Override
    public void run() {
        for (int i = 0; i < 25; i++) {
            System.out.println("Run from thread: " +
                               Thread.currentThread().getName());
        }
        latch.countDown();
    }
}

```

This code uses the `CountDownLatch` instance to hold the thread on the final line until the other thread counts down to zero.

Listing 11-6 runs the `Runnable` code in the same thread.

LISTING 11-6: Providing an executor to run code in the same thread

```

@Test
public void sameThread() {
    final Executor executor = new Executor() {
        @Override
        public void execute(final Runnable command) {
            command.run();
        }
    };

    System.out.println("Main thread: " +
                       Thread.currentThread().getName());
    executor.execute(new FiniteThreadNamePrinter());
}

private static class FiniteThreadNamePrinter implements Runnable {
    @Override

```

```

public void run() {
    for (int i = 0; i < 25; i++) {
        System.out.println("Run from thread: " +
            Thread.currentThread().getName());
    }
}
}

```

There is no need to use the `Executors` library; it is simple enough to provide an `Executor` implementation that simply *runs* the `Runnable`. When you run this test, you will see that the `FiniteThreadNamePrinter` code is run in the same thread name as that which kicks off the test.

The beauty with providing your own `Executor` here is that your test is simpler to read, there is no overhead or boilerplate code from anything like the `CountDownLatch` example in Listing 11-5.

If you need to provide `Executors` to your production code as constructor arguments and make them a dependency of the class, you can change the `Executor` implementation for tests with no change to any production code.

WAITING FOR OTHER JOBS TO COMPLETE

Whenever you want to block a thread until another unit of work completes, such as the `CountDownLatch` example in Listing 11-5, you should never block infinitely.

The `await` method, like the `Future.get` method, is overloaded to take either no parameters, or a pair of parameters to determine how long to wait. If you use the option of waiting for a specific time, an `InterruptedException` will be thrown if no response has come in the specified time.

If you use the option with no parameters, the method will block forever. So if the operation you are waiting on has crashed, and will never return, your application will never progress.

Whenever you are waiting on a parallel operation to complete, whether that is in another thread, or, for instance, on a web service to return you some data, you should consider how to react if that operation *never* returns.

WORKING WITH CONCURRENCY

How do you manage shared state between threads?

Examine Listing 11-7.

LISTING 11-7: Sharing state

```

public class SharedState {

    @Test
    public void sharedState() {
        final ExecutorService executorService =
            Executors.newCachedThreadPool();

        final SimpleCounter c = new SimpleCounter();
        executorService.execute(new CounterSetter(c));

        c.setNumber(200);
        assertEquals(200, c.getNumber());
    }

    private static class CounterSetter implements Runnable {
        private final SimpleCounter counter;

        private CounterSetter(SimpleCounter counter) {
            this.counter = counter;
        }

        @Override
        public void run() {
            while(true) {
                counter.setNumber(100);
            }
        }
    }

}

public class SimpleCounter {

    private int number = 0;

    public void setNumber(int number) {
        this.number = number;
    }

    public int getNumber() {
        return number;
    }

}

```

This interesting test creates a new thread, which takes a `Counter` object, and continually sets that value to 100. In the main thread, it takes that same `Counter` object, and sets the value to 200. The test then asserts that the value is indeed 200.

If you run this test a few times, you will find that sometimes it passes, and sometimes it doesn't. Again, this is because of allowing the JVM to schedule when threads are run. If the main thread does the set and get on the counter object before the JVM interrupts that thread to allow the `CounterSetter` to run, the value will still be set to 100.

To make sure that any shared state is not modified while another thread is trying to read or write to it, you must *lock* that shared state. Listing 11-8 improves on this test to make sure the `CounterSetter` cannot interfere with the main thread for the test assertions.

LISTING 11-8: Locking shared state

```
public class LockedSharedState {

    @Test
    public void lockedSharedState() {
        final ExecutorService executorService =
            Executors.newCachedThreadPool();

        final SimpleCounter c = new SimpleCounter();
        executorService.execute(new CounterSetter(c));

        synchronized (c) {
            c.setNumber(200);
            assertEquals(200, c.getNumber());
        }
    }

    private static class CounterSetter implements Runnable {
        private final SimpleCounter counter;

        private CounterSetter(SimpleCounter counter) {
            this.counter = counter;
        }

        @Override
        public void run() {
            while(true) {
                synchronized (counter) {
                    counter.setNumber(100);
                }
            }
        }
    }
}
```

Every Java object has the capability to hold a lock on it. By surrounding code with `synchronized(object)`, only one running thread is allowed to execute inside that block at any one time. The object marked on the `synchronized` block is used as the lock. In Listing 11-8, the actual counter object itself is used.

Any reads and writes involving that counter instance have been surrounded in a `synchronized` block, so while the main thread is updating and then re-reading the counter's value, the `CounterSetter` code cannot interfere with that value. You will find now that the test always passes.

Of course, there is a trade-off for this locking. The threads will need to wait for others while they are locked, so you may see a performance impact. It is therefore advisable to only lock the reads and writes, and no more, releasing the lock as soon as possible.

What do the atomic classes provide?

The Java concurrency framework added the `Atomic` classes, which wrap all of the primitive types, such as `AtomicInteger`, `AtomicBoolean`, and also one for reference types, `AtomicReference`.

These classes all provide guaranteed atomic operations. That is, any other thread will not modify the underlying references backed by these classes until your method call completes. Listing 11-9 has updated the original test from Listing 11-7 to use an `AtomicInteger` as the counter. It is continually setting the counter to the same value. The test manages to get the value from the counter, add one to it, and get that same value, without the `CounterSetter` class able to reset the value during that period.

LISTING 11-9: Using `AtomicIntegers`

```
public class AtomicSharedState {

    @Test
    public void atomicSharedState() {
        final ExecutorService executorService =
            Executors.newCachedThreadPool();

        final AtomicCounter c = new AtomicCounter();
        executorService.execute(new CounterSetter(c));

        final int value = c.getNumber().incrementAndGet();
        assertEquals(1, value);
    }

    private static class CounterSetter implements Runnable {
        private final AtomicCounter counter;

        private CounterSetter(AtomicCounter counter) {
            this.counter = counter;
        }

        @Override
        public void run() {
            while(true) {
                counter.getNumber().set(0);
            }
        }
    }

}

public class AtomicCounter {

    private final AtomicInteger number = new AtomicInteger(0);

    public AtomicInteger getNumber() {
        return number;
    }

}
```

The synchronized blocks are no longer necessary, because the `AtomicInteger` class makes sure that the value is not changed during a read or write, performing the `incrementAndGet` as one operation.

`AtomicIntegers` are often used as counters, rather than using the primitive pre- or post-increment.

USING THE INCREMENT AND DECREMENT OPERATORS

Be aware that when `x++` and `x--` are compiled, the bytecode is three operations: get from memory the value stored in `x`, increment that value by 1, and set the value of `x` back into memory. It is possible for the JVM to interrupt a thread midway through these three operations, and the thread it switches to may also change the value of `x`. If this is the case, then when the JVM switches back to the other thread, the value set back to memory will be invalid, and wrong for the interrupted thread.

Why should you use immutable objects?

In Chapter 8, Listings 8-30 and 8-31 examined how to create immutable objects.

One main advantage for using immutable objects is that, because the values cannot ever change, they can be freely passed from thread to thread without ever needing locking.

This means that, because there is no locking with synchronized blocks, there is no bottleneck where multiple threads need to wait for a single mutable value to be available.

Usually when you have a need to update the value in an immutable object, you make a new copy of the object with the updated value.

ACTORS

What is Akka?

Akka is a framework that provides a different approach to writing and working with concurrent code. It uses an *actor* model, and was inspired by the approach taken from the functional programming language, Erlang. Akka itself is written in Scala, but it also has a complete Java API available. It is available for download from www.akka.io, or can be included in your Maven project as a dependency:

```
<dependency>
  <groupId>com.typesafe.akka</groupId>
  <artifactId>akka-actor_2.10</artifactId>
  <version>2.2.3</version>
</dependency>
```

Actors are designed to process messages, and the Akka framework concentrates on distributing and delivering these messages. This means that you can design your application around the flow of messages rather than having to focus on concurrency with threads and locking.

Listing 11-10 is a simple example.

LISTING 11-10: Using actors in Java

```
public class Message implements Serializable {
    private final String message;

    public Message(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
}

public class MessageActor extends UntypedActor {

    @Override
    public void onReceive(Object message) {
        if (message instanceof Message) {
            Message m = (Message) message;
            System.out.printf("Message [%s] received at %s\n",
                m.getMessage(),
                new Date().toString());
        }
    }

    public static void main(String[] args) {
        final ActorSystem system = ActorSystem.create("actorSystem");
        final ActorRef actorRef = system.actorOf(
            Props.create(MessageActor.class),
            "messageProcessor");

        actorRef.tell(new Message("Hello actor system"), null);
    }
}
```

The `MessageActor` class is an *actor*. Actors implement an `onReceive` method, which is called by the actor system when a message is dispatched to that actor.

The actor is created by the actor system, not by the client code, using the method call `ActorSystem.actorOf`. This allows the system to manage the actor, and keep a reference to it at all times. If you try to create an instance of an actor using a regular `new` constructor call, this will throw an exception.

The actor system itself deals with how to run code. It will often spawn new threads, because that is Java's primitive approach for running concurrent code. Within each thread, Akka will interleave different blocks of execution to give the effect of many concurrent blocks running at once.

What advantage does working with `ActorRefs` instead of `Actors` give?

This level of indirection, letting the actor system create the actor and you dealing with an `ActorRef` object instead, means that should your actor crash, the actor system can create a new actor for a given `ActorRef`. In fact, the client code with the `actorRef` does not even know that the actor has crashed, and can still send messages under the assumption it is running. Listing 11-11 shows this in action, while introducing how to return values to the calling code.

LISTING 11-11: Letting actors crash

```
public class ActorCrash extends UntypedActor {

    public static void main(String[] args) {
        final ActorSystem system = ActorSystem.create("actorSystem");

        final ActorRef crashRef =
            system.actorOf(Props.create(ActorCrash.class));
        final ActorRef dividingActorRef =
            system.actorOf(Props.create(DividingActor.class));

        dividingActorRef.tell(5, crashRef);
        dividingActorRef.tell(0, crashRef);
        dividingActorRef.tell(1, crashRef);
    }

    @Override
    public void onReceive(Object message) {
        System.out.printf("Received result: [%s]%n", message);
    }
}

public class DividingActor extends UntypedActor {
    @Override
    public void onReceive(Object message) {
        if(message instanceof Integer) {
            Integer number = (Integer) message;
            int result = 10 / number;
            this.getSender().tell(result, getSelf());
        }
    }

    @Override
    public SupervisorStrategy supervisorStrategy() {
        return new OneForOneStrategy(
            10,
```

continues

LISTING 11-11 *(continued)*

```

        Duration.Inf(),
        new Function<Throwable, SupervisorStrategy.Directive>() {
            @Override
            public SupervisorStrategy.Directive apply(Throwable t) {
                return SupervisorStrategy.restart();
            }
        });
    }
}

```

This actor system has two classes. When the `DividingActor` receives an integer message, it simply divides ten by the received value. Of course, if zero is received, then this results in an `ArithmeticException`, for division by zero.

By default, the actor is not restarted, although in this case, the `SupervisorStrategy` is defined to always restart.

The code in the `ActorCrash` actor is oblivious to the crash. It simply sends messages to the actor, and is able to receive messages because it is an actor itself. This decoupling of sending and receiving messages allows much flexibility because there is no blocking for a result; the actor's `onReceive` method is simply called when the result is returned.

By developing your application to deal with message processing rather than the imperative approach of call and then block until you receive a response, you can create highly responsive systems.

How can you achieve parallel message processing?

By allowing the actor system to manage and create actors, you can configure it to create multiple actors when necessary. Because actors are message processors, instances of these can usually be considered to be standalone. Consider Listing 11-12.

LISTING 11-12: Processing multiple messages

```

public class MultiActors {

    public static void main(String[] args) {
        final ActorSystem system = ActorSystem.create("actorSystem");
        final ActorRef ref = system
            .actorOf(Props.create(LongRunningActor.class))

        System.out.println("Sending m1");
        ref.tell("Message 1", null);
        System.out.println("Sending m2");
        ref.tell("Message 2", null);
        System.out.println("Sending m3");
        ref.tell("Message 3", null);
    }
}

```

```

    }
}

class LongRunningActor extends UntypedActor {
  @Override
  public void onReceive(Object message) {
    System.out.printf("Being run on ActorRef: %s%n", getSelf());
    try {
      Thread.sleep(10000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.printf("Received %s%n", message);
  }
}

```

This code should give an output similar to the following:

```

Sending m1
Sending m2
Sending m3
Being run on ActorRef: Actor[akka://actorSystem/user/$a#-1672686600]
Received Message 1
Being run on ActorRef: Actor[akka://actorSystem/user/$a#-1672686600]
Received Message 2
Being run on ActorRef: Actor[akka://actorSystem/user/$a#-1672686600]
Received Message 3

```

The output of this code shows that all three messages to send are sent at once, and then each message is processed individually by the same actor instance. Given that the actor takes ten seconds to process each message, this is not the most efficient.

By introducing a *router*, the system can create multiple actors, and can then distribute messages to each of the instances. This all takes place inside the actor system, so your code does not need to manage this at all.

Simply changing the actor creation line to

```

final ActorRef ref = system
    .actorOf(Props.create(LongRunningActor.class)
    .withRouter(new RoundRobinRouter(3)));

```

creates three instances of the actor, and sends messages to each actor in a round-robin approach.

It should provide output similar to the following:

```

Sending m1
Sending m2
Sending m3
Being run on ActorRef: Actor[akka://actorSystem/user/$a/$a#-644282982]
Being run on ActorRef: Actor[akka://actorSystem/user/$a/$c#-1755405169]
Being run on ActorRef: Actor[akka://actorSystem/user/$a/$b#-935009175]
Received Message 1

```

```
Received Message 2  
Received Message 3
```

The `ActorRefs` have new names, and all three messages are being processed at the same time.

By coupling the routing of messages to different actor instances, along with the capabilities to automatically restart, it is possible to build a highly concurrent, fault-tolerant message processing system. Note that the small systems built in this section needed no code for creating threads or managing shared state. By providing immutable objects as messages, and allowing the actor system to create and manage concurrent pieces of work, your own code becomes more streamlined, with less boilerplate, making it easier to understand and maintain.

SUMMARY

Every useful application will run its code in parallel, whether it is a web server or a standalone application with its own user interface.

Understanding the interaction between different blocks of code running in parallel is key, and it is not always clear how and where different threads of execution meet.

In modern web applications, you can often believe that because each request is in its own thread, you do not need to worry about concurrency. This is simply not true. You will have interactions with a database or filesystem. Also, if you use any kind of dependency injection framework, such as Spring or Guice, it is likely that you will create singleton objects for services and data access objects, and these will be shared across all running code in the application.

Try to mitigate the effects of concurrency as much as possible. Always work with immutable objects where possible, even if you aren't going to use them in a concurrent way. Set up your IDE to warn when a variable can be marked as `final`. If it starts to become second nature, these traits will come across well in an interview.

The next section of the book looks at some common libraries and frameworks that most Java developers will use at some point in their careers and will be expected to know and understand. It also covers programming applications to talk to the outside world, either through the Internet on HTTP, or to a database using JDBC or SQL.

PART III

Components and Frameworks

There is more to acing a Java interview than understanding the language. For most roles, knowing the syntax and how to run a Java application is assumed; it is the knowledge of how to use the language to interact, manipulate, and display data.

Many reusable pieces of code, such as algorithms, code to connect to the Internet, or writing to disk have been modularized and abstracted into their own libraries. One of Java's main strengths is that it is mature enough that most common scenarios have their own library, saving you the time and effort of writing and rewriting the functionality from scratch every time. This part focuses on common libraries and application frameworks to help you produce professional, robust applications capable of communicating with the outside world, whether that is by producing a server that displays web pages, or an application that is used to persist information to a database.

Chapter 12 covers databases, introducing the concept of a database store and how Java can interact with the database.

Chapter 13 introduces different web application servers, explaining how to set them up and provide dynamic content to web pages.

Chapter 14 talks about HTTP and REST: how Java applications can communicate with other services over the Internet.

Chapter 15 is about serialization, and how Java objects are represented outside of the JVM.

Chapter 16 covers the Spring Framework, a hugely popular library that helps integrate many components and services, from databases to web applications, while concentrating on testable code.

Chapter 17 covers Hibernate, an object-relational mapper. Hibernate allows your application to convert from Java objects to database tables, and vice versa.

Chapter 18 covers some useful libraries that are often usable in many situations, from helping with I/O to time and date manipulation. You can think of these libraries as valuable functionality that could have been included as part of the standard Java libraries.

Chapter 19 is about the build tools of Maven and Ant. These help manage and integrate your application into a production-ready binary artifact when your application is too large to manage with `javac`.

Chapter 20, the final chapter of this part, takes a brief look at development for Android. Developing mobile phone applications has grown enormously with the rise of powerful smartphones. Although Android phones do not run a JVM, the development is done in Java.

12

Integrating Java Applications with Databases

Object-oriented languages have the capability to model real-world objects in a rich way. Relational databases persist data into tables, which are usually a flatter structure and a more limited representation of these same objects.

This chapter explains how to connect to and work with a relational database, and how to persist Java data types into the database. MySQL is used for all of the examples in this chapter, either from the MySQL console (represented by the `mysql>` prompt), or through Java's interface to the database.

SQL: AN INTRODUCTION

SQL is an abstract, declarative language used to perform queries and data manipulation on a relational database. It is a standard, and not tied to a particular database implementation. Most, if not all, mature relational database products in use today support SQL.

How can you retrieve data from a relational database with SQL?

The data in a relational database is stored across many tables comprised of rows and columns. Each table usually describes a logical *entity*, or *relation*—for instance, a person, an address, or an inventory item. Each row in a database is a record, which can be uniquely identified. The unique identifier is called a *primary key*. Imagine Table 12-1 is an excerpt from a database table for staff records.

TABLE 12-1: The employees Table

EMPLOYEE_ NUMBER	NAME	TITLE	HOME_ ADDRESS	HIRE_ DATE	OFFICE_ LOCATION_ID
1	Bob Smith	CEO	1, Pine Drive	2010-01-25	1
2	Alice Smith	CTO	1, Pine Drive	2010-01-25	1
3	Cassandra Williams	Developer	336, Cedar Court	2010-02-15	2
4	Dominic Taylor	Receptionist	74A, High Road	2011-01-12	2
5	Eric Twinge	Developer	29, Acacia Avenue	2012-07-29	1

You can see that each record can be identified from the `employee_number` column. This table stores only information relating to employees.

SQL gives a flexible language to interpret and manipulate relational database data to provide a representation that is meaningful to your domain. To *query* a particular table, you use the SQL `SELECT` statement:

```
SELECT name, office_location_id FROM employees;
```

This query returns all the rows from the `employees` table, but only the `name` and `office_location_id` columns:

```
mysql> SELECT name, office_location_id FROM employees;
+-----+-----+
| name           | office_location_id |
+-----+-----+
| Bob Smith      | 1                  |
| Alice Smith    | 1                  |
| Cassandra Williams | 2                  |
| Dominic Taylor | 2                  |
| Eric Twinge    | 1                  |
+-----+-----+
5 rows in set (0.00 sec)
```

SQL STATEMENTS ARE CASE INSENSITIVE

As a rule, SQL statements are case insensitive. This applies to the keywords of the statements (such as `SELECT`, `FROM`, and `WHERE`), and also the objects in the query, such as the columns and table names.

One usual convention is to write SQL keywords in uppercase and the objects in lowercase. This helps anyone reading your code to logically break up the query based on the case changes.

You may be interested in retrieving only a specific row, or a certain set of rows. To do this, you can add a `WHERE` clause to the `SELECT` statement. This clause will evaluate to a boolean expression, and will be evaluated for each row. If the expression holds true for the row, it is included in the result set:

```
mysql> SELECT name, office_location_id FROM employees WHERE employee_number > 2;
+-----+-----+
| name           | office_location_id |
+-----+-----+
| Cassandra Williams | 2 |
| Dominic Taylor   | 2 |
| Eric Twinge      | 1 |
+-----+-----+
3 rows in set (0.00 sec)
```

Notice here that the `WHERE` clause was evaluating on a column that was not included in the resulting data set: You can access any column in the query, even if it is not included in the final result.

If you want to return all the columns for a query, you can use the wildcard `*` to represent all columns:

```
SELECT * FROM employees WHERE employee_number = 1;
```

If you add, remove, or otherwise change the columns for the table, this query will still work; however, you will receive different results.

If this table were part of a larger database for a company, you could imagine other tables representing different entities, such as information about all the company offices and who works where, or even a line-management structure. The real appeal and power of databases lie in the interaction between these relations.

Table 12-2 defines another table for this company database, called `office_locations`.

TABLE 12-2: The `office_locations` Table

OFFICE_LOCATION_ID	LOCATION_NAME	ADDRESS	LEASE_EXPIRY_DATE
1	New York	Union Plaza	NULL
2	Paris	Revolution Place	NULL
3	London	Piccadilly Square	2014-05-30

Although this information is logically separate from the `employees` table, there is definitely a relationship between the two. If you were to select from both tables:

```
mysql> select employees.name, office_locations.location_name
-> from employees, office_locations;
+-----+-----+
| name           | location_name |
+-----+-----+
| Bob Smith      | New York      |
| Bob Smith      | Paris         |
| Bob Smith      | London        |
+-----+-----+
```

```
| Alice Smith      | New York |
| Alice Smith      | Paris    |
| Alice Smith      | London   |
| Cassandra Williams | New York |
| Cassandra Williams | Paris    |
| Cassandra Williams | London   |
| Dominic Taylor    | New York |
| Dominic Taylor    | Paris    |
| Dominic Taylor    | London   |
| Eric Twinge       | New York |
| Eric Twinge       | Paris    |
| Eric Twinge       | London   |
+-----+-----+
15 rows in set (0.00 sec)
```

This data isn’t quite as expected. What has happened here? The database isn’t quite sure how the rows fit together, so it took each row from the `employees` table and paired it with each row from the `office_locations` table. For a more meaningful result, you must inform the database of how the tables fit together, as shown in Listing 12-1.

LISTING 12-1: Joining two tables

```
SELECT *
FROM employees
JOIN office_locations
  ON employees.office_location_id = office_locations.office_location_id;
```

This query returns the expected data set:

```
mysql> select employees.name, office_locations.location_name
-> from employees
-> join office_locations
-> on employees.office_location_id = office_locations.office_location_id;
+-----+-----+
| name      | location_name |
+-----+-----+
| Bob Smith  | New York      |
| Alice Smith | New York      |
| Cassandra Williams | Paris        |
| Dominic Taylor | Paris        |
| Eric Twinge | New York      |
+-----+-----+
5 rows in set (0.00 sec)
```

Performing a query over two or more tables is known as a *join*. There is an alternative syntax for performing joins. You can simply list the tables in the `FROM` part of the statement and join the tables in a `WHERE` clause:

```
SELECT *
FROM employees, office_locations
WHERE employees.office_location_id = office_locations.office_location_id;
```

This may not seem like a particularly big difference, but the original query, known as an *ANSI Join*, has a clear separation for the relationship. If that query had a *WHERE* clause, then all the parts of that clause would have been specifically for filtering rows, and not for defining the relationship between the tables.

The `office_location_id` column on the `employees` table references the column with the same name in the `office_locations` table. This is known as a *foreign key*, and is the term given to a column referencing another table. Informing the database of foreign key relationships is optional; however, it does help to optimize queries because it will expect queries to be joined and filtered on that specific column.

Defining foreign keys also enables you to enforce *referential integrity*: It is not possible to have a value for an `office_location_id` in the `employees` table that does not exist in the `office_locations` table. Conversely, it is not possible to delete rows from `office_locations` while there is still an entry for that location in the `employees` table: An employee must have a valid office to work in.

Omitting the foreign key relationship when defining tables means there is no referential integrity between the tables: The referencing column can contain data that is not referenced in a joined table. Sometimes this is what you want.

What are inner and outer joins?

By default, rows from both tables that meet the matching criteria are included in the join. It is possible to specify that all the rows from one or either table from a join are included, filling a missing match with `NULLS`.

The `office_locations` table includes some rows that are not referenced at all in the `employees` table. This implies that, for some reason, the company has an office that is currently empty. The results of Listing 12-1 show no mention of that location.

In the `JOIN` clause of the query, if you specify `LEFT OUTER JOIN` instead, the result will include all rows from the left side of the query, filling in `NULLS` for no matches on the right, and vice versa for `RIGHT OUTER JOIN`. Listing 12-2 shows a query with a right outer join for matching the `employees` and `office_locations`.

LISTING 12-2: A right outer join

```
SELECT *
FROM employees
RIGHT OUTER JOIN office_locations
ON employees.office_location_id = office_locations.office_location_id;
```

This query returns an entry for every `location_name`, regardless of whether there is an employee based at that location:

```
mysql> select employees.name, office_locations.location_name
-> from employees
-> right outer join office_locations
```

```
-> on employees.office_location_id = office_locations.office_location_id;
+-----+-----+
| name          | location_name |
+-----+-----+
| Bob Smith     | New York      |
| Alice Smith   | New York      |
| Eric Twinge   | New York      |
| Cassandra Williams | Paris        |
| Dominic Taylor | Paris        |
| NULL          | London        |
+-----+-----+
6 rows in set (0.00 sec)
```

It is also possible to perform a left and right join simultaneously; filling in NULLs from *both* tables where no match occurs, with the syntax `FULL OUTER JOIN`. This is not applicable to the join between `employees` and `office_locations`, because `office_location_id` is a foreign key relationship: It is not possible to have an employee working in a location that does not exist in the database.

Figure 12-1 visualizes the difference between the joins on two tables.

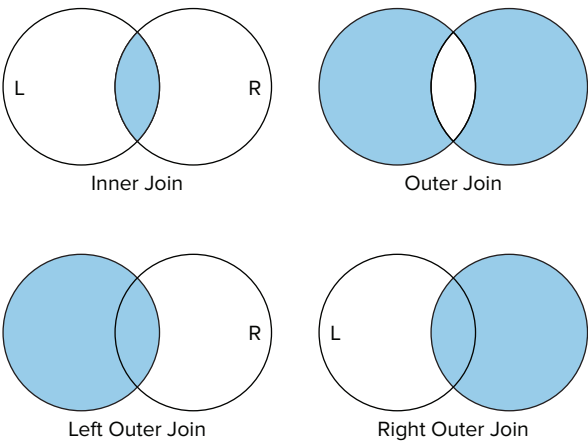


FIGURE 12-1

Does SQL have the capability to perform analysis of the data in a database?

SQL includes many functions that enable you to perform calculations on columns. If you have any experience of working with a relational database, more than likely you will have used the `COUNT` function to count the number of rows in a table:

```
SELECT COUNT(*) from employees;
```

This query returns the number of rows in the `employees` table.

Table 12-3 defines a table for employees’ salary details.

TABLE 12-3: The salaries Table

EMPLOYEE_NUMBER	ANNUAL_SALARY	LAST_UPDATE_DATE
1	20000	2013-06-29
2	12000	2013-06-29
3	6000	2012-06-10
4	6000	2013-06-16
5	12000	2013-05-26

The `SUM` function adds all the rows for a specified column. Applying this to the `annual_salary` in Listing 12-3 would allow the finance team to calculate their annual sum spent on wages.

LISTING 12-3: Calculating the sum total for a column

```
mysql> select sum(annual_salary) from salaries;
+-----+
| sum(annual_salary) |
+-----+
|          56000 |
+-----+
1 row in set (0.00 sec)
```

You can also use these functions to perform calculations on a certain column, but grouping common rows together. For instance, if you want to see the average salary for each role in the company, you need to use the `AVG` function, and group your data by the employees' titles, as shown in Listing 12-4.

LISTING 12-4: Using the GROUP BY clause

```
mysql> select e.title, avg(s.annual_salary)
-> from employees e
-> join salaries s
-> on e.employee_number = s.employee_number
-> group by e.title;
+-----+-----+
| title          | avg(s.annual_salary) |
+-----+-----+
| CEO            | 20000.0000 |
| CTO            | 12000.0000 |
| Developer      | 9000.0000 |
| Receptionist   | 6000.0000 |
+-----+-----+
4 rows in set (0.00 sec)
```

By including the title column, you need to specify the `GROUP BY` clause; otherwise the database is not sure how to collate the data for each of the different titles. Omitting that clause produces nonsense data, calculating the average for the annual salary column and including the original first row for the title column, CEO.

NOTE *The collection of tables and columns used in the examples here includes some long, verbose names. Listing 12-4 uses aliases for the tables, which allows you to rename tables for the purpose of a query, in this case to a shorter name.*

If you ever join a table on to itself, you will need to create an alias for each table so that the database understands which exact instance of the table you are referring to when joining or using a `WHERE` clause.

How do you persist data to a database?

The uses of SQL so far have only looked at how to examine data already present in tables. You can insert data into a table using an `INSERT` statement, modify current rows using `UPDATE`, and remove rows using `DELETE`.

An `INSERT` statement contains the data to be inserted into a particular table:

```
INSERT INTO employees VALUES (1, "Bob Smith", "CEO", "1, Pine Drive", CURDATE(), 1);
```

You can insert multiple rows this way, too, by providing a comma-separated list of row tuples:

```
INSERT INTO employees VALUES
(2, "Alice Smith", "CTO", "1, Pine Drive", CURDATE(), 1),
(3, "Cassandra Williams", "Developer", "336, Cedar Court", CURDATE(), 2),
(4, "Dominic Taylor", "Receptionist", "74A, High Road", CURDATE(), 2),
(5, "Eric Twinge", "Developer", "29, Acacia Avenue", CURDATE(), 1);
```

It is also possible to insert partial rows, as long as the constraints on the table allow for the omitted columns to be `NULL`, or have a default value. You need to specify the columns to update:

```
INSERT INTO employees (employee_number, name, title, office_location_id)
VALUES (6, "Frank Roberts", "Developer", "2");
```

For updating data, you specify which column to change, the new value, and optionally, the matching criteria for which rows to update:

```
UPDATE employees
set home_address = "37, King Street"
where employee_number = 6;
```

If you use the primary key to match the rows, by definition you are only updating one row, because the primary key uniquely identifies a row. Also, the table will have an index against the primary key, so the database should be able to locate the row to update instantly.

If you do not provide a `WHERE` clause on an `UPDATE` statement, the update is applied to all rows.

Deleting rows also takes a `WHERE` clause, specifying which rows to remove:

```
DELETE FROM employees WHERE name = "Frank Roberts";
```

Be careful! If you omit a `WHERE` clause, all rows are deleted:

```
mysql> select count(*) from employees;
+-----+
| count(*) |
+-----+
|          5 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> delete from employees;
Query OK, 5 rows affected (0.00 sec)
```

Whenever you are changing data in a table, you should always use a transaction. This will enable you to `ROLLBACK` any mistakes before you commit the transaction.

What is a view?

As a convenience, it is possible to capture a particular query or join that is used often and present that as a “virtual table,” known as a *view*. Views can often provide help to developers or operations staff who are running several queries directly on a database console.

If a query were used often to find employees and their work addresses, the following statement would create a relevant view:

```
CREATE VIEW employees_and_locations AS
SELECT employee_number, name, location_name, address
FROM employees
JOIN office_locations
ON employees.office_location_id = office_locations.office_location_id;
```

You would be able to select from this view as if it were a regular table:

```
mysql> select * from employees_and_locations;
+-----+-----+-----+-----+
| employee_number | name           | location_name | address           |
+-----+-----+-----+-----+
| 1 | Bob Smith | New York | Union Plaza |
| 2 | Alice Smith | New York | Union Plaza |
| 3 | Cassandra Williams | Paris | Revolution Place |
| 4 | Dominic Taylor | Paris | Revolution Place |
| 5 | Eric Twinge | New York | Union Plaza |
+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```

It is possible to delete rows from a view. Of course, the usual rules about referential integrity apply. You can insert rows, too. Any columns missing from the view definition will have a `NULL` value applied. If the column is not allowed to have a `NULL` value, it is not possible to insert into that particular view.

As a rule of thumb, it is advisable not to delete from views. Views are so lightweight that it is very easy to change their definition, which can have knock-on effects on inserting `NULL`s when a column definition is removed from a view.

What are DDL and DML?

So far the examples discussed have only been relevant to data manipulation. SQL is also used for creating, removing, and altering tables. *Data Manipulation Language* (DML) uses the `SELECT`, `INSERT`, `UPDATE`, and `DELETE` keywords to manipulate data. *Data Definition Language* (DDL) is used to create and manipulate table structures.

Tables are created with a `CREATE TABLE` statement. The statement contains a list of columns and the data types for each column:

```
CREATE TABLE meeting_rooms (
    meeting_room_id    INT,
    office_location_id INT,
    meeting_room_name  VARCHAR(100)
);
```

This creates a table with three columns, two of type `INT` and one of type `VARCHAR(100)`, a string of up to 100 characters. Be aware that the valid data types can vary between different databases. You will need to check the up-to-date documentation from your database's vendor for which types are allowed.

Although there is a column called `office_location_id`, there is no reference to the `office_locations` table; there is no referential integrity over the data allowed in this column.

As you should expect, DDL is able to do more than create tables. It can change the table definitions too, such as adding a column:

```
ALTER TABLE meeting_rooms ADD COLUMN telephone_extension VARCHAR(100);
```

Performing a `SELECT` on the `meeting_rooms` table displays that column as if it was there all along. If the table already had data, the column would be blank, with a `NULL` value. You can specify a default value on the `ALTER TABLE` statement.

To remove, or *drop* a column, run the statement as:

```
ALTER TABLE meeting_rooms DROP COLUMN telephone_extension;
```

You can also modify the definition of a column in place. For instance, you can add a foreign key constraint to the table:

```
ALTER TABLE meeting_rooms
ADD FOREIGN KEY (office_location_id)
REFERENCES office_locations(office_location_id);
```


Once this constraint is added, only valid `office_location_ids` will be allowed in that column for the `meeting_rooms` table. The statement will be rejected if there is already data in the column that does not meet the constraint.

You can add arbitrary constraints, too, such as defining a column as not allowing `NULL` values, or allowing only a certain range of integer values.

How can you speed up an inefficient query?

Looking at the original `employees` table, the records will be stored sequentially on disk, most likely in the order of the `employee_number` column, so employee number 101 will follow 100, and so on.

For most purposes, storing the data in this fashion does not make much sense; very few cases exist when you would want to list the employees in the order of their employee number.

One possible query over this data would be to find all the employees who work in a particular office:

```
SELECT * FROM employees WHERE office_location_id = 2;
```

This query will need to examine every row sequentially to see if the `WHERE` predicate matches. If the table has several thousand rows, this query could take a while, especially if no rows meet the criteria.

By adding an index for this column, the database will provide a direct lookup for that column; giving a direct reference to the row locations. Figure 12-2 illustrates how this works.

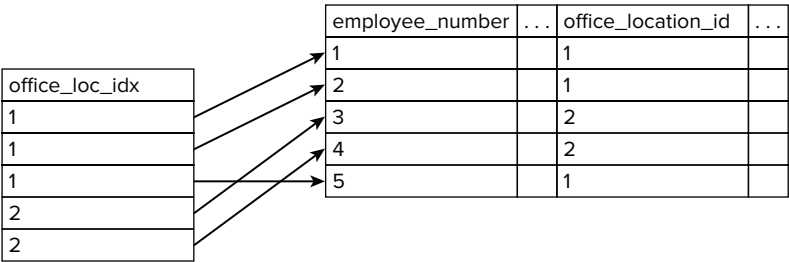


FIGURE 12-2

The grouping of `office_location_ids` will be stored alongside this table, and the database will use this to look up rows based on office location. The trade-off is that more disk space is needed to store the index. Listing 12-5 shows how to create this index.

LISTING 12-5: Creating an index

```
create index office_loc_idx on employees (office_location_id);
```

Once in place, an insert into the `employees` table will also result in an entry in the index, possibly making insertion time longer. The trade-off is that lookups will be much quicker, especially once the table has grown to several thousand rows in size.

What can you do when SQL is not enough?

Stored procedures give you the opportunity to do more than the simple create, read, update, and delete functionality that SQL provides. You have the ability to make more procedural steps.

One such example could be to create a procedure that grants pay raises to employees who have not had a pay increase for more than a year. Listing 12-6 shows how you could implement this in MySQL.

LISTING 12-6: Creating a stored procedure

```
CREATE PROCEDURE annual_salary_raise (
    IN percent INT,
    OUT total_updated INT
)
BEGIN
    SELECT COUNT(*)
    INTO total_updated
    FROM salaries
    WHERE last_update_date > DATE_SUB(CURDATE(), INTERVAL 1 YEAR);

    UPDATE salaries
    SET annual_salary = annual_salary * ((100 + percent) / 100),
        last_update_date = CURDATE()
    WHERE last_update_date > DATE_SUB(CURDATE(), INTERVAL 1 YEAR);
END;
```

The procedure takes two parameters: an `IN` parameter, which is a value for the percentage pay increase; and an `OUT` parameter, which you can think of as a return value from the procedure. You pass a handle here, the procedure sets its value, and you can read from that handle after you have run the procedure. In this case, the `OUT` parameter holds the number of rows updated. You can have as many `IN` and `OUT` parameters as you require.

The `SELECT` statement also takes a new form, applying the result of the `COUNT` into the `total_updated` `OUT` parameter. The query will fail if your `SELECT` statement returns zero or more than one row.

The procedure applies the percentage increase to those rows that have not been updated for more than a year. MySQL provides a `DATE_SUB` function for manipulating `DATE` data types—in this case, the function calculates the date a year ago from `CURDATE`, which is today's date. The update statement also sets the updated date to `CURDATE`, so that the updated rows will not be updated for another year. Listing 12-7 shows this procedure in action on the MySQL console, applying a 4 percent pay raise.

LISTING 12-7: Calling MySQL stored procedures

```
mysql> select curdate();
+-----+
| curdate() |
+-----+
| 2013-06-29 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> select * from salaries;
+-----+-----+-----+
| employee_number | annual_salary | last_update_date |
+-----+-----+-----+
| 1 | 20000 | 2013-06-29 |
| 2 | 12000 | 2013-06-29 |
| 3 | 6000 | 2012-06-09 |
| 4 | 6000 | 2013-06-15 |
| 5 | 12000 | 2013-05-25 |
+-----+-----+-----+
5 rows in set (0.00 sec)
```

```
mysql> call annual_salary_raise(4, @updated_count);
Query OK, 1 row affected (0.05 sec)
```

```
mysql> select * from salaries;
+-----+-----+-----+
| employee_number | annual_salary | last_update_date |
+-----+-----+-----+
| 1 | 20000 | 2013-06-29 |
| 2 | 12000 | 2013-06-29 |
| 3 | 6240 | 2013-06-29 |
| 4 | 6000 | 2013-06-15 |
| 5 | 12000 | 2013-05-25 |
+-----+-----+-----+
5 rows in set (0.00 sec)
```

```
mysql> select @updated_count;
+-----+
| @updated_count |
+-----+
| 1 |
+-----+
1 row in set (0.00 sec)
```

Stored procedures are useful for “batching up” series of statements that affect the data in some bespoke way, like the procedure shown here. Another use is in *triggers*. When certain database events happen, such as removing or updating a row, you can configure procedures to be run before, after, or instead of that action.

One practical use for triggers is in logging database events. Whenever a sensitive table is updated, such as financial records like an employee’s salary, you can set a procedure to run that inserts the data as it was before and after the update into an auxiliary audit table, which can be examined at a later date for any discrepancies.

What are transactions?

A problem with the stored procedure presented in Listing 12-6 is that it is possible for the number of updated rows returned in the `OUT` parameter to be different than the actual number of updated rows. The statement performs two actions: First, it gathers a count of the number of rows that will be updated, and then it performs that update. If a client runs the `annual_salary_raise` procedure, and at the same time another client updates the `salaries` table with data that will be affected by the update statement, it is possible that the second client's update will happen after the first client's `SELECT` to count the rows and the `UPDATE` to change the salary information.

By making the procedure *transactional*, this *race condition* can be avoided.

A transaction, in any domain, whether a database or not, needs to comply with four properties, collectively known as *ACID*:

- **Atomic**—The contents of the transaction are done as a whole, or not at all. It is not possible to view the database in a state where only half of the transaction is applied.
- **Consistent**—After the transaction, the database meets all the requirements for your domain. For instance, when making a payment between two parties, the amount debited from one party should match the amount credited to the other.
- **Isolated**—Transactions running in parallel should each behave as though they are the only transaction running on the database.
- **Durable**—Once the transaction has been committed, it is committed permanently. Any subsequent database change, such as an update or a crash, will have no effect on the committed transaction.

To make the stored procedure in Listing 12-6 transactional, you must open a transaction with `START TRANSACTION`, and successfully commit the transaction with `COMMIT`, as shown in Listing 12-8.

LISTING 12-8: Using transactions

```
CREATE PROCEDURE annual_salary_raise_transactional (
    IN percent INT,
    OUT total_updated INT
)
BEGIN
    START TRANSACTION;
    SELECT COUNT(*)
    ...
    ... <AS LISTING 12-6>
    ...
    WHERE last_update_date < DATE_SUB(CURDATE(), INTERVAL 1 YEAR);
    COMMIT;
END;
```

When this transaction is run, any other connection trying to update the `salaries` table will be blocked until the transaction completes.

It is also possible to abort a transaction yourself, if, while running a stored procedure, you find your database in an inconsistent state, or perhaps if some criteria for your procedure haven't been met. You abort transactions with the `ROLLBACK` statement.

A rollback is usually managed with a database log, and all log entries are reversed from the start of the transaction. It is possible that other running transactions are dependent on the result of a transaction that is rolled back—in this situation, those dependent transactions would be rolled back too.

What is NoSQL?

NoSQL is an umbrella term given to databases that do not necessarily follow the relational database model. They often rely on denormalized data and are stored as key-value pairs, whole documents, or graphs in order to make retrieval as quickly as possible, to make applications responsive, particularly when working with huge amounts of data, such as terabytes or even petabytes of storage.

There is no SQL equivalent standard for access to a NoSQL database, as each database stores data in its own way. One drawback to this is that you are often tied in to a particular vendor once you start to use a particular NoSQL database, as migration will be a bespoke and laborious process, requiring you to map between the models of different databases.

Some popular NoSQL databases are:

- **MongoDB**—Data is stored as JSON objects, and may not share all of the same fields as other similar objects.
- **Cassandra**—Data is stored in *column families*, similar to tables, but rows do not need to share the same columns.
- **Memcached**—A distributed key-value store cache
- **Redis**—A clustered, persistent key-value store

JDBC: COMBINING JAVA AND THE DATABASE

How do you use Java to connect to a relational database?

Java Database Connectivity, or *JDBC*, is the mechanism built into the standard Java libraries to connect to a database. When using JDBC to connect to your database, you must make sure that the vendor's JDBC implementation is on the classpath. Before Java 6, you needed to make sure that your database vendor's driver for JDBC had been loaded by the classloader, by calling `Class.forName("<Vendor class>")`. This has now been automated with the service provider mechanism, which provides a lookup inside the `META-INF/services` directory of the JAR.

To get a connection, you need to provide JDBC with the relevant connection parameters, in the form of a URL, as shown in Listing 12-9.

LISTING 12-9: Confirming an active database connection

```
@Test
public void connectToDb() throws SQLException {
    final Connection connection = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/company_db", "nm", "password");
    assertFalse(connection.isClosed());
    connection.close();
    assertTrue(connection.isClosed());
}
```

The connection string to `DriverManager.getConnection` passes in a full URL to the database, called `company_db`, which is running on a local server, on the default port of 3306, with the username `nm` and the password `password`. The test confirms it can create the connection, verifies it is connected, and then closes the connection.

How do you use Java to perform SQL queries?

Once a connection has been established, you will want to perform SQL operations to retrieve and manipulate data. Listing 12-10 shows a simple query against that database, populating the results into regular Java types.

LISTING 12-10: Querying a database with JDBC

```
@Test
public void retrieveRows() throws SQLException {
    final Connection connection = DriverManager
        .getConnection(
            "jdbc:mysql://localhost:3306/company_db",
            "nm",
            "password");
    final Statement stmt = conn.createStatement();
    final ResultSet rs = stmt.executeQuery(
        "select employee_number, name from employees");

    final Map<Integer, String> employeeMap = new HashMap<>();

    while (rs.next()) {
        final int employeeNumber = rs.getInt("employee_number");
        final String name = rs.getString("name");
        employeeMap.put(employeeNumber, name);
    }

    final Map<Integer, String> expectedMap = new HashMap<Integer, String>() {{
        put(1, "Bob Smith");
    }}
```

```

        put(2, "Alice Smith");
    }
}

assertEquals(expectedMap, employeeMap);

rs.close();
stmt.close();
}

```

A database connection provides statements, and statements run queries, which return result sets.

The `ResultSet` interface has an `Iterator`-like approach. While the `ResultSet` instance is open, the database has an open cursor, and each call to `next` moves the cursor on to the next row. The method returns a boolean, returning false if there are no more rows from the query.

Each database vendor provides its own implementation of the JDBC classes, such as `Statement`, `ResultSet`, and `Connection`. As a result, it is up to the vendor to provide a mapping from the database data types to the Java types. Ordinarily, this does not pose much of a problem, because most data types have a logical equivalent. In the case of Listing 12-10, the `employee_number` column is an `Integer` in the MySQL database, and so calling `getInt` on the `ResultSet` will convert to a Java `int`. Similarly, for the name column, this is a `VARCHAR`, which can be mapped to a `String`.

The `getXXX` methods are overloaded, and can take either a string, representing the column name, or an `int`, representing the column ordering, indexed from 1.

How can you avoid SQL injection attacks?

If you have a specific query that is run repeatedly, the database can assist in the query preparation, which will result in a faster, and actually more secure, query.

The use of *query parameters* allows the database to compile and prepare the query, and then you can execute the actual query by providing the parameters. Listing 12-11 shows how this can work.

LISTING 12-11: Using query parameters

```

@Test
public void queryParameters() throws SQLException {
    final PreparedStatement pstmt = conn.prepareStatement(
        "insert into office_locations values (?, ?, ?, NULL)");

    final Map<String, String> locations = new HashMap<String, String>() {{
        put("London", "Picadilly Square");
        put("New York", "Union Plaza");
        put("Paris", "Revolution Place");
    }};

    int i = 1;
    for (String location : locations.keySet()) {
        pstmt.setInt(1, i);
        pstmt.setString(2, location);
    }
}

```

continues

LISTING 12-11 *(continued)*

```

        pstmt.setString(3, locations.get(location));
        pstmt.execute();
        i++;
    }

    pstmt.close();

    final Statement stmt = conn.createStatement();
    final ResultSet rs = stmt.executeQuery(
        "select count(*) from office_locations " +
        "where location_name in ('London', 'New York', 'Paris')");
    assertTrue(rs.next());
    assertEquals(3, rs.getInt(1));

    rs.close();
    stmt.close();
}

```

In Listing 12-10, the query was performed using a `Statement` object, but in this case, the query uses a `PreparedStatement` object. When you create a `PreparedStatement`, you need to provide the query you want to run. Parameters that you intend to vary over the different query invocations are presented as question marks.

For each query execution, you set the parameters using `setXXX` methods, identifying the type of the data for the query. You can see that this is the mirror of the `getXXX` methods on the `ResultSet`. Again, be aware that the parameters are indexed from 1, not from zero.

Using the `setXXX` methods with a particular data type rather than hard-coding the parameter values as a `String` gives a certain level of safety from attacks such as SQL injection.

Imagine you had a web service that displayed the contents of a database table, depending on the ID of a certain column. You could expect the URL to look something like `/showResults?id=3`. A simple implementation of this could take the parsed query parameter, and pass it straight to the database in a query:

```
stmt.executeQuery("select * from records where id = " + queryParameters.get("id"));
```

However, a malicious user could replace the `id` parameter value with `3; DROP TABLE users;`. This would have the previous code run two queries:

```
SELECT * FROM records WHERE id = 3; DROP TABLE users;
```

If you are unlucky enough to have a table called `users`, it will have just been deleted.

Instead, if this query had been executed with a `PreparedStatement`:

```

final PreparedStatement ps = conn.prepareStatement("select * from records where id = ?");
ps.setInt(1, queryParameters.get("id"));
ps.execute();

```


when the query parameter attempts to set the value to an `int`, it will fail with an exception at run time, because the malicious string is not an `int`.

You should always sanitize your database inputs, and you should always use the type system where you can.

How do you run stored procedures from Java?

Calling stored procedures follows a similar pattern to that of creating prepared statements. You can run the previous `salary_update` procedure using the test in Listing 12-12.

LISTING 12-12: Calling a stored procedure

```
@Test
public void callStoredProc() throws SQLException {
    final CallableStatement stmt = conn.prepareCall("{call annual_salary_raise(?, ?)}");
    stmt.setInt(1, 4);
    stmt.registerOutParameter(2, Types.INTEGER);

    stmt.execute();

    int updatedEmployees = stmt.getInt(2);
    assertEquals(1, updatedEmployees);
}
```

For this test, it is assumed that before running, there is only one employee in the database who is eligible for a pay raise, as per the data in Table 12-3.

A `CallableStatement` is created with the `prepareCall` on the `Connection` object. Similar to the `PreparedStatement` object, the statement to run on the database is presented here, with the parameters set as question marks.

All the different parameters are set, along with their data types. Note here that JDBC differentiates the output parameters. These are collected after the statement has been executed.

The string to call the stored procedure is database-specific:

```
{call ANNUAL_SALARY_RAISE(?, ?)}
```

The structure of this string varies from vendor to vendor. Be aware that the way you initiate the call can be different between the vendors. For instance, a call to a stored procedure of the same name in Oracle PL/SQL would look like this:

```
conn.prepareCall("BEGIN ANNUAL_SALARY_RAISE(?, ?); END;");
```

The call is surrounded by a `BEGIN...END` block instead of curly braces. The parameters are still set by question marks.

How can you manage database connections?

A connection to the database is a limited resource and the process of creating a connection is extremely time-consuming, especially when the connection must be negotiated over a network. Quite often, the creation of a database connection can take longer than the operation on the database itself.

In a similar approach to working with thread pools, applications often manage database connection pools.

Several open source libraries create and work with connection pools. They all follow a similar approach: You provide the library with details of how to make a connection, and when you require a connection, the library provides it. It will either create one for you, or reuse a previously created connection, sparing you the time taken to set it up.

One open source project for managing connections is C3P0. Listing 12-13 shows an example of how to acquire a connection.

LISTING 12-13: Using a connection pool

```
@Test
public void connectionPools() throws SQLException {
    final ComboPooledDataSource cpds = new ComboPooledDataSource();
    cpds.setJdbcUrl("jdbc:mysql://localhost:3306/company_db");
    cpds.setUser("nm");
    cpds.setPassword("password");

    final Connection conn = cpds.getConnection();
    final Statement stmt = conn.createStatement();
    final ResultSet rs = stmt.executeQuery(
        "select count(*) from office_locations " +
        "where location_name in ('London', 'New York', 'Paris')");
    assertTrue(rs.next());
    assertEquals(3, rs.getInt(1));

    DataSourcees.destroy(cpds);
}
```

You do not close the connection when you have finished. The connection pool keeps a reference to any connections it creates, and periodically checks the health of the connection by running a simple query, such as `SELECT 1` or `SELECT 1 FROM DUAL`, depending on the database.

This has the added benefit of making your application more resilient to a database outage. Should your database crash and restart, all your connections are lost, so any references to those connections are invalid. When your application makes new requests for connections, brand new connections will be created against the restarted database, because the pool will verify that the old connections are stale.

How can you manage database releases for your application?

When releasing a Java application, generally you build and deploy the application as a single unit, and any upgrades are also performed as a whole entity, replacing any existing binary.

This is not always possible for databases, because the data stored in the database must be preserved across releases.

Luckily, several tools are available that integrate well with current build systems for managing and performing database deployments and upgrades.

DBDeploy is one such tool, and works directly with Maven and Ant (if you are unfamiliar with these tools, look at Chapter 19: “Developing with Build Tools”).

Every time you want to change your database in an upcoming release, you add a numbered file to your build, numbered one higher than your last database change file. DBDeploy manages a table on your database, holding a list of the run scripts in a table called `changelog`. If DBDeploy finds any files that have not been recorded in the `changelog` table, it will run those in numerical order, as shown in Listing 12-14.

LISTING 12-14: Running DBDeploy with new database changes

```
update-database:
[dbdeploy] dbdeploy 3.0M3
[dbdeploy] Reading change scripts from directory /javainterviews/chapter12...
[dbdeploy] Changes currently applied to database:
[dbdeploy]      (none)
[dbdeploy] Scripts available:
[dbdeploy]      1, 2
[dbdeploy] To be applied:
[dbdeploy]      1, 2
[dbdeploy] Applying #1: 001_database_creation.sql...
[dbdeploy] Applying #2: 002_stored_proc.sql...
[dbdeploy] -> statement 1 of 2...
[dbdeploy] -> statement 2 of 2...
```

Any re-run of DBDeploy will ignore those files, because they will be recorded as having been successfully applied, as shown in Listing 12-15.

LISTING 12-15: Running DBDeploy with no new database changes

```
update-database:
[dbdeploy] dbdeploy 3.0M3
[dbdeploy] Reading change scripts from directory /javainterviews/chapter12...
[dbdeploy] Changes currently applied to database:
[dbdeploy]      1, 2
[dbdeploy] Scripts available:
[dbdeploy]      1, 2
[dbdeploy] To be applied:
[dbdeploy]      (none)
```

TESTING WITH IN-MEMORY DATABASES

Is it possible to maintain a consistent environment for developing against a database?

When doing active development against an application that needs access to a database, keeping the database in a maintainable state can sometimes be problematic. One approach is to use a different database distribution than the one you use in your production and integration environments: an in-memory database.

As you would expect, the contents of an in-memory database are lost whenever the JVM shuts down. This makes them useful for working with test data, where you don't actually care about the persistence of the data, but more that the interaction between the application and the database works as expected. The in-memory database is run on your local machine too, so there will be no network traffic to manage and the database does not need to deal with complicated connection management from multiple clients.

A few open-source relational databases have an in-memory option, which are SQL-compliant. H2 is one such database. It has been developed in such a way that you need no special lines of code to get the database running. Listing 12-16 shows just how simple it is to create and connect to an H2 database.

LISTING 12-16: Creating and using an in-memory database

```
@Test
public void connectToH2() throws SQLException {
    final Connection conn = DriverManager.getConnection(
        "jdbc:h2:mem:test;MODE=MySQL", "nm", "password");
    final Statement stmt = conn.createStatement();
    stmt.executeUpdate("create table teams (id INTEGER, name VARCHAR(100))");
    stmt.executeUpdate("insert into teams values (1, 'Red Team')");
    stmt.executeUpdate("insert into teams values (2, 'Blue Team')");
    stmt.executeUpdate("insert into teams values (3, 'Green Team')");

    final ResultSet rs = stmt.executeQuery("select count(*) from teams");
    assertTrue(rs.next());
    assertEquals(3, rs.getInt(1));
}
```

The connection string `jdbc:h2:mem:test` is enough for H2 to create an in-memory database, called `test`, which will act like any other JDBC connection. The `teams` table is available only while the test is running—it is not stored anywhere, so this test will run successfully every time. When connecting to a live database, you would need to find out the current state of the `teams` database before running any tests against the table. The username and password parameters are irrelevant.

You can customize the behavior of the database by applying some settings as part of the connection string. The string `jdbc:h2:mem:test;MODE=MySQL` indicates that H2 should understand the MySQL

specifics; that is, the MySQL data types and any other SQL and language differences that are specific to the MySQL platform.

By coupling this with DBDeploy, before your test suite runs, you can load your DBDeploy scripts into a fresh H2 database, and your tests will act as if they are connecting to a regular MySQL database.

Another approach is to run tests on a real relational database inside an open transaction, and roll-back after the test is run, on success or failure. This can be an effective, but risky way to test. If what you are testing commits a transaction, then you cannot do it this way. You are also dependent on the database being in the correct state.

SUMMARY

Java has a well-understood, and well-used, interface between Java objects and their representation in a database. This chapter has tackled some of the hurdles around connecting to and manipulating databases, including regular, vanilla SQL, and more expressive stored procedures.

Although SQL is the standard for database querying and manipulation, database distributions from separate vendors still behave differently: They use their own data types, they have their own extensions, and they even approach the use of stored procedures differently.

Any interview for an experienced Java developer will most likely include an element of database interaction. Quite often, applications in industry, particularly in finance, concentrate the majority of their efforts on the database and its performance. A Java application is then a presentation of that database. It is important to understand the database you are connecting to—understanding JDBC and SQL is not enough.

The next chapter complements this one by looking at how to use Java to produce web applications. More often than not, web applications are backed by some kind of persistent data store. For more about the interaction between Java and databases, you can jump to Chapter 17, which looks at object-relational mapping with Hibernate.

13

Creating Web Applications

This chapter covers Tomcat and Jetty, two of the main applications used to serve web pages, and it also covers a more recent framework, Play. Both Tomcat and Jetty implement the Java Servlet API. This API is part of Java Enterprise Edition, a platform covering many applications and services often used in developing software for the enterprise. As well as the Servlet API, this includes topics such as Java's Messaging Service and some advanced topics around database transaction management and persistence.

Tomcat and Jetty are quite similar, in that they both follow the Servlet API. In fact, some teams often use both within their development cycle for creating a single application: using Jetty for active day-to-day development, and using Tomcat for deployment for use with multiple users, whether that's a QA for testing or live production deployment.

The relevant sections in this chapter try to differentiate between Tomcat and Jetty, and where they can be appropriately used.

Play is a newer application server inspired by other frameworks, such as Rails, that concentrates on rapid prototyping and fast development, iterating development on top of a barebones, but functional, application.

TOMCAT AND THE SERVLET API

Java's Servlet API defines a set of interfaces and file definitions for building web applications for use on the JVM.

How are web applications defined within the Servlet API?

The Servlet API defines that a web application uses a *deployment descriptor* named `web.xml`, which is located on the classpath at `/webapp/WEB-INF/web.xml`. This file defines the *servlets*, and how they are configured and served by a *servlet container*.

The `web.xml` file is used to determine which servlet handles a certain request, based on the path of the request. Listing 13-1 shows an example deployment descriptor holding multiple servlets.

LISTING 13-1: An example `web.xml` deployment descriptor

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  metadata-complete="false"
  version="3.0">

  <filter>
    <filter-name>HeaderFilter</filter-name>
    <filter-class>com.example.HeaderFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>HeaderFilter</filter-name>
    <servlet-name>AdminConsole</servlet-name>
  </filter-mapping>

  <servlet>
    <servlet-name>Application</servlet-name>
    <servlet-class>com.example.ApplicationServlet</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>AdminConsole</servlet-name>
    <servlet-class>com.example.AdminConsoleServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>AdminConsole</servlet-name>
    <url-pattern>/admin/*</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>Application</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>

</web-app>
```

This descriptor has two main parts: the filter definitions and mappings, and the servlet definitions and mappings.

Examining the servlet definitions and mappings first, this descriptor has two servlets, the classes `com.example.ApplicationServlet` and `com.example.AdminConsoleServlet`. These classes are implementations of `javax.servlet.http.HttpServlet`.

These servlets are mapped to URL patterns. If a client makes a request to anything on `/admin`, that request will be handled by the `AdminConsoleServlet`. The `ApplicationServlet` handles any other request.

Listing 13-2 defines a very simple implementation of `HttpServlet`.

LISTING 13-2 AN EXAMPLE HTTP SERVLET

```
public class TimeServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        final String now = new Date().toString();

        response.setContentType("text/html");
        response.setStatus(HttpServletResponse.SC_OK);

        final String responseString = String.format("<html>" +
            "<head>" +
            "<title>Time</title>" +
            "</head>" +
            "<body>The time is %s</body>" +
            "</html>", now);

        response.getWriter().print(responseString);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}
```

`HttpServlet` provides Java methods to be called for the corresponding HTTP methods. If a GET request is forwarded to this servlet, the `doGet` method will be called. Similarly, a POST request would be forwarded to `doPost`. The servlet in Listing 13-2 has been defined to handle POSTS and GETS in the same way. If you were to make a request using a different HTTP method, say, PUT, the response code would be a HTTP 405 – Method Not Allowed, or HTTP 400 – Bad Request. These codes are the default behavior on `HttpServlet` if you do not override the appropriate `doXXX` method.

THE HTML RESPONSE OF LISTING 13-2

Listing 13-2 responds directly with a chunk of HTML advising the requestor of the current time on the server. Explicitly writing the response in this manner is not advisable, because it can become hard to manage and maintain very quickly as your application grows.

Libraries are dedicated to managing text manipulation for dynamic responses inside HTML. Additionally, it is good practice to separate the business logic from how those results are displayed. The *Model-View-Controller (MVC)* pattern is a popular implementation to take care of this, and several frameworks implement this pattern for handling the interaction between the data and its display. Spring's implementation, Spring MVC, is covered in Chapter 16.

How can requests to a particular servlet be securely managed?

The deployment descriptor from Listing 13-1 also defines a filter, and it maps this filter to the `AdminConsole` servlet. Classes defined inside a `filter-class` tag implement the `javax.servlet.Filter` class. Listing 13-3 is such an implementation.

LISTING 13-3: Filtering HTTP requests

```
public class HeaderFilter implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        // do nothing
    }

    @Override
    public void doFilter(ServletRequest request,
                        ServletResponse response,
                        FilterChain chain) throws IOException, ServletException {
        if (request.getParameter("requiredParameter") != null) {
            chain.doFilter(request, response);
        } else {
            ((HttpServletResponse) response).sendError(
                HttpServletResponse.SC_UNAUTHORIZED);
        }
    }

    @Override
    public void destroy() {
        // do nothing
    }
}
```

The `Filter` class defines three methods—`init`, `doFilter` and `destroy`—and these methods are called at appropriate times during the lifetime of the application server. `init` is called when the filter is first initialized, `doFilter` is called before every request, and `destroy` is called when the server is shutting down.

For this example, no additional setup is needed once the class is instantiated, so the `init` method does nothing. The `FilterConfig` parameter holds additional metadata such as the name for this filter in the deployment descriptor (in the `filter-name` tag).

The `destroy` method gives the filter the opportunity to do any necessary housekeeping, such as closing connections and cleaning file systems, before it is destroyed for good.

The `doFilter` is called on every request, and it gives the filter the opportunity to inspect the request, and decide whether to reject the request or allow it to pass before it is forwarded to the servlet for execution.

It is possible to have more than one filter defined for a servlet, known as a *filter chain*; the order of the `filter-mapping` tags in the deployment descriptor defines the order of the chain for execution.

When a filter has finished processing the request, it should forward the request to the next filter in the chain. Calling `doFilter` on the `FilterChain` object parameter does this. Omission of this call inside your filter means the request will not be processed by the servlet. Figure 13-1 illustrates the filter chain.

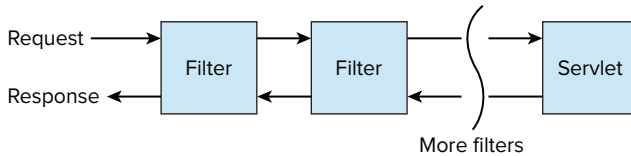


FIGURE 13-1

The example in Listing 13-3 inspects the request's query parameters, and if `requiredParameter` is not set, the client is returned the status code `HTTP 401 - Unauthorized` and the request is not forwarded to the servlet. Naturally, checking the existence of a required parameter is not a secure approach to authenticating clients of a web application, but it should serve to explain how the filter is operating.

Note that the `Filter` interface is not part of the `javax.servlet.http` package, and the request and response parameters passed to `doFilter` are `ServletRequest` and `ServletResponse`, not the `HttpServletRequest` and `HttpServletResponse` subclasses. This `Filter` interface has been designed for use with any kind of request/response servlet, not just HTTP servlet containers. The `sendError` method is only on the `HttpServletResponse` class, which is why a cast is necessary to return an error to the client.

It is safe to perform this cast here, because this filter is only going to be used inside an HTTP servlet container for servicing HTTP requests.

Filters are not just for determining if a request is to be handled. They can provide additional value to requests. Some examples of filter use include:

- **Logging all requests**—The filter logs the request and immediately passes the request to the next filter in the chain.
- **Authentication**—A stronger form than that described in Listing 13-3: If a user is not authenticated, the request could be redirected to a login page first.
- **Serving static content**—Images or other content such as JavaScript or CSS files could be loaded without the servlet needing to be called.

What is a WAR file?

A *Web Archive* file, or WAR, file is built in the same way as Java's JAR files. It is specifically used for bundling up web applications. It contains all the compiled class files needed to run the application, as well as any provided properties and other configuration files.

Most importantly, a WAR file contains the deployment descriptor—the `web.xml` file—which is the main definition of how the web application is set up. It serves as instructions to the application server of how to deploy and serve the application.

Most IDEs such as IntelliJ and Eclipse can create WAR files within a couple of clicks for a given project. If you want to automatically build and deploy WARs, you will need your build tool to create them for you. For Maven, setting the top-level `packaging` tag to `war` will instruct your build to create a WAR artifact over a JAR. If Maven is new to you, it may help to fast-forward to Chapter 19 on build tools to understand how to build Java projects using Maven.

How are web applications managed within Tomcat?

Tomcat is designed to run more than one web application at a time. A web application is defined as a WAR file that contains a deployment descriptor, `web.xml`.

Usually a server will be running one Tomcat instance, and the relevant WAR files are installed under the `webapps` directory of the Tomcat instance. Naturally, this is a default choice, and it is possible to configure this location.

TOMCAT'S COMPONENTS

Tomcat is built from several components, including one for the servlet container, and one for processing JSPs. The servlet container component is called *Catalina*, and the JSP component is called *Jasper*.

Be aware that when reading documentation for Tomcat usage, it is not uncommon to see references to these names—for example, the default name for the output log file is `catalina.out`, or the environment variable for Tomcat's home directory is called `$CATALINA_HOME`.

By default, a running Tomcat instance is listening for any changes to the `webapps` directory. Should a new WAR file be saved to this directory, Tomcat will extract it into a directory matching the WAR filename, and serve the application on the context matching that name.

For example, if you deployed a WAR file called `BookingApplication.war` to the `webapps` directory, this would be extracted to a directory called `BookingApplication`. Tomcat would then read the `web.xml` hosted under `WEB-INF` and serve the application from this definition. The application would be served as `/BookingApplication` to clients.

This approach is extremely useful for rapid turnarounds on development. You can include the deployment in a continuous delivery pipeline, automatically deploying a built and unit-tested WAR file to an accessible server, ready to be integration tested. You could even deploy it automatically to a live environment, if your test coverage is good enough and your team's development process allows it.

How do you handle request parameters with the Servlet API?

The `HttpServletRequest` class has a method, `getParameterMap`. This method returns a map of each pair of parameters that formed the request.

Given the key may appear more than once on an HTTP request, the type of the map is `Map<String, String[]>`. That is, each key has an array for all the values that were part of the request with that same key.

Consider the following code:

```
final StringBuffer output = new StringBuffer();

final Map<String, String[]> parameterMap = request.getParameterMap();
for (String parameterName : parameterMap.keySet()) {
    output.append(
        String.format("%s: %s<br/>",
            parameterName,
            Arrays.asList(parameterMap.get(parameterName))));
}

IOUtils.write(output, response.getOutputStream());
```

If you were to send a request with the query string `a=123&b=asdf&a=456`, then this would produce the output:

```
b: [asdf]<br/>a: [123, 456]<br/>
```

Notice how the values for the `a` key were collated, as that key appeared more than once.

As this functionality is part of the Servlet API, it is available to any application container that supports the Servlet API. This includes Tomcat, and Jetty.

JETTY

Jetty has been designed to be lightweight and easy to use. You can even create and run servers purely in Java code, with no configuration.

How can you create a running web application inside a Java class?

You can configure Jetty to run in several lines of code. Listing 13-4 shows two classes: one to start a server and the other to define how that server responds to requests.

LISTING 13-4: Defining a Jetty server in code

```
public class JettyExample {

    public static void main(String args[]) throws Exception {
        final Server jettyServer = new Server(8080);
        jettyServer.setHandler(new EchoHandler());
        jettyServer.start();
    }
}
```

continues

LISTING 13-4 *(continued)*

```

        jettyServer.join();
    }
}

class EchoHandler extends AbstractHandler {

    @Override
    public void handle(String target,
                      Request baseRequest,
                      HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException {
        response.setContentType("text/plain");
        response.setStatus(HttpServletResponse.SC_OK);
        baseRequest.setHandled(true);

        final ServletInputStream requestInputStream =
            request.getInputStream();
        final ServletOutputStream responseOutputStream =
            response.getOutputStream();

        IOUtils.copy(requestInputStream, responseOutputStream);
    }
}

```

This code configures a server to respond to any client with the same body of the request. The HTTP POST and PUT methods have bodies, so requests with these methods will respond appropriately.

PERFORMING HTTP REQUESTS

A web browser is fine for performing requests, but to make a POST, PUT, or anything other than a GET request will require you to write a small amount of HTML, and perhaps use the help of a JavaScript library such as JQuery.

You will find that using a dedicated HTTP client will make it easier to create and debug HTTP requests. UNIX operating systems (including Mac OS X) ship with the command-line client *curl*. The following line demonstrates its use for the server created in Listing 13-4:

```
> curl -X PUT -d "Hello World" http://localhost:8080
```

This will respond with the same “Hello World.” It will not have a trailing newline.

For a graphical HTTP client, the Google Chrome plug-in called Postman has a rich feature set and an intuitive user interface.

Chapter 14 covers using HTTP clients within Java code, and also looks at the HTTP methods.

The final line of the `handle` method in the `EchoHandler` is a utility method from the Apache Commons IO library for connecting `InputStreams` to `OutputStreams`. The Apache Commons library is covered in Chapter 18.

Extending the `AbstractHandler` class is a Jetty-specific implementation, and though this may be fine for your needs, it also ties your code to the Jetty platform. Rather than using Jetty's handlers, you can implement your specific behavior as a servlet. Listing 13-5 shows two classes: the Jetty configuration in code and the implementation of the `HttpServlet` class.

LISTING 13-5: Using Jetty with the Servlet API

```
public class JettyServletExample {

    public static void main(String[] args) throws Exception {
        final Server jettyServer = new Server(8080);
        final ServletHandler servletHandler = new ServletHandler();
        jettyServer.setHandler(servletHandler);
        servletHandler.addServletWithMapping(EchoServlet.class, "/*");
        jettyServer.start();
        jettyServer.join();
    }
}

public class EchoServlet extends HttpServlet {

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/plain");
        response.setStatus(HttpServletResponse.SC_OK);

        final ServletInputStream requestInputStream =
            request.getInputStream();
        final ServletOutputStream responseOutputStream =
            response.getOutputStream();

        IOUtils.copy(requestInputStream, responseOutputStream);
    }
}
```

Jetty's API has a specific handler for dealing with servlets. This means that it is possible to use both Jetty-specific handlers and servlet implementations in the same code. The following snippet shows an example:

```
final HandlerList handlers = new HandlerList();
handlers.setHandlers(new Handler[] { jettyHandler, servletHandler });
jettyServer.setHandler(handlers);
```

Note that this implementation of the same functionality is more restricted than the implementation using only Jetty's handlers: it will only respond to `POST` requests. To perform the same action for another method, say `PUT`, the `doPut` method should be overridden too:

```

@Override
protected void doPut(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    doPost(request, response);
}

```

You can see from the main method that different servlets can be mapped to different contexts, much like you would do inside a `web.xml` deployment descriptor:

```

servletHandler.addServletWithMapping(AppServlet.class, "/app");
servletHandler.addServletWithMapping(AdminServlet.class, "/admin");

```

How can you perform integration tests against a running web server?

The simplicity and flexibility of Jetty and its capability to configure and run within Java code makes it an ideal candidate as a server for use within integration tests, even if your production servlet container is different. Listing 13-6 is a JUnit test running against the `EchoServlet` defined in Listing 13-5.

LISTING 13-6: Testing servlets with Jetty and JUnit

```

public class JettyJUnitTest {

    private static Server jettyServer;

    @BeforeClass
    public static void startJetty() throws Exception {
        jettyServer = new Server(8080);
        final ServletHandler servletHandler = new ServletHandler();
        jettyServer.setHandler(servletHandler);
        servletHandler.addServletWithMapping(EchoServlet.class, "/echo/*");
        jettyServer.start();
    }

    @Test
    public void postRequestWithData() throws IOException {
        final HttpClient httpClient = new DefaultHttpClient();
        final HttpPost post = new HttpPost("http://localhost:8080/echo/");
        final String requestBody = "Hello World";
        post.setEntity(new StringEntity(requestBody));

        final HttpResponse response = httpClient.execute(post);
        final int statusCode = response.getStatusLine().getStatusCode();

        final InputStream responseBodyStream = response.getEntity().getContent();
        final StringWriter stringWriter = new StringWriter();

        IOUtils.copy(responseBodyStream, stringWriter);
        final String receivedBody = stringWriter.toString();

        assertEquals(200, statusCode);
    }
}

```



```

        assertEquals(requestBody, receivedBody);
    }

    @AfterClass
    public static void stopJetty() throws Exception {
        jettyServer.stop();
    }
}

```

A new server is started when this test suite is created, and the server is stopped once all the tests are run. For this implementation, the server holds no state, so it is not necessary to create a new server for each test run. Should you require a new server for each test run, the server could be started in a `@Before` annotated method.

Note that inside the test method, there are no references to Jetty, nor any code related to the server implementation; this is only testing the behavior of the server over the HTTP protocol. Apache's HTTP client API is used here, and again, Apache's Commons IO API simplifies reading the response `InputStream` and converting to a string.

Although only one test has been defined here, you can certainly think of more test cases for this servlet, such as different HTTP methods, or making requests with no data.

How can you perform active development against a local application server?

Jetty has a Maven plug-in, which allows for easy running of web applications built from a Maven project.

You can set the Maven Jetty plug-in for the project by adding the plug-in to the project's POM:

```

<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>9.0.3.v20130506</version>
</plugin>

```

You invoke the plug-in by running `mvn jetty:run`. This will compile and package the project first. By Maven's main rule of following convention over configuration, the plug-in will look in the default location for the deployment descriptor: `/src/main/webapp/WEB-INF/web.xml`. Aside from defining the plug-in in your POM, you do not need any Jetty-specific configuration; the presence of this file will be enough for Maven to load a Jetty server with the configured servlets. Listing 13-7 loads the previous `EchoServlet`.

LISTING 13-7: Running Jetty through Maven

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=

```

continues

LISTING 13-7 *(continued)*

```

        "http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
        metadata-complete="false"
        version="3.0">

<servlet>
  <servlet-name>Echo</servlet-name>
  <servlet-class>
    com.wiley.javainterviewsexposed.chapter13.EchoServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Echo</servlet-name>
  <url-pattern>/echo/*</url-pattern>
</servlet-mapping>

</web-app>

```

Running `mvn jetty:run` loads the server on your local host, on port 8080 by default. The port configuration is not part of the `web.xml` deployment descriptor, so if you require your server to run on a different port, this is specified by the `jetty.port` property: Running `mvn -Djetty.port=8089 jetty:run` starts it on port 8089.

Another huge advantage to this approach is that you need no bespoke software to get a fully functional application server running locally. Aside from Java and Maven, you do not need to download and install any other applications from the Internet. Any newcomers to your team can simply check out your code, and run `mvn jetty:run`.

Any specific application settings are set inside the `plugin` tag in your POM. For example, to set some specific system properties, the configuration in your POM would look like Listing 13-8.

LISTING 13-8: Setting specific properties for the maven-jetty-plugin

```

<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>9.0.3.v20130506</version>
  <configuration>
    <systemProperties>
      <systemProperty>
        <name>environment</name>
        <value>development</value>
      </systemProperty>
    </systemProperties>
  </configuration>
</plugin>

```

Using the Maven Jetty plug-in in this fashion is often a popular choice for active development. You can have a local web server running quickly with minimal setup needed, and the server is running with the same deployment configuration inside the `web.xml`, which will be deployed to the live production environment, perhaps even on a different application server, such as Tomcat.

The use of system properties to determine which environment you are running against enables you to provide different configurations for different environments at run time, with no code changes needed to switch between the environments.

PLAY FRAMEWORK

Play is a relatively new player in the web application space. It takes a fresh approach to developing web applications, and does not make use of Java's Servlet API.

How do you create a new Play application?

Play has a different approach to developing web applications. Rather than creating a new project yourself, using Maven or similar, you ask Play to create it for you.

After downloading and unzipping Play, you run the `play` UNIX script or `play.bat` Windows batch file from a fresh empty directory, instructing it to create a new project. You also provide the name of the project:

```
> play new echo-server
```

This will take you through a very short process to create the project for you. Play asks you for the name of the project. You can keep it as `echo-server` for this project:

```
What is the application name?  
> echo-server
```

You then have the option of creating a new Java or Scala project, or a completely blank project. Choose the Java project:

```
Which template do you want to use for this new application?  
  
1 - Create a simple Scala application  
2 - Create a simple Java application  
3 - Create an empty project  
  
> 2
```

This has created a skeleton project for you, in a directory called `echo-server`.

Before inspecting the code, you can check that the application actually runs. The `play` application works slightly differently compared to more traditional applications you may have come across, such as Maven or Ant. Inside the `echo-server` directory, running `play` with no parameters takes you to

a console, which enables you to interactively specify build steps you want to run. On execution, you should see a prompt and some welcoming text similar to the following:

```
> play
[info] Loading global plugins from /Users/noel/.sbt/plugins
[info] Loading project definition from /playProjects/echo-server/project
[info] Set current project to echo-server (in build file:/playProjects/echo-server)

  _ _ _ _ _
 |'_ _ \ | | /'_ _ \ | | | |
 | | _ / | | \___/ \___/ ( )
 | |

play! 2.0, http://www.playframework.org

> Type "help play" or "license" for more information.
> Type "exit" or use Ctrl+D to leave this console.

[echo-server] $
```

At this prompt, you can type `run`. This may take some time while Play downloads dependencies, but eventually you should see confirmation that the application is running:

```
[info] play - Listening for HTTP on port 9000...

(Server started, use Ctrl+D to stop and go back to the console...)
```

At this point, you can visit `http://localhost:9000` in your favorite web browser. The page displayed should be a boilerplate page informing you that the server is up and running, with some links for next steps.

Pressing `Ctrl+D` in your Play console stops the server and takes you back to the command prompt.

What are the different components of a Play application?

Figure-13-2 shows the default layout of a Play application.

The `app` directory contains two subdirectories: `controllers` and `views`. This should be familiar if you are aware of the Model-View-Controller pattern. The `controllers` directory is where you will place Java classes for dealing with processing incoming requests. The `views` directory holds templates for displaying the results of requests.

The `conf` directory is where you store configuration for your web application: `application.conf` is a traditional Java properties file, containing key-value pairs. This is where you would configure any logging for your application, as well as database configuration. You can also specify your own properties here, which will be available to the running application.

The `routes` file instructs Play what to do with incoming requests. The file contains three columns. The first defines the HTTP method. The second is the path of the request. If Play receives a request that matches these two entries, it will execute the method defined in the third column. Listing 13-9 shows an example entry in a `routes` file.

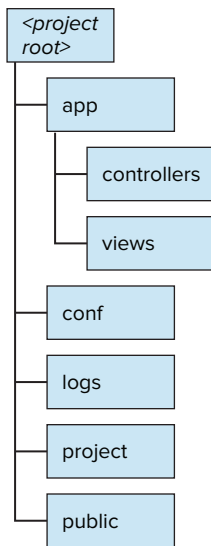


FIGURE 13-2

LISTING 13-9: An example entry in Play's routes file

```
GET      /time                               controllers.Application.time()
```

When a client makes a `GET` request to the `/time` path, the `time()` method on the `Application` class will be called.

The `logs` directory contains any logging output from your application. You have the ability to control the logging from the `conf/application.conf` file. The setup is the same as the standard Log4J format.

The `project` directory contains any build-specific setup, such as any necessary library dependencies, or specific build steps, such as deploying builds to an artifact server like Nexus or Artifactory.

The `public` directory contains any static files to be served, such as images, JavaScript, or CSS. Note that access to these assets is still controlled through the `routes` file, and is included in the default project setup.

Remember, Play applications are not designed to work with Java's Servlet API; Play does not create WAR files. When your project is complete and ready to be run outside of the `play` console, you can run the `dist` command. This will create a new top-level directory called `dist`, and put a zipped build of your application there. Inside the zip is a `start` script. Running this script will start your server in the same fashion as calling `run` inside the `play` console.

How is Java code executed as a result of an HTTP request?

The `controllers` directory underneath `app` is where any Java classes will be placed. This is actually a top-level package, so if you are more comfortable with your organization's package structure here, you use that here.

For the new application created earlier in this section, you will need a small amount of code to mimic an echo server. Create a new class, called `Echo`, which extends `play.mvc.Controller`. This class will have one method, `echoReply`, which takes no parameters. Listing 13-10 contains this method.

LISTING 13-10: The Play controller for an echo service

```
public class Echo extends Controller {

    public static Result echoReply() {
        RequestBody messageBody = request().body();
        return ok(messageBody.asText());
    }
}
```

The `Result` object generated by the `ok` method contains all the necessary metadata for Play to understand how to respond to the client. The `ok` method specifies the HTTP 200 – OK response code. It is a static import from the `Results.Status` class, and as you would expect, this contains methods for most response codes: `notFound()` for HTTP 404, `internalServerError()` for HTTP 500, and so on.

This method is extracting the message body as text, due to the `messageBody.asText()` method call. This expects the request to have the `Content-Type: text/plain` HTTP header set. This `RequestBody` object can create different types for the message body depending on the `Content-Type` header, such as a rich JSON object for requests coming in as `application/json`, or even an XML DOM Document for those requests received as `text/xml`.

You need to instruct Play to call this method when a valid request comes in. Listing 13-11 instructs the server to respond with this echo whenever a POST or PUT is called on `/echo`.

LISTING 13-11: The routes for the echo service

```
POST    /echo    controllers.Echo.echoReply()
PUT     /echo    controllers.Echo.echoReply()
```

As you are executing plain Java code, you do not need to do anything special to test it: You can simply write unit and integration tests as usual, using JUnit or something similar.

How is the Java code separated from the HTML generation with Play?

The echo service created is an extremely useful approach for creating applications like Web Services and REST APIs, though if you are creating web pages to be viewed by a human being in a web browser, you will still need an approach to make them legible.

Rather than generating an HTML response by hand, Play has tight integration with its own template framework. You can do the hard processing inside the regular Java class, and then hand off to the template to display your response in a pleasing way.

Extend the `Echo` class to process requests containing query parameters. These parameters will be parsed by the controller, and then passed to the template to generate the HTML. Listing 13-12 shows the new method on the `Echo` class, Listing 13-13 shows the new entry in the routes file, and Listing 13-14 is the template for displaying the output as HTML.

LISTING 13-12: The echo service controller for displaying request parameters

```
public static Result echoParameters() {
    final Map<String, String[]> queryString = request().queryString();

    final String timestamp = new Date().toString();

    return ok(paramPrintout.render(queryString, timestamp));
}
```

LISTING 13-13: The new routes entry

```
GET      /echo/parameters      controllers.Echo.echoParameters()
```

LISTING 13-14: The template for displaying the request parameters

```
@(queryString: Map[String, Array[String]])(timestamp: String)

<html>
  <head><title>Query Parameters</title></head>
  <body>
    <h1>The request parameters for this page:</h1>
    <ul>
      @for((param, values) <- queryString) {
        <li>
          @param : @for(value <- values) { @value }
        </li>
      }
    </ul>
    This page was generated at: @timestamp
  </body>
</html>
```

The template file is called `paramPrintout.scala.html`. This name is important, because during the process of building the site, Play will convert this template into compiled Scala code, an instance called `paramPrintout`.

Scala code is used here, because the Play Framework itself is written in Scala. Don't be alarmed if you've not seen any Scala code before, this template language is similar to many of the other template languages in use, such as Freemarker or Velocity.

If you launch this application using the `run` command from within the `play` console, and you visit the URL `http://localhost:9000/echo/parameters?a=123&b=456&a=abc`, then hopefully you can see how this request fits into the human-readable response.

If you are particularly skilled with HTML, JavaScript, and CSS, you can see how Play would make a useful back end allowing rapid turnaround for development.

How can you use Play to interact with a JDBC database?

Inside the `application.conf` file, you configure a set of properties for working with a database. For simplicity, this setup is with an in-memory database, which was covered in Chapter 12.

The properties `db.default.driver` and `db.default.url` should be set if you use a database:

```
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:testDatabase"
```

Inside your controller, you can get a regular JDBC connection with the static method call `play.db.DB.getConnection()`. Once you have a reference to the `java.sql.Connection` object, which this method returns, the code for accessing and manipulating database objects is exactly the same, as shown in Listing 13-15.

LISTING 13-15: Accessing a JDBC database inside a Play application

```
final Connection conn = play.db.DB.getConnection();
final Statement stmt = conn.createStatement();
stmt.executeQuery("select * from ...")
```

Remember, this is just a regular Java class, so you can use any libraries that you are comfortable with to help with database access, such as Spring's `JdbcTemplate` (covered in Chapter 16) or even something more advanced, such as an object-relational mapper like `Hibernate` (covered in Chapter 17).

SUMMARY

This chapter has provided a tour of the different flavors of HTTP servers available to Java developers. The Java Servlet API has been in existence almost as long as Java itself, just as the Internet and applications on the World Wide Web were starting to take off.

By looking at the Play framework as well, this should show that using Java servlets is not the only way to create web servers on the JVM. The uses of libraries and technologies is always changing, and hopefully by taking a look at Play, you should understand that staying on top of the current trends and applications in use is imperative to staying competitive in the job market.

A large number, if not the majority, of jobs available to Java developers will include some kind of web component. Companies tend toward developing products for their clients with a web interface, and every desktop has a web browser: Companies can get their product to their clients without the need for creating custom installers, or creating and delivering a boxed product. Be aware that if you are going to have an interview with a company that has any kind of presence on the Internet—for instance, an online shopping site, or even a social gaming company—any reputable employer will test you on your understanding of how a Java application communicates with the outside world.

The next chapter continues using Play for servicing HTTP requests, taking a deeper look at the HTTP methods and how they relate to the popular REST services, as well as how to call out as a client to other HTTP services.

14

Using HTTP and REST

HTTP is one of the core ways for requesting and receiving data over the Internet. It is used for communication over the World Wide Web. HTTP servers, such as Apache's HTTPD, Microsoft's IIS, and Tomcat, are configured to listen for structured incoming requests, and respond appropriately.

THE HTTP METHODS

What is HTTP?

A good exercise in understanding how the HTTP protocol works is to perform an HTTP request outside of a browser using an application such as Telnet which is provided with most modern operating systems.

To make a request to a site, such as Wikipedia, open a Telnet connection to port 80. You should see output similar to Listing 14-1.

LISTING 14-1: Initiating an HTTP request using Telnet

```
> telnet en.wikipedia.org 80
Trying 91.198.174.225...
Connected to wikipedia-lb.esams.wikimedia.org.
Escape character is '^]'.
```

Your command prompt should be on a blank new line. The server hosting your request is now waiting for you to make your request for a certain page. Type **GET / HTTP/1.1**, followed by new lines. Your Telnet session should respond with a set of HTTP headers, followed by some HTML, and will then close the connection and take you back to the command line, like in Listing 14-2.

LISTING 14-2: Creating an HTTP request using Telnet

```
GET / HTTP/1.1

HTTP/1.0 200 OK
Server: Apache
X-Content-Type-Options: nosniff
Cache-Control: s-maxage=3600, must-revalidate, max-age=0
Last-Modified: Mon, 13 May 2013 17:07:15 GMT
Vary: Accept-Encoding
Content-Length: 6071
Content-Type: text/html; charset=utf-8
Connection: close

<!DOCTYPE html>
<html lang="mul" dir="ltr">

... WEB PAGE CONTENT ...

</body>
</html>Connection closed by foreign host.
>
```

Typing `GET / HTTP/1.1` instructs the server of three key instructions: *Use* the GET method, *get* the page `/`, and *use* version 1.1 of the HTTP protocol. This example sent no HTTP headers to the server; any headers in the form `Key: Value` would have been included on the lines underneath the GET line. These headers often include metadata about your request, such as how you would like the format of the response, or what browser you are using. This is why you need a completely blank line to complete the request, because the server will be expecting many lines for the request.

The server responds in a similar fashion. The initial line, `HTTP/1.0 200 OK`, informs the client that the response is using version 1.0 of the protocol (this server is clearly backward compatible, to accept a version 1.1 request, and reply with a 1.0 response), and informs the client that the request resulted in a `200 OK` response, which is a successful response.

The following lines are the response headers, which contain some metadata about the response. These can help clients make judgments about how to parse the body of the response, indicating details such as the language of the body and its length.

Again, a blank line signifies the end of the headers, and the start of the body of the response. For a rich HTML web page such as the one requested from Wikipedia, this response will be several hundred lines long. Following the end of the HTML, the connection is closed, and the Telnet client is closed, returning you to the command line.

HTML is not the only data that can be sent as a response to an HTTP request. Any plain text or binary data can be received. The format of the content is usually included in the `Content-Type` header.

Take time to explore HTTP requests and responses this way. Make requests to some of your favorite sites. Try requesting different pages and other resources within that site. Look at the response

content, the different headers and their values, and the content of any response. The Telnet client is an extremely valuable tool for understanding exactly how a server is responding. It can be very useful in debugging connection and response issues.

What are the HTTP methods?

The HTTP methods, sometimes called verbs, are instructions to the web server about what to do with the resource you are requesting. Listing 14-2 used the `GET` method. This instructs the server to find the resource requested, and serve it to the client without any modifications. This is the method used when requesting web pages in a browser.

The common methods implemented by servers are `GET`, `POST`, `PUT`, `DELETE` and `HEAD`. Only `GET` and `HEAD` are required to be implemented, although `POST` is usually implemented too; this is the method used when submitting forms from a web page.

- `GET`—Serve the requested resource; no changes should be made.
- `POST`—Update the resource to the content of the request.
- `PUT`—Set the content of the resource to the content of the request.
- `DELETE`—Remove the requested resource.
- `HEAD`—Mimic a `GET` request, but only return the response code and headers. This can be used to verify that a very large resource exists without the need to download it. If the client has previously downloaded the resource, it can check the `Last-Modified` header, which will allow the client to see if it has the most up-to-date copy.

The difference between `POST` and `PUT` can sometimes be confusing. As a rule, you use `PUT` to set new content, and `POST` to overwrite content that already exists.

The HTTP specification also advises that the `OPTIONS` method is implemented. This method is a way for clients to discover which other HTTP methods are available.

What do HTTP response codes signify?

The first line of an HTTP response includes a response code, and a short description of what that code means. The example in Listing 14-2 responded with `200 OK`. This is the code for a successful HTTP request and response. The codes are broken up into several sections, with the first digit representing the classification of the response. Any `2xx` code represents a successful request and response:

- `201 Created`—Often returned from `PUT` requests, noting that the request was successful and the resource was created
- `204 No Content`—The request was successful; the server has no more information to tell the client. This, too, is often used for successful `PUT`, `POST`, or `DELETE` requests.

The 4xx classification indicates a client error, with the server instructing the client that it did not make a valid request. Some common codes include:

- 400 Bad Request—The client did not send a well-formed request. You can often trigger this from a Telnet request, and send a junk string as the first line of a request.
- 403 Forbidden—This is often a response for a client that has provided a valid login, but they are not permitted to request the given resource.
- 404 Not Found—The requested resource does not exist.
- 405 Method Not Allowed—A resource has been requested with the wrong HTTP method.

Some servers may reply with 404 rather than 405 when making a request with the wrong HTTP method, for instance making a GET request when the server is only configured to listen for POST.

The 5xx classification indicates that there was an issue on the server side. It is not usually possible for a client to recover from these exceptions without changes happening on the server. Some common codes include:

- 500 Internal Server Error—This is a generic message. Note that this may be for a specific resource, or set of resources, rather than the entire server being unavailable.
- 503 Service Unavailable—The server is currently unavailable; this code signifies that the outage is only temporary.

The error codes can include a response body, usually in HTML or plain text, which may explain the reason for the code in more detail.

HTTP CLIENTS

How can you make HTTP requests in a Java class?

Listing 14-3 shows a simple example of performing a GET request using the standard classes with the Java library.

LISTING 14-3: Making a barebones HTTP request

```
@Test
public void makeBareHttpRequest() throws IOException {
    final URL url = new URL("http", "en.wikipedia.org", "/");
    final HttpURLConnection connection = (HttpURLConnection) url.openConnection();
    connection.setRequestMethod("GET");

    final InputStream responseInputStream = connection.getInputStream();

    final int responseCode = connection.getResponseCode();
}
```

```

        final String response = IOUtils.toString(responseInputStream);

        responseInputStream.close();

        assertEquals(200, responseCode);
        System.out.printf("Response received: [%s]%n", response);
    }

```

This approach is Java's equivalent to using Telnet to make requests. A better approach would be to use a dedicated Java HTTP client library. One of the most popular libraries for this use is Apache's `HttpClient`. Listing 14-4 shows the same request issued in Listing 14-3.

LISTING 14-4: Requests with Apache `HttpClient`

```

@Test
public void makeApacheHttpClientRequest() throws IOException {
    final CloseableHttpClient client = HttpClients.createDefault();
    HttpGet get = new HttpGet("http://en.wikipedia.org/");
    final HttpResponse response = client.execute(get);
    final int responseCode = response.getStatusLine().getStatusCode();

    final HttpEntity entity = response.getEntity();
    final InputStream responseBody = entity.getContent();

    assertEquals(200, responseCode);
    System.out.printf("Response received: [%s]%n", IOUtils.toString(responseBody));
}

```

For a small example like this, it may not appear to have saved much effort: The number of lines of code is similar, you still need to consume an input stream to read the response body, and the response code is now hidden inside a `StatusLine` object.

By wrapping the HTTP method inside a corresponding class, you are less likely to make a mistake in defining the method you want to use. A typo in the HTTP request method in the barebones example will not be spotted until run time. With the `HttpClient` example, your code will only compile with a correct HTTP method class. Using types rather than strings where appropriate is one of the main reasons for using a statically typed, compiled language. It makes sense to take advantage of that where possible.

What are the advantages of using a dedicated HTTP client library rather than using Java's built-in classes?

By using a dedicated client rather than a facility to make HTTP requests, you get an extra amount of functionality, which you would otherwise have to build in yourself.

For example, if your HTTP requests are required to run through a proxy, then for a request going through the barebones `URLConnection` class, the system properties `http.proxyHost` and `http.proxyPort` are set. If you have a more complex setup, such as local area network

requests not using a proxy but external requests do need one, it is not possible without modifying these properties at run time. For `HttpClient`, you can attach your own implementation of its `AuthenticationStrategy` interface, which can specify the need for a proxy on a request-by-request basis.

Cookies are another component that is not handled by the `HttpURLConnection` class. If you want to manage cookies, automatically providing the correct cookies to the correct sites, with the correct validity, all obeying the correct specification, then that is something you would need to implement yourself. Alternatively, you can use the `CookieStore` object attached to the `DefaultHttpClient` object.

CREATING HTTP SERVICES USING REST

Representational State Transfer, or *REST* for short, is a style of using HTTP to provide remote API calls between systems.

One of the goals of using REST is to leverage the ubiquity of the HTTP protocol, in both clients and servers, to allow systems and devices to communicate easily regardless of their implementation.

How are REST URIs designed?

REST URIs point to *resources*, and these URIs will either point to a single resource, or a collection of resources. For example, you would expect `/user/1234` to contain the details of user ID 1234, and `/users` would contain a list of all users.

The HTTP verbs signify the action to take for that resource. Using `GET` on `/user/1234` would return a representation of that user. Using `PUT` on `/user/2500` would create a new user, with ID 2500. `POST` would modify an existing user; `DELETE` would remove that user from the system. For `POST` and `PUT` requests, the representation would be in the body of the request.

The relationships between entities in a given system are built with these URIs. `/user/1234/rentals` would return a list of “rental” items appropriate to user 1234.

Using the HTTP status codes can inform the client as to the success or failure of specific requests: calling `GET` on a nonexistent item should return `404 Not Found`. Creating a new item with `PUT` could realistically return `204 No Content`. The code signifies success (the item was created), and the server has no new information to tell you about it.

How can you implement a REST service in Java?

Any of the well-known applications to create web applications, such as those shown in Chapter 13, can be used to create REST services. Libraries such as Spring MVC from version 3.0 onward carry the capability to extract variables from a request path, such as the example in Listing 14-5.

LISTING 14-5: Using Spring MVC to create REST services

```

@RequestMapping("/user/{id}")
public String getUserId(@PathVariable int userId) {
    // id path variable extracted into the userId variable
}

```

However, the Play framework is ideally suited to creating REST services. The `routes` file provides an ideal breakdown of the path variables, and how they map on to individual Java methods. This file, at a glance, makes it simple to understand your application's REST API and how it interacts with the Java code. Listing 14-6 shows an example `routes` file for use in a movie rental service.

LISTING 14-6: Creating a REST API for a movie rental service

GET	/users	controllers.App.getAllUsers()
GET	/user/:id	controllers.App.getUserById(id: Int)
PUT	/user/:id	controllers.App.createUser(id: Int)
POST	/user/:id	controllers.App.updateUser(id: Int)
DELETE	/user/:id	controllers.App.removeUser(id: Int)
GET	/user/:id/rentals/	controllers.App.showRentalsForUser(id: Int)
POST	/user/:id/rental/:movie	controllers.App.rentMovie(id: Int, movie: String)
GET	/movies	controllers.App.getAllMovies()
GET	/movie/:name	controllers.App.getMovie(name: String)
PUT	/movie/:name	controllers.App.createMovie(name: String)
GET	/rentals	controllers.App.showAllRentals()
DELETE	/rental/:movie	controllers.App.returnMovie(movie: String)

For a simple example, it is fine that all of these operations map to methods within a single class. But for a full-scale, production-ready solution, these services would be broken down into their own classes, each with its own services objects for interacting with a database, or something similar.

When viewed with the HTTP method, the paths describe the action to take. Calling `GET /users` will return a representation of all the users in the system. Calling `PUT /movie/Batman` will create a movie on the server called Batman. For this call, you can see that the movie title has been bound to a name variable, of type `String`.

The more interesting path, `PUT /rental/:user/:movie`, links the two entities, allowing users to rent movies.

Listing 14-7 shows some of the `App` class definition for creating users.

LISTING 14-7: Creating users for the movie rental service

```

private static List<User> users = new ArrayList<>();

public static Result getAllUsers() {

```

continues

LISTING 14-7 (*continued*)

```

        return ok(users.toString());
    }

    public static Result getUserById(int id) {
        final User user = findUser(id);
        if (user == null) {
            return notFound(String.format("User ID [%s] not found", id));
        }
        return ok(user.toString());
    }

    public static Result createUser(int id) {
        final JsonNode jsonBody = request().body().asJson();
        final String username = jsonBody.get("name").asText();
        users.add(new User(id, username));
        return noContent();
    }
}

```

When creating users, the service expects a small piece of JSON data, containing the name of the user, for example `{"name": "alice"}`. For larger applications, this data would be much larger, containing much more information about the created item.

One interesting point to note when creating new items is that you `PUT` to the resource location. This implies that you must be able to already uniquely identify the resource (the `id` in this case). To find a unique identifier, you may need to find all items (`/users`, which calls `getAllUsers` in this case), and work out a new identifier for your creation. Not only is this tedious and inefficient, but also if multiple clients are using the application, you may find yourself in a *race condition* where multiple clients have found what they think is a unique identifier, and then try to create the resource with that same ID.

An alternative is to have the server create the identifier for you. You can create an endpoint, `PUT /user`, and then rather than that call returning `204 No Content`, it could return `200 OK`, and the body of the response could contain the full URI to the created item. Note that this means that multiple calls to this endpoint would result in multiple resources being created.

The code for creating movies is very similar to that for creating users. Listing 14-8 shows the Java code for the rental URIs.

LISTING 14-8: Creating and removing rentals

```

private static Map<Movie, User> rentals = new HashMap<>();

public static Result showAllRentals() {
    return ok(rentals.toString());
}

public static Result showRentalsForUser(int id) {
    final User user = findUser(id);
}

```

```

        if (user == null) {
            return notFound(String.format("User ID [%s] not found", id));
        }
        final List<Movie> moviesForUser = new ArrayList<>();
        for (Map.Entry<Movie, User> entry : rentals.entrySet()) {
            if (entry.getValue().equals(user)) {
                moviesForUser.add(entry.getKey());
            }
        }

        return ok(moviesForUser.toString());
    }

    public static Result rentMovie(int userId, String movieName) {
        final Movie movie = findMovie(movieName);
        final User user = findUser(userId);
        final List<String> errors = new ArrayList<>();
        if (movie == null) {
            errors.add("Movie does not exist");
        }
        if (user == null) {
            errors.add("User ID does not exist");
        }
        if (movie != null && rentals.containsKey(movie)) {
            errors.add("Movie already out for rent");
        }
        if (!errors.isEmpty()) {
            return badRequest(
                String.format("Unable to rent movie. Reason: %s",
                    errors.toString()));
        }

        rentals.put(movie, user);
        return noContent();
    }

    public static Result returnMovie(String movieName) {
        final Movie movie = findMovie(movieName);
        if (movie == null) {
            return badRequest("Movie does not exist");
        }
        rentals.remove(movie);
        return noContent();
    }
}

```

These method calls are doing some basic validation of the input, such as that users and movies exist, and that the movie is available for rent if that request has been made.

Of course, this application has plenty of issues—it is not thread-safe and it is insecure. But it does show how to build an application that will respond to the RESTful requests.

Though it may appear obvious, it is worth pointing out that the clients of this service are completely platform independent. By agreeing on an interface for the HTTP calls, any platform capable of

making HTTP calls—such as a web browser, mobile phone, or command-line client that has access to the server, either over the Internet or a local intranet—can make use of it.

Another feature that is important to a REST service is the notion of *idempotency*. For REST services, `PUT` and `DELETE` should be idempotent, and it is implied that a `GET` request is idempotent too. A REST call can be defined as idempotent if multiple requests have the same effect as one. When you call `PUT` on a resource, you are creating a new resource on the server, so making that same call again should result in that exact same resource being created. The same is true for `DELETE`: After `DELETE` has been called, the resource is no longer available. If you call `DELETE` again, that resource is still not available. With `GET`, all that should happen is a specified resource is returned to the client, no changes made; so making that same `GET` should result in the same data being returned.

The advantage of an idempotent API is that if a client cannot be sure that a request was successful, submitting the request again will be fine. This can be extremely useful with mobile phone applications, which may lose a network connection at any time, even halfway through making a request, or after a request has been made, but before the response is received.

The definition of `POST` is that an update is made to an existing resource. For instance, if you had a service endpoint that updated the value of the resource by 1, then multiple calls to that will result in the resource having a different new value after every call.

SUMMARY

This chapter has covered several uses of the HTTP protocol and its interaction with Java, both as a client and as a service.

Using HTTP is a well-understood, known way for different applications to interact with each other. More often than you realize, clients will be communicating with services over HTTP. REST and HTTP APIs in general can often be split into public and private APIs. Be aware that most implementations are private, and are specific to an organization or even a specific application, and are used for such an application to talk to its own servers. These are a lot more prevalent than public APIs, such as Twitter's API or services such as the Weather Channel's API.

Considering that most APIs are not open to the outside world, they are used a lot more than people appreciate. It is very important to have a good understanding of how HTTP and REST work, regardless of whether you would be interviewing for a server-side or front-end developer role, because the interoperability between the two is key. REST serves as a language and vocabulary that both sides understand, so it is important that you do too.

The movie rental service described in this chapter included a small amount of *JSON*, or *JavaScript Object Notation*. This is a way of taking an instance of an object, and creating a representation of the state held in its fields. This representation can be transmitted somehow to another application, and that application can reconstruct the same object. This technique is known as *serialization*, and is common whenever two or more applications or services talk to each other. The next chapter covers the concept of serialization in more depth.

15

Serialization

Interoperation between a client and server can be separated into two distinct parts: *how* the data is transmitted, and *what* is transmitted. The previous chapter gave one method for how to transmit and receive data, and this chapter explains different methods for representing your data over that transmission.

The methods here are not specific to network communication. Object serialization is simply a way of “exporting” Java objects from the JVM: The serialized data can be written to disk or another I/O interface rather than to the network.

For convenience, most of the listings in this chapter read and write serialized data to and from the filesystem, using `FileInputStreams` and `FileOutputStreams`. These can be substituted for any `InputStream` and `OutputStream`, such as the ones provided by `HttpServletRequest` and `HttpServletResponse` if you wanted to use the data in an HTTP server.

READING AND WRITING JAVA OBJECTS

What does the JVM provide for writing objects to a third party?

The standard Java libraries provide a pair of classes for writing and reading Java objects: `ObjectOutputStream` and `ObjectInputStream`. These are part of the `InputStream` and `OutputStream` family of classes.

For writing objects, `ObjectOutputStreams` are constructed using the *Decorator Pattern*, and require another `OutputStream` to physically write the serialized data. The `ObjectOutputStream` class has methods for all the primitive types, as well as a method for reference types, for writing these values to the stream. Listing 15-1 shows the use of the `ObjectOutputStream`.

LISTING 15-1: Writing Java types to a file

```
public class Pair implements Serializable {
    private final int number;
    private final String name;

    public Pair(int number, String name) {
        this.number = number;
        this.name = name;
    }

    public int getNumber() {
        return number;
    }

    public String getName() {
        return name;
    }
}

@Test
public void writeData() throws IOException {
    final FileOutputStream fos = new FileOutputStream("/tmp/file");
    final ObjectOutputStream oos = new ObjectOutputStream(fos);

    oos.writeInt(101);
    oos.writeBoolean(false);
    oos.writeUTF("Writing a string");

    final Pair pair = new Pair(42, "Forty two");
    oos.writeObject(pair);

    oos.flush();
    oos.close();
    fos.close();
}
```

If you take a look at the data written to the file `/tmp/file`, you will see that it is binary data. This has been written using a form that the JVM understands and is able to read back, too, as Listing 15-2 shows.

LISTING 15-2: Reading from an ObjectInputStream

```
@Test
public void readData() throws IOException, ClassNotFoundException {
    final FileInputStream fis = new FileInputStream("/tmp/file");
    final ObjectInputStream ois = new ObjectInputStream(fis);

    final int number = ois.readInt();
    final boolean bool = ois.readBoolean();
    final String string = ois.readUTF();
}
```

```

        final Pair pair = (Pair) ois.readObject();

        assertEquals(101, number);
        assertFalse(bool);
        assertEquals("Writing a string", string);
        assertEquals(42, pair.getNumber());
        assertEquals("Forty two", pair.getName());
    }

```

It is important to read the data in the same order in which you wrote it! If you, for instance, try to call `readObject` when the next type in the stream is an `int`, you will get an exception.

There are two important things of note for the small `Pair` class created for this example. Notice that the `readData` test can possibly throw a `ClassNotFoundException`. If you are reading this file in a different JVM than the one in which it was written, and the `Pair` class is not on the classpath, the JVM will not be able to construct the object.

Also, the `Pair` class must implement the `Serializable` interface. This is a *marker interface*, which signals to the JVM that this class can be serialized.

What does the `transient` keyword do?

If you have a `Serializable` object, but have fields that you do not want to be serialized when writing the data to a stream, you can apply the `transient` modifier to the field declaration. When a `transient` field is deserialized, that field will be null. Listing 15-3 shows an example.

LISTING 15-3: Using transient fields

```

public class User implements Serializable {
    private String username;
    private transient String password;

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public String getPassword() {
        return password;
    }
}

@Test
public void transientField()
    throws IOException, ClassNotFoundException {

```

continues

LISTING 15-3 *(continued)*

```

    final User user = new User("Noel", "secret321");

    final FileOutputStream fos = new FileOutputStream("/tmp/user");
    final ObjectOutputStream oos = new ObjectOutputStream(fos);

    oos.writeObject(user);

    oos.flush();
    oos.close();
    fos.close();

    final FileInputStream fis = new FileInputStream("/tmp/user");
    final ObjectInputStream ois = new ObjectInputStream(fis);

    final User deserialized = (User) ois.readObject();

    assertEquals("Noel", deserialized.getUsername());
    assertNull(deserialized.getPassword());
}

```

The main use cases for creating transient fields are when you are holding a private field as a cache, so you can simply regenerate the cache after deserializing, or when data can be reconstructed after it has been deserialized, such as a field created from other fields:

```

private String firstName = "Bruce";
private String lastName = "Springsteen";
private String fullName = String.format("%s %s", firstName, lastName);

```

As Listing 15-3 shows, you can hold secure or sensitive data in a transient field, and be confident that it will not be written to an `ObjectOutputStream`.

USING XML

Is it possible for applications written in different languages to communicate?

Rather than delegating the representation of objects to the JVM and the mechanics of `ObjectInputStream` and `ObjectOutputStream`, you could define the format yourself.

One huge advantage of doing this is that once the object has been serialized, it can be read by any process that understands your format, whether that is another JVM or perhaps a different setup altogether.

A popular notation for defining domain objects is XML. It allows for a well-structured representation of data, and most languages have well-established libraries for parsing and writing XML.

A meta-language, called *XML Schema Definition* (XSD), enables you to define a domain object using XML markup, which is used to create XML documents. Java has a library called JAXB,

which understands XSD notation and can create Java objects from these documents. This library is often used within other libraries to serialize and deserialize objects, for use in SOAP applications, or similar.

Listing 15-4 shows an XSD for representing players on a sports team.

LISTING 15-4: A sample XML Schema document

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Team">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Name" type="xs:string"/>
        <xs:element name="DateFounded" type="xs:date"/>
        <xs:element name="Players">
          <xs:complexType>
            <xs:sequence minOccurs="2" maxOccurs="11">
              <xs:element name="Player">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Name" type="xs:string"/>
                    <xs:element name="Position" type="xs:string"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

This schema allows the creation of `Team` types, and among the attributes of that type, it has a collection of `Player` types. This allows for the creation of sample XML documents such as the one in Listing 15-5.

LISTING 15-5: A Team XML document

```
<?xml version="1.0" encoding="utf-8"?>
<Team xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="team.xsd">
  <Name>AllStars</Name>
  <DateFounded>2001-09-15</DateFounded>
  <Players>
    <Player>
      <Name>Adam Smith</Name>
      <Position>Goal</Position>
    </Player>
    <Player>
```

continues

LISTING 15-5 *(continued)*

```

        <Name>Barry Jenkins</Name>
        <Position>Defence</Position>
    </Player>
</Players>
</Team>

```

By including the schema location in the `Team` definition, any software you use that understands XSD can validate this document. This includes parsers such as JAXB, which you will see next, or even IDEs with smart XML and XSD support. These can provide smart auto-completion to help you generate the document.

Can XSD documents be used to create Java domain objects?

JAXB is run through a separate compiler, which parses XSD content and creates Java source files. The `xjc` command is distributed as part of the JDK. If you run

```
xjc -p com.wiley.acinginterview.chapter15.generated /path/to/team.xsd
```

you should see the following output:

```

parsing a schema...
compiling a schema...
com/wiley/acinginterview/chapter15/generated/ObjectFactory.java
com/wiley/acinginterview/chapter15/generated/Team.java

```

If you inspect the `Team.java` file, you can see that it is a regular Java class. This class is a representation of the `team.xsd` created in Listing 15-5, and can be used as any Java object, such as that shown in Listing 15-6.

LISTING 15-6: Using JAXB-generated Java objects

```

@Test
public void useJaxbObject() {
    final Team team = new Team();
    team.setName("Superstars");

    final Team.Players players = new Team.Players();

    final Team.Players.Player p1 = new Team.Players.Player();
    p1.setName("Lucy Jones");
    p1.setPosition("Striker");

    final Team.Players.Player p2 = new Team.Players.Player();
    p2.setName("Becky Simpson");
    p2.setPosition("Midfield");
    players.getPlayer().add(p1);
}

```

```

        players.getPlayer().add(p2);

        team.setPlayers(players);

        final String position = team
            .getPlayers()
            .getPlayer()
            .get(0)
            .getPosition();

        assertEquals("Striker", position);
    }

```

The point of interest with this generated object is that the rich `Player` and `Players` objects are static inner classes of the `Team` object, because these were created as nested complex types in the original XSD. These could have been created as top-level objects if you wanted to use them elsewhere, as shown in Listing 15-7.

LISTING 15-7: Top-level XSD types

```

<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Player">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Name" type="xs:string"/>
        <xs:element name="Position" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Players">
    <xs:complexType>
      <xs:sequence minOccurs="2" maxOccurs="11">
        <xs:element ref="Player"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  ...
</xs:schema>

```

You can see here that generated types are included with the `ref` attribute, rather than `type`.

This then creates Java objects, which can be used as shown in Listing 15-8.

LISTING 15-8: Using top-level XSD types

```

@Test
public void useTopLevelJaxbObjects() {
    final Team team = new Team();

```

continues

LISTING 15-8 *(continued)*

```

    team.setName("Superstars");

    final Players players = new Players();

    final Player p1 = new Player();
    p1.setName("Lucy Jones");
    p1.setPosition("Striker");

    final Player p2 = new Player();
    p2.setName("Becky Simpson");
    p2.setPosition("Midfield");
    players.getPlayer().add(p1);
    players.getPlayer().add(p2);

    team.setPlayers(players);

    final String position = team
        .getPlayers()
        .getPlayer()
        .get(0)
        .getPosition();

    assertEquals("Striker", position);
}

```

Clearly, this is much easier to read and work with.

One point that often surprises new users is that you do not need to create, or *set*, a collection type, such as the `Players` in this example. Rather, you call *get* to get the collection instance, and then you can add or remove from that collection.

Can XSD and JAXB be used to serialize a Java object?

Once the JAXB bindings have been created using `xjc`, parsing an XML document that conforms to an XSD is relatively simple, as shown in Listing 15-9.

LISTING 15-9: Reading XML into a Java object

```

@Test
public void readXml() throws JAXBException {
    final JAXBContext context =
        JAXBContext.newInstance(
            "com.wiley.javainterviewsexposed.chapter15.generated2");
    final Unmarshaller unmarshaller = context.createUnmarshaller();
    final Team team = (Team) unmarshaller
        .unmarshal(getClass()
            .getResourceAsStream("/src/main/xsd/teamSample.xml"));

    assertEquals(2, team.getPlayers().getPlayer().size());
}

```

Although this XML document was provided as a resource on the classpath, it could easily have come from any source, such as a messaging queue or an HTTP request.

Similarly, writing Java objects as XML is also simple, as Listing 15-10 demonstrates.

LISTING 15-10: Using JAXB to serialize a Java object

```
@Test
public void writeXml() throws JAXBException, FileNotFoundException {
    final ObjectFactory factory = new ObjectFactory();
    final Team team = factory.createTeam();
    final Players players = factory.createPlayers();
    final Player player = factory.createPlayer();
    player.setName("Simon Smith");
    player.setPosition("Substitute");
    players.getPlayer().add(player);

    team.setName("Megastars");
    team.setPlayers(players);

    final JAXBContext context =
        JAXBContext.newInstance(
            "com.wiley.javainterviewsexposed.chapter15.generated2");
    final Marshaller marshaller = context.createMarshaller();
    marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
    marshaller.marshal(team, new FileOutputStream("/tmp/team.xml"));
}
```

Listing 15-10 produces the following XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Team>
  <Name>Megastars</Name>
  <Players>
    <Player>
      <Name>Simon Smith</Name>
      <Position>Substitute</Position>
    </Player>
  </Players>
</Team>
```

It is important that the `Marshaller` was created from a specific `JAXBContext`, which was configured to the package where `xjc` created the Java objects. If you use a different package for your automatically generated objects, the `Marshaller` will not be able to parse your Java objects into XML.

What approach can be used to make sure the XSD definitions and your source code are kept in sync?

It is possible to have your Java sources compiled as part of your build. Listing 15-11 shows the `plugin` tag for the Maven POM for the examples used in this section.

LISTING 15-11: Automatically generating JAXB objects

```

<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>jaxb2-maven-plugin</artifactId>
  <version>1.5</version>
  <executions>
    <execution>
      <id>xjc</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>xjc</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <packageName>
      com.wiley.javainterviewsexposed.chapter15.generated3
    </packageName>
  </configuration>
</plugin>

```

This assumes that your XSD files are contained in `src/main/xsd`. This plug-in will fire with the `generate-sources` phase of the build, meaning that any XSD objects will be re-created before the main source of the project is compiled.

Having your domain objects generated by the build is an extremely useful feature, because any breaking changes to the model will also break your build, and the tests will not likely work, either. Fixing your code following a breaking change gives you confidence that your serialization and deserialization routines are working properly.

JSON

What is JSON?

JavaScript Object Notation, or JSON, is another serialization approach, similar to using XML with XSD. It uses a human-readable approach, which can be parsed and processed by a number of languages. Indeed, JSON originated as part of JavaScript, and has been adopted by libraries for other languages as a lightweight, less verbose approach over XML.

Listing 15-12 illustrates the team model in JSON, previously used in earlier examples in this chapter.

LISTING 15-12: A JSON example

```

{
  "name": "AllStars",
  "dateFounded": "2001-09-15",
  "players": [
    {
      "name": "Adam Smith",
      "position": "Goal"
    },
    {
      "name": "Barry Jenkins",
      "position": "Defence"
    }
  ]
}

```

For JSON, there is a meta-language called *JSON Schema*, but it is not widely used. A JSON document is loosely defined, but is still well structured. There are very few types: A string is text surrounded by quotes, there is no differentiation between floating point and integral numbers, and you can use the boolean values `true` and `false`. A list is a comma-separated list, surrounded by square brackets, like the list of players in Listing 15-12. An object is a collection of key-value pairs surrounded by curly braces. The keys are always strings; the value is any type. Any type can be `null`.

How can JSON be read in a Java application?

An implementation of JSON parsing and processing is contained in a library called Jackson. It upholds JSON's simplicity by being straightforward to use. Listing 15-13 contains sample domain objects for the `Team` and `Players`, and Listing 15-14 illustrates how to parse the JSON into these objects.

LISTING 15-13: Sample domain objects for Team and Players

```

public class Team {

    private String name;
    private String dateFounded;
    private List<Player> players;

    ... // include getters and setters
}

public class Player {

    private String name;
    private String position;
    ... // include getters and setters
}

```

LISTING 15-14: Parsing JSON into rich domain objects

```

@Test
public void readJson() throws IOException {
    final ObjectMapper mapper = new ObjectMapper();
    final String json =
        "/com/wiley/javainterviewsexposed/chapter15/team.json";
    final Team team = mapper.readValue(
        getClass().getResourceAsStream(json),
        Team.class);

    assertEquals(2, team.getPlayers().size());
}

```

This code makes heavy use of reflection to make sure the JSON data will fit into the given domain object, so make sure you are thorough with your testing if you use this approach.

Can JSON be used to serialize a Java object?

The Jackson library can also be used to create JSON from a domain object, which works exactly as you would expect, as shown in Listing 15-15.

LISTING 15-15: Writing domain objects to JSON

```

@Test
public void writeJson() throws IOException {
    final Player p1 = new Player();
    p1.setName("Louise Mills");
    p1.setPosition("Coach");

    final Player p2 = new Player();
    p2.setName("Liam Turner");
    p2.setPosition("Attack");

    final Team team = new Team();
    team.setPlayers(Arrays.asList(p1, p2));

    final ObjectMapper mapper = new ObjectMapper();
    mapper.writeValue(new File("/tmp/newteam"), team);
}

```

This code produces the following JSON:

```

{
  "name": null,
  "dateFounded": null,
  "players": [
    {
      "name": "Louise Mills",

```



```
        "position": "Coach"
    },
    {
        "name": "Liam Turner",
        "position": "Attack"
    }
]
}
```

Note that `name` and `dateFounded` are `null`, because these were not set on the `Team` object.

SUMMARY

Creating Java applications and servers that can communicate with other services and devices is often one of the main talking points in interviews. Being able to create stable and robust services is extremely important; however, defining how the services talk to each other can have a major influence on application performance, as well as architectural choices made early in a project.

Many development teams, particularly ones that work heavily in cross-platform communication, such as between a mobile device and server, are turning to JSON for their communications over the wire. It is much less verbose than XML, which is useful when taking volumes of data into account. But it is also less strict because there is no document definition as there is with XSD, so be aware of this trade-off: You will need to be sure that your document is correct before parsing or sending to a recipient.

Several serialization technologies not covered here communicate using a binary protocol rather than the plaintext approaches of XML and JSON. However, unlike Java's own serialization techniques, the binary protocols such as Google's Protocol Buffers or Apache Thrift are designed to be cross-platform, and work over a multitude of languages and devices.

The next chapter covers the Spring Framework, including a section on creating web services using Spring MVC. The techniques used in this chapter can be applied to Spring MVC in defining exactly what is transmitted when querying or receiving responses from a web server.

16

The Spring Framework

The Spring Framework is designed to be a lightweight, non-intrusive framework, providing you with tools and facilities to write standalone, testable components for use in many common applications and scenarios.

Spring enables you to write *Plain Old Java Objects*, or *POJOs*, and then connect them together using *dependency injection*, through a central core component called the *application context*. POJOs are classes that have no unnecessary restriction, such as implementing an interface for a persistence framework, or extending an abstract class for use in a web application.

The notion of POJOs comes as a reaction from the early approaches to *Enterprise Java Beans* (EJBs), where your simple domain object needed to follow complicated patterns and implement several interfaces in order to be used within application servers. This code was often brittle and untestable. In turn, this led to a rise in popularity of frameworks such as Spring, which concentrated on simple approaches around application configuration and focused on writing testable code.

This chapter covers several core areas of the Spring Framework: the application context, database programming with the `JdbcTemplate`, the integration testing classes, and a section on producing web applications using Spring MVC.

CORE SPRING AND THE APPLICATION CONTEXT

How do you use dependency injection with Spring?

Chapter 9 showed a brief example of what dependency injection is, and what advantage it gives. However, the management of the dependencies was completely manual: If you wanted to provide another implementation of a particular dependent interface, you would need to make the changes in the actual code, and recompile and repackage your new built artifact.

Dependency Injection is one of Spring's core features. It enables you to remove any specific dependencies a class may have on other classes or third party interfaces, and load these dependencies into the class at construction time. The benefit is that you can change the implementation of your dependencies without having to change the code in your class, which allows for easier, isolated testing. You can even rewrite implementations of your dependencies without having to change your class, as long as the interface does not change.

Using the Spring's mechanism for dependency injection mitigates some of these issues, and can provide a useful framework for writing robust, tested, flexible code. Listing 16-1 is a possible application as a simple spell checker. The `checkDocument` method returns a list of numbers: the list indices of the misspelled words.

LISTING 16-1: A simple spell checker

```
public class SpellCheckApplication {

    private final Dictionary dictionary;

    public SpellCheckApplication(final Dictionary dictionary) {
        this.dictionary = dictionary;
    }

    public List<Integer> checkDocument(List<String> document) {
        final List<Integer> misspelledWords = new ArrayList<>();

        for (int i = 0; i < document.size(); i++) {
            final String word = document.get(i);
            if (!dictionary.validWord(word)) {
                misspelledWords.add(i);
            }
        }

        return misspelledWords;
    }
}
```

You should recognize this pattern from Chapter 9: The spell checker *depends on* the dictionary; it calls the boolean `validWord(String)` method from the `Dictionary` interface to check whether a given word is spelled correctly. This spell checker doesn't know, or even need to care about, the language of the document, or if it should be using a specific dictionary with particularly difficult or technical words, or even if there is a valid difference between a capitalized word and one that is not. This can be configured when the class is actually running.

You should recognize that you don't even need a concrete `Dictionary` implementation for testing; the `validWord` method can be easily mocked to allow all the code paths of the `checkDocument` method to be tested properly.

But what happens when it comes to running this code for real? Listing 16-2 provides a naïve, but working implementation.

LISTING 16-2: A class to run the spell checker

```

public class SpellCheckRunner {

    public static void main(String[] args) throws IOException {
        if(args.length < 1) {
            System.err.println("Usage java SpellCheckRunner <file_to_check>");
            System.exit(-1);
        }

        final Dictionary dictionary =
            new FileDictionary("/usr/share/dict/words");
        final SpellCheckApplication checker =
            new SpellCheckApplication(dictionary);
        final List<String> wordsFromFile = getWordsFromFile(args[0]);

        final List<Integer> indices = checker.checkDocument(wordsFromFile);

        if (indices.isEmpty()) {
            System.out.println("No spelling errors!");
        } else {
            System.out.println("The following words were spelled incorrectly:");
            for (final Integer index : indices) {
                System.out.println(wordsFromFile.get(index));
            }
        }
    }

    static List<String> getWordsFromFile(final String filename) {
        /* omitted */
    }
}

```

The specifics of the `FileDictionary` class do not matter; from its use here, you can visualize that the constructor takes a `String` argument to a location on disk of a list of valid words.

This implementation is completely inflexible. If the location of the dictionary file changes, or perhaps a requirement comes to use a third-party online dictionary, you will need to edit this code, potentially making other mistakes as you update it.

An alternative is to use a dependency injection framework, such as Spring. Rather than constructing the objects and injecting the necessary dependencies, you can let Spring do it for you. Listing 16-3 shows a definition of a Spring application context represented in XML.

LISTING 16-3: A Spring context for the spell checker

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation=" http://www.springframework.org/schema/beans

```

continues

LISTING 16-3 (*continued*)

```

http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="dictionary"
        class="com.wiley.javainterviewsexposed.chapter16.FileDictionary">
        <constructor-arg value="/usr/share/dict/words"/>
    </bean>

    <bean id="spellChecker"
        class="com.wiley.javainterviewsexposed.chapter16.SpellCheckApplication">
        <constructor-arg ref="dictionary"/>
    </bean>

</beans>

```

This XML file defines two *beans*, one for the dictionary, and one for the spell checker. Values for the constructors are provided here, too: The `FileDictionary` class takes a `String`, and this is defined inside the bean definition with the `constructor-arg` tag.

As well as providing configuration to Spring beans with the `constructor-arg` tag, you can also use the `property` tag. These call setters on that class. For instance, having a property such as `<property name="location" value="London"/>` will call the method `setLocation` with the parameter `London`.

Of course, this will fail if such a method does not exist, and that failure will not happen until run time.

The `SpellCheckApplication` class uses the constructed dictionary bean in its own constructor. The `constructor-arg` tag uses the `ref` attribute to reference another Spring bean.

Now that you have defined the relationship between these classes, you still need a running application to do your spell checking. Listing 16-4 uses the application context defined in Listing 16-3.

LISTING 16-4: Using the Spring application context

```

public class SpringXmlInjectedSpellChecker {

    public static void main(String[] args) {
        if(args.length < 1) {
            System.err.println("Usage java SpellCheckRunner <file_to_check>");
            System.exit(-1);
        }

        final ApplicationContext context =
            new ClassPathXmlApplicationContext(
                "com/wiley/javainterviewsexposed/" +
                "chapter16/applicationContext.xml");

        final SpellCheckApplication checker =

```

```

        context.getBean(SpellCheckApplication.class);

        final List<String> wordsFromFile = getWordsFromFile(args[0]);

        final List<Integer> indices = checker.checkDocument(wordsFromFile);

        if (indices.isEmpty()) {
            System.out.println("No spelling errors!");
        } else {
            System.out.println("The following words were spelled wrong:");
            for (final Integer index : indices) {
                System.out.println(wordsFromFile.get(index));
            }
        }
    }
}

```

Following some boilerplate code checking the program arguments, an `ApplicationContext` object is created. There are several implementations of `ApplicationContext`; the one used here, `ClasspathXmlApplicationContext`, examines the JVM classpath for the given *resource*, and parses that XML document.

The construction of the application context will instantiate the objects defined in that XML file.

Following the construction of the application context, you can ask it to return the bean instances it has constructed, using the `getBean` method.

The remaining code in this class is the same as the simple `SpellCheckRunner` in Listing 16-2.

If you want to make changes to the way the dictionary implementation and the spell checker are set up, you can make these changes without touching the code that has already been written. If you were to use a different file location for the dictionary, updating the `constructor-ref` of the dictionary bean is all that is necessary. If you add a new `Dictionary` implementation, once you have written your class, you add the definition to your context, and make sure your `SpellCheckApplication` instance is referring to the correct bean. Listing 16-5 shows a possible application context for this case.

LISTING 16-5: Swapping Dictionary implementations

```

<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

    <bean id="dictionary"
        class="com.wiley.javainterviewsexposed.chapter16.FileDictionary">
        <constructor-arg value="/usr/share/dict/words"/>
    </bean>

    <bean id="thirdPartyDictionary"
        class="com.wiley.javainterviewsexposed.chapter16.OnlineDictionary">
        <constructor-arg value="http://www.example.org/api/dictionary"/>
    </bean>
</beans>

```

continues

LISTING 16-5 *(continued)*

```

    </bean>

    <bean id="spellChecker"
        class="com.wiley.javainterviewsexposed.chapter16.SpellCheckApplication">
        <constructor-arg ref="thirdPartyDictionary"/>
    </bean>

</beans>

```

One concern with using a plain-text XML document for constructing Spring beans and defining dependencies is that you lose the value of compile-time type checking; any issues such as using the wrong type in a constructor, or even the wrong number of arguments in a constructor, are not found until the application context is constructed at run time. If you have good integration test coverage, which is run as part of your regular application build, this is one way to mitigate that risk. Another approach is not to use XML definitions at all.

How do you define an application context in Java code?

Introduced in Spring 3.0, it is now possible to have Spring scan the classpath for a configuration.

If you provide a class annotated with `@Configuration`, then any methods in that class that are annotated with `@Bean` will be treated in the same manner as those defined in XML with the `<bean>` tag. Listing 16-6 illustrates the same configuration as that in Listing 16-3.

LISTING 16-6: A Spring context defined in Java

```

@Configuration
public class SpringJavaConfiguration {

    @Bean
    public Dictionary dictionary() throws IOException {
        return new FileDictionary("/usr/share/dict/words");
    }

    @Bean
    public SpellCheckApplication spellCheckApplication() throws IOException {
        return new SpellCheckApplication(dictionary());
    }
}

```

There is one extra change to make; you still need to initialize the application context. You can do this one of two ways: Either instantiate the application with the `AnnotationConfigApplicationContext` class, which takes a list of classes annotated with `@Configuration`:

```

final ApplicationContext context =
    new AnnotationConfigApplicationContext(SpringJavaConfiguration.class);

```


or you can still use the original method, loading the application context from an XML file. In that case, you need to give a scope for which packages to look for the `@Configuration` annotation, by including this line:

```
<context:component-scan
    base-package="com.wiley.javainterviewsexposed.chapter16"/>
```

The beauty of this method is that you can mix both annotated configuration and XML configuration. This may seem silly, and a potential nightmare to manage two different systems for managing dependencies, but it does give you the scope to migrate from one to the other at your own pace.

What are scopes?

On examination of Listing 16-6, you see that the dependency between the `SpellCheckApplication` and `Dictionary` is made explicit, by calling `dictionary()` in the constructor to the `SpellCheckApplication` class.

It is important to note here that a class defined with an `@Configuration` annotation *is not an application context*. It defines how that context is created and managed. This is exactly the same for the XML configuration, too. Taking any of the Spring-managed application contexts from this chapter, you can make multiple calls to `getBean` on the `ApplicationContext` instance, and it will always return the same instance. By default, Spring beans are created when the application context is instantiated, called *eager instantiation*. Additionally, only one instance of a Spring bean is created for a given definition. This is called *singleton scope*.

When a singleton bean is provided from the context, it is important to note that several concurrent threads may be using that instance at the same time. Therefore, it is very important that any operations are thread-safe.

It is possible for the application context to return a new instance on each invocation of `getBean`. Listing 16-7 shows such an example for a bean defined as `<bean id="date" class="java.util.Date" scope="prototype"/>`.

LISTING 16-7: Using the prototype scope

```
@Test
public void differentInstanceScope() {
    final ApplicationContext context =
        new ClassPathXmlApplicationContext(
            "com/wiley/javainterviewsexposed/chapter16/applicationContext.xml");
    final Date date1 = context.getBean(Date.class);
    final Date date2 = context.getBean(Date.class);

    assertNotSame(date1, date2);
}
```

Having the context return a new instance of a bean on each call to `getBean` is called *prototype scope*.

Other scope types exist, too, such as `request`, where a bean lives for the life of a particular HTTP request, or `session`, where the bean lives for the life of the HTTP session.

How can environment-specific properties be set in a manageable way?

The class `PropertyPlaceholderConfigurer` is a simple and easy way to set properties external from a build. It is good practice to keep any environment-specific configuration outside of a regular build, and loaded at run time, rather than building a per-environment artifact for each of the environments you test for.

REBUILDING YOUR APPLICATION FOR DIFFERENT ENVIRONMENTS

Some applications are deployed to several environments for different rounds of testing, before eventually being released to a “production,” or “live” environment.

Sometimes, the configuration for each environment is pulled in to the application binary as part of the build, and is subsequently rebuilt for each environment. This is a risky approach, because this means that the actual compiled binary artifact you deploy to production is not the same one as was deployed to your test environments.

If you have your environment-specific configuration external to the build, and pulled in at run time using a `PropertyPlaceholderConfigurer`, it means that you are testing the exact binary artifact you will eventually deploy to your live environment. You have more confidence in your release, because you will not have introduced the risk of creating the binary in a different way for your live environment as to the other environments.

Indeed, if you compile one binary on your computer, and your colleague issues the same build instruction on his or her computer, these could differ in many ways, such as different compiler versions, different operating systems, or even different environment variables set.

Listing 16-8 is a sample application context, which uses the `PropertyPlaceholderConfigurer`, and then uses the properties to set up other Spring beans.

LISTING 16-8: Using the `PropertyPlaceholderConfigurer`

```
<bean
  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property
    name="location"
    value="/com/wiley/javainterviewsexposed/chapter16/environment.properties"/>
  <property name="placeholderPrefix" value="$property{" />
  <property name="placeholderSuffix" value="}"/>
</bean>
```

```

</bean>

<bean class="com.wiley.javainterviewsexposed.chapter16.ServerSetup">
  <constructor-arg value="$property{server.hostname}"/>
</bean>

```

The `PropertyPlaceholderConfigurer` class includes several useful properties. In Listing 16-8, the `location` property is set to a specific property file, and the class parses that file. The `PropertyPlaceholderConfigurer` expects the `environment.properties` file to hold the properties as key-value pairs.

This file is parsed on the startup of the application context, and these properties can be used as configuration parameters for other beans. The property in the file as `server.hostname` is used to create a `ServerSetup` bean.

It should be quite clear that, for this application, it is possible to change the values stored in `environment.properties`, restart the application, and the `ServerSetup` bean will be configured with a different value to its constructor. This is a simple but powerful way to separate environment-specific configuration from the actual code.

What is autowiring?

Autowiring is the process of letting the application context figure out the dependencies for you. If you are trying to construct a bean with dependencies, sometimes you do not need to explicitly define that link. Listing 16-9 and Listing 16-10 illustrate.

LISTING 16-9: XML configuration for autowiring

```

<bean id="dictionary" class="com.wiley.javainterviewsexposed.chapter16.FileDictionary">
  <constructor-arg value="/usr/share/dict/words"/>
</bean>

<bean id="spellChecker"
      class="com.wiley.javainterviewsexposed.chapter16.SpellCheckApplication"
      autowire="constructor"/>

```

LISTING 16-10: Using autowiring

```

@Test
public void getAutowiredBean() {
    final ApplicationContext context =
        new ClassPathXmlApplicationContext(
            "com/wiley/javainterviewsexposed/" +
            "chapter16/applicationContextAutowiring.xml");
    final SpellCheckApplication bean =
        context.getBean(SpellCheckApplication.class);
    assertNotNull(bean);
}

```

The XML configuration has instructed the application context to find types that can be used in the constructor of the `SpellCheckApplication` class, that is, a class of type `Dictionary`. Because the only other class in this application context is a `FileDictionary`—an implementation of `Dictionary`—this is used to construct the `SpellCheckApplication` class.

Had there not been a `Dictionary` implementation in the context, an error would have been thrown when trying to construct the application context: `NoSuchBeanDefinitionException`.

Had there been multiple `Dictionary` beans, the context would have tried to match the beans *by name*, that is, match the name given by the ID in the XML definition to the name of the parameter in the constructor. If there were no match, an exception would be thrown; again, a `NoSuchBeanDefinitionException`.

Again, it cannot be stressed enough that if you are using the XML configuration approach, you *must* have a good suite of integration tests to ensure that your beans are constructed properly.

Listing 16-9 and Listing 16-10 illustrated a common pattern for everyday use of the Spring application context: using XML context to define the beans, and then autowiring to integrate these beans.

Another use of autowiring is to annotate dependency to be autowired. Listing 16-11 shows how this looks on the `SpellCheckApplication` constructor.

LISTING 16-11: Using the `@Autowired` annotation on a constructor

```
@Autowired
public SpellCheckApplication (final Dictionary dictionary) {
    this.dictionary = dictionary;
}
```

Using the annotation in this way, you do not need to have an `autowired` attribute on the XML bean definition.

You can even autowire fields within a class, without a publicly accessible constructor or setter method. Listing 16-12 autowires the `Dictionary` instance on to a private field. There is no setter for the `Dictionary` in the class, and the constructor can be removed.

LISTING 16-12: Using `@Autowired` on fields

```
@Autowired
private Dictionary dictionary;
```

Note that this field cannot be `final` anymore. The application context will access and set this field after it has been constructed, using reflection.

If your context contains more than one bean of the same type, you can still use the `@Autowired` annotation, but you need to be explicit as to which bean you mean. A second annotation, `@Qualifier`, is used for this. The annotation's constructor takes a string, the name of the bean you require for wiring:

```
@Autowired
@Qualifier("englishDictionary")
private Dictionary dictionary;
```

These annotations are specific to Spring. Should you ever decide to change containers for a different provider, such as Google's Guice, then, although the concepts for your application code do not change, you would need to re-implement your dependency injection and bean creation code to work with their implementation. A specification called JSR-330 worked to standardize the annotations across different providers. Instead of using `@Autowired`, you can use `@Inject` instead.

ABOUT JAVA SPECIFICATION REQUESTS (JSRS)

JSRs are requests from the Java development community, and other interested parties, to add new functionality to upcoming releases of the Java platform.

Past JSRs have included the introduction of the `assert` keyword in Java 1.4, the specification of the Java Message Service (JMS), and JavaServer Pages (JSPs).

However you decide to construct your beans and manage their dependencies, it is important to remain consistent. As your application grows, it can be hard to manage these dependencies if they are set up in several different ways, especially if you do not have good test coverage.

SPRING JDBC

How does Spring improve the readability of JDBC code?

Listing 16-13 is an example of how to use JDBC using APIs shipped as part of the Java Development Kit. Database access and JDBC in general are covered in Chapter 12.

LISTING 16-13: Using the JDK APIs for database access

```
@Test
public void plainJdbcExample() {
    Connection conn = null;
    PreparedStatement insert = null;
    PreparedStatement query = null;
    ResultSet resultSet = null;
    try {
        Class.forName("com.mysql.jdbc.Driver");

        conn = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/springtrans?" +
            "user=nm&" +
            "password=password");

        insert = conn.prepareStatement("insert into user values (?, ?)");

        final List<String> users = Arrays.asList("Dave", "Erica", "Frankie");

        for (String user : users) {
            insert.setString(1, UUID.randomUUID().toString());
```

continues

LISTING 16-13 *(continued)*

```

        insert.setString(2, user);

        insert.executeUpdate();
    }

    query = conn.prepareStatement("select count(*) from user");
    resultSet = query.executeQuery();

    if (!resultSet.next()) {
        fail("Unable to retrieve row count of user table");
    }

    final int rowCount = resultSet.getInt(1);

    assertTrue(rowCount >= users.size());
} catch (ClassNotFoundException | SQLException e) {
    fail("Exception occurred in database access: " + e.getMessage());
} finally {
    try {
        if (resultSet != null) {
            resultSet.close();
        }
        if (query != null) {
            query.close();
        }
        if (insert != null) {
            insert.close();
        }
        if (conn != null) {
            conn.close();
        }
    } catch (SQLException e) {
        fail("Unable to close Database connection: " + e.getMessage());
    }
}
}

```

The prevalent matter in this listing is how much code is necessary to insert some data, and verify that the data is there. All of the heavyweight classes needed for database access, such as `Connection`, `PreparedStatement`, and `ResultSet` can all throw checked `SQLExceptions` on their creation, use, and destruction. These exceptions cannot simply be ignored: Should a `close` method throw a `SQLException`, then that connection to the database may still exist. If this happens several times, all of the connections in the database connection pool will be in use, and any subsequent requests for database access will fail, bringing your entire application to a messy halt.

So using the `java.sql.*` classes results in a pattern of surrounding all your database access in a `try` block, and closing the database statements, result sets, and connections inside a `finally` block. But because these `close` statements inside the `finally` block can fail too, they need to be in a `try` block. If that close fails, there isn't really much you can do, besides being aware that it has happened, and manually killing the database connection to the running program.

Moreover, because the database resource objects need to be initialized inside the `try` block but closed inside the `finally` block, the reference variables cannot be assigned as final variables, which runs the risk (through programming error) of reassigning to these variables before any chance of calling their close method.

Thankfully, Spring provides some utilities for taking care of some of this boilerplate approach. Listing 16-14 provides an alternative `finally` block for Listing 16-13.

LISTING 16-14: A friendlier way to close database resources

```
try {
    // as Listing 16-13
} catch (ClassNotFoundException | SQLException e) {
    // as Listing 16-13
} finally {
    JdbcUtils.closeResultSet(resultSet);
    JdbcUtils.closeStatement(query);
    JdbcUtils.closeStatement(insert);
    JdbcUtils.closeConnection(conn);
}
```

You are encouraged to take a look at the implementation of these utility functions. They contain the same code as the `finally` block in Listing 16-13, but throw no checked exceptions. This is a huge reduction in necessary boilerplate code; they make your own code shorter and much easier to read and reason what is happening.

The `JdbcUtils` class provides some other useful functions, too, such as extracting results from a `ResultSet`, or even converting from a database naming convention (that is, `separated_by_underscores`) to Java's naming convention (`definedWithCamelCase`).

Using the try-with-resources statement, introduced in Java 8 and discussed in Chapter 8, can also reduce the boilerplate code here, however you must still have a catch block for a `SQLException`, as that can be thrown when creating the resource.

How can Spring remove most boilerplate JDBC code?

The `JdbcUtils` class is intended for use as an enhancement to plain old JDBC code. If you are starting any new development in Java code that needs JDBC access, you should use Spring's `JdbcTemplate`.

Spring's approach to using JDBC abstracts much of the necessary setup to your application context, and therefore removes it from polluting any program logic. Once the template is set up, for executing queries, for instance, you only need to provide functionality for converting each row from a `ResultSet` to a specific domain object. As its name suggests, this is an application of the Template design pattern, which was covered in a more abstract sense in Chapter 6.

Listing 16-15 and Listing 16-16 show how the `JdbcTemplate` can be used for everyday database manipulations.

LISTING 16-15: Configuring a database in an application context

```

<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:3306/springtrans" />
    <property name="username" value="nm" />
    <property name="password" value="password" />
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <constructor-arg ref="dataSource"/>
</bean>

```

LISTING 16-16: Using the SimpleJdbcTemplate

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
    "classpath:com/wiley/javainterviewsexposed /chapter16/jdbcTemplateContext.xml")
public class JdbcTemplateUsage {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Test
    public void retrieveData() {
        final int rowCount =
            jdbcTemplate.queryForInt("select count(*) from user");

        final List<User> userList = jdbcTemplate.query(
            "select id, username from user",
            new RowMapper<User>() {

                @Override
                public User mapRow(ResultSet rs, int rowNum) throws SQLException {
                    return new User(rs.getString("id"), rs.getString("username"));
                }
            });

        assertEquals(rowCount, userList.size());
    }
}

```

This code accesses the database twice. The method `queryForInt` expects the given query to return one row, with one column, and the type of that column is some kind of numerical type, which can be interpreted as an `int`. Therefore, you do not need to worry about creating a statement, performing the query, and obtaining the `ResultSet`, making sure the `ResultSet` has one row and one column, and closing all resources afterwards.

The second query shows a slightly more complex approach, where each row in the database is translated into a `User` domain object. The template is able to perform the query and iterate over the results. The only thing it does not know how to do is to transform each row into the object you desire. You provide an anonymous implementation of the `RowMapper` interface, and the `mapRow` is called for each row in the database. You have access to the data and metadata (by calling the method `ResultSet.getMetaData()`), and also the row number.

Each object returned from the `mapRow` call is collected by the template and collated in a `List` of those mapped instances.

USING THE JDBCTEMPLATE IN JAVA 8

Because the `RowMapper` interface has only one method, this fits nicely with the new Lambda functionality, which should appear in Java 8 (see the section “Looking Forward to Java 8” in Chapter 8). This will make the query method even clearer and more concise than it already is:

```
jdbcTemplate.query(queryString, (rs, rowNum) -> {
    return new User(rs.getString("id"), rs.getString("username"));
});
```

This requires no change to any Spring library code.

Again, you do not need to deal with opening database connections or any of the heavy database resource objects, and you do not need to close any resources after your query—the template does all this for you. It should be clear that this code is more concise, clearer, and much more maintainable than the approach shown earlier in Listing 16-13.

INTEGRATION TESTING

How do you test that an application context is configured correctly?

In the earlier section on using the application context, having to load the `ApplicationContext` object manually for each test was becoming unwieldy and distracting, taking the focus away from the test.

Spring has a good integration for using application contexts in integration tests. Listing 16-17 revisits the spell-checking application, showing how Spring helps write integration tests.

LISTING 16-17: Using Spring’s integration testing

```
@ContextConfiguration(
    "classpath:com/wiley/javainterviewsexposed/chapter16/applicationContext.xml")
@RunWith(SpringJUnit4ClassRunner.class)
```

continues

LISTING 16-17 *(continued)*

```

public class SpringIntegrationTest {

    @Autowired
    private SpellCheckApplication application;

    @Test
    public void checkWiring() {
        assertNotNull(application);
    }

    @Test
    public void useSpringIntegrationTests() {

        final List<String> words = Arrays.asList(
            "correct",
            "valid",
            "dfgluharg",
            "acceptable");
        final List<Integer> expectedInvalidIndices = Arrays.asList(2);

        final List<Integer> actualInvalidIndices =
            application.checkDocument(words);

        assertEquals(expectedInvalidIndices, actualInvalidIndices);
    }
}

```

The class is annotated to run with the `SpringJUnit4ClassRunner`, which understands how to load a Spring application context through the `@ContextConfiguration` class-level annotation.

Running tests this way allows the beans under test to be autowired for use in a particular test. Any `@Autowired` beans are reloaded from the application context for each test; similar to the way the `@Before` annotated methods are run before each test.

If you have a sufficiently complex test, which may change the state of the application context, you can annotate your class with `@DirtiesContext`, and this will reload the whole context from scratch for each test. This can significantly increase the running time for all the tests in a defined test suite.

How do you keep databases clean when running integration tests?

The Spring integration testing functionality has a remarkably useful feature for working with databases and transactions. If you have a JUnit test suite, which extends `AbstractTransactionalJUnit4SpringContextTests`, then each of the tests run inside this suite will be run inside a database transaction, and will be rolled back at the end of each test.

This has the useful capability to check that your code meets any requirements specified by your database, such as referential integrity, or any complex knock-on effects such as those specified in a database trigger. Listing 16-18 shows a test that appears to insert data into a database. The test will fail if the precondition of any of the usernames to insert is already in the database.

LISTING 16-18: Using Spring's transactional support for integration tests

```
@ContextConfiguration(
    "classpath:com/wiley/javainterviewsexposed/" +
        "chapter16/databaseTestApplicationContext.xml")
public class SpringDatabaseTransactionTest extends
    AbstractTransactionalJUnit4SpringContextTests {

    private int rowCount;

    @Before
    public void checkRowsInTable() {
        this.rowCount = countRowsInTable("user");
    }

    @Before
    public void checkUsersForTestDoNotExist() {
        final int count = this.simpleJdbcTemplate.queryForInt(
            "select count(*) from user where username in (?, ?, ?)",
            "Alice",
            "Bob",
            "Charlie");
        assertEquals(0, count);
    }

    @Test
    public void runInTransaction() {

        final List<Object[]> users = Arrays.asList(
            new Object[]{UUID.randomUUID().toString(), "Alice"},
            new Object[]{UUID.randomUUID().toString(), "Bob"},
            new Object[]{UUID.randomUUID().toString(), "Charlie"}
        );

        this.simpleJdbcTemplate.batchUpdate(
            "insert into user values (?, ?)", users);

        assertEquals(rowCount + users.size(), countRowsInTable("user"));
    }
}
```

Before any of the `@Before` methods are run, a transaction is opened. Once the test has been run, and all of the `@After` methods have completed, the transaction is rolled back, as if the data had never been there. Naturally, you need to make sure that none of your code in the `@Test` method performs a commit.

Had this test been run outside of a transaction, it would fail on its second run, due to the insertion of the usernames Alice, Bob, and Charlie. But because the insertion is rolled back, it can be run time and time again with no fear of the data persisting. This makes it suitable for running as part of a regular application build, or as part of a set of integration tests, run on a continuous delivery server.

JUnit 4.11 introduced the `@FixMethodOrder` annotation, which allows you to define the order your `@Test` methods are run. In Listing 16-18, you could write another test that double-checks that the transaction had indeed been rolled back and the users did not exist in the table, and have that test run after the `runInTransaction` test.

Specifying the order of tests is bad practice, as you should be able to run each test in isolation. If this test was run on a continuous delivery server, and there was a problem with the transaction rollback, then the test would fail on its second run, which would be early in the development cycle if you have a regular running build.

By extending `AbstractTransactionalJUnit4SpringContextTests`, and providing an application context with a valid `DataSource`, the integration test suite gives you access to a `SimpleJdbcTemplate` object, with which you can perform any relevant database operations for your test. The test suite also provides a couple of utility methods: `countRowsInTable` and `deleteFromTables`.

SPRING MVC

The Model-View-Controller (MVC) pattern is a common approach to working with user interfaces. As its name implies, it breaks up an application into three parts.

The model is a representation of the data your application is trying to work with.

The view presents the user with a representation of this model. Depending on the nature of the application, this could simply be text on a screen, or perhaps a more complicated visualization using graphics and animation.

The controller joins everything up. It is responsible for processing any input, which will govern how the model is created, and then it passes this model on to an appropriate view.

This section demonstrates how Spring's web applications use this pattern to provide a clean separation between these three parts and how they fit together, building a small web application for displaying sports results. Listing 16-19 shows the interface to the sports results service.

LISTING 16-19: The `SportsResultsService` interface

```
public interface SportsResultsService {  
    List<Result> getMostRecentResults();  
}
```

How does Spring serve web applications?

Spring's MVC framework is designed to be compatible with the Java Servlet specification. Many providers and vendors of applications use this specification, such as freely available Apache Tomcat, and Jetty, and also commercial implementations like IBM's Websphere Application Server. All of these can be used with Spring MVC.

When the application server starts, it looks for the file `web.xml`, in the `META-INF` package on the classpath. This file specifies what applications, or *servlets*, the server will provide. Inside `web.xml` you provide a class that extends `HttpServlet`, and that defines how to respond to the HTTP methods such as GET and POST.

Although you can provide your own implementation, to use Spring MVC, you use its implementation, `org.springframework.web.servlet.DispatcherServlet`. Listing 16-20 gives a complete definition of a sample `web.xml` file, passing any request starting with `/mvc` to the `DispatcherServlet`.

LISTING 16-20: Configuring a servlet container for Spring MVC

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    metadata-complete="false"
    version="3.0">

    <servlet>
        <servlet-name>mvc</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>mvc</servlet-name>
        <url-pattern>/mvc/*</url-pattern>
    </servlet-mapping>

</web-app>
```

When control is passed to the `DispatcherServlet`, it immediately looks for an application context XML definition, which is held at the classpath location `WEB-INF/[servlet-name]-servlet.xml`. For Listing-16-20, it will be called `mvc-servlet.xml`.

This file is a regular, familiar, Spring application context. You can specify any Spring beans here, and they will be instantiated when the server starts. Listing 16-21 is a very simple application context.

LISTING 16-21: An application context for a sample Spring MVC application

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/context
         http://www.springframework.org/schema/context/spring-context.xsd">

  <context:component-scan
    base-package="com.wiley.javainterviewsexposed.chapter16.mvc"/>

  <bean id="freemarkerConfig"
        class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
    <property name="templateLoaderPath" value="/WEB-INF/freemarker/" />
  </bean>

  <bean
    id="viewResolver"
    class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver">
    <property name="prefix" value="" />
    <property name="suffix" value=".ftl" />
  </bean>

  <bean
    class="com.wiley.javainterviewsexposed.chapter16.mvc.DummySportsResultsService"/>

</beans>
```

There are three main actions here. Most Spring MVC controllers are defined using annotations, so the `context:component-scan` tag informs the application context of which packages to scan for classes annotated with `@Controller`. There is the setup of the view resolver, which is used to render and display the results of any request; Spring MVC will request the bean of type `ViewResolver` when it needs to display the output of a request. Finally, this application has one application-specific bean, an instance of `DummySportsResultsService`.

This is just a regular application context. You can use it outside of an MVC application if you need to, such as with Spring's integration testing framework.

How can you create easily testable web controllers with Spring MVC?

Controllers in Spring are lightweight, Plain Old Java Objects. By having very few dependencies, they have been designed to be easily testable, and the framework gives many options for their behavior

to fit in with your way of working. Listing 16-22 is the simple example for Spring's controllers in its MVC framework.

LISTING 16-22: A simple Spring controller

```
@Controller
@RequestMapping("/results")
public class ResultsController {

    private final SportsResultsService resultsService;

    @Autowired
    public ResultsController(final SportsResultsService resultsService) {
        this.resultsService = resultsService;
    }

    @RequestMapping(value = "/recent")
    public String getMostRecentResults(final Model model) {
        model.addAttribute("results", resultsService.getMostRecentResults());
        return "resultsView";
    }

    @RequestMapping("/recent/{team}")
    public String getMostRecentResultsForTeam(
        @PathVariable("team") final String team,
        final Model model) {
        final List<Result> results = resultsService.getMostRecentResults();
        final List<Result> resultsForTeam = new ArrayList<Result>();
        for (Result result : results) {
            if (Arrays.asList(
                result.getHomeTeam(),
                result.getAwayTeam()
            ).contains(team)) {
                resultsForTeam.add(result);
            }
        }

        if (resultsForTeam.isEmpty()) {
            return "teamNotFound";
        } else {
            model.addAttribute("results", resultsForTeam);
            return "resultsView";
        }
    }
}
```

This controller provides GET access to two URLs, `/results/recent` and `/results/recent/<team>`, for a specific team.

Before looking at the mechanics of how this controller integrates with a servlet engine, it should be clear that this class is easily testable as a unit test. Listing 16-23 shows a test for the method `getMostRecentResultsForTeam`.

LISTING 16-23: A unit test for the ResultsController

```

@Test
public void specificTeamResults() throws Exception {
    final SportsResultsService mockService = mock(SportsResultsService.class);

    final List<Result> results = Arrays.asList(
        new Result("Home", "Away", 1, 2),
        new Result("IgnoreHome1", "IgnoreAway1", 0, 0)
    );

    when(mockService.getMostRecentResults()).thenReturn(results);

    final List<Result> expectedResults = Arrays.asList(
        new Result("Home", "Away", 1, 2));

    final ResultsController controller = new ResultsController(mockService);
    final Model model = new ExtendedModelMap();

    controller.getMostRecentResultsForTeam("Home", model);

    assertEquals(expectedResults, model.asMap().get("results"));
}

```

Examining Listing 16-22, the class is annotated with the `@Controller` annotation, so the component-scan directive in the servlet's application context picks it up.

This controller is specific for any URL starting in `/results`, so that, too, can be annotated at the class level, with the `@RequestMapping` annotation. Any method-level `@RequestMapping` annotations will be relative to this.

The `getMostRecentResultsForTeam` method has a parameter annotated with `@PathVariable`, given the value `team`. This value corresponds to the `{team}` binding in the `@RequestMapping` annotation. This allows for a *RESTful* approach if you so wish; it is very simple to parameterize your request URLs. This is type-safe too; Spring can interpret the value as any simple type, such as the primitive types `Strings` and `Dates`, and it is possible for you to provide your own conversion from the `String` in the request path to your own complex type.

Methods on controllers have the single role of providing the view technology with the model to render.

The two methods have similar signatures. Both take at least one parameter, of a `Model` object, and both return `Strings`.

The `Model` parameter is passed to the controller method by the framework. At its core, the `Model` object is a mapping from `Strings` to `Objects`. The controller performs the necessary work, such as retrieving data from a service accessing a database or third-party web application, and populates the given `Model`. Remember, Java objects are references, so the framework can still reference the model instance after you have updated it, meaning you do not need to return your updated model from the method.

This allows the method to return another necessary component, the name of the view to render. You have the ability to configure Spring's MVC framework so that it understands how to map a `String` to a particular view; look ahead to the next question for how this works.

To allow for as much flexibility as possible, and to fit with your own preference of how to define your own controller, the Spring MVC framework understands many different configurations for your method definition. You can specify several annotations on the method parameters, such as `@RequestParam`, for multipart form uploads, or `@RequestHeader`, for assigning a specific header to a variable.

You can also specify many different parameter types for the method signature, including:

- `InputStream`, providing access to the raw HTTP request stream
- `OutputStream`, allowing you to write directly to the HTTP response stream
- `HttpServletRequest`, for the exact request object itself
- `ServletResponse`, another facility for writing directly to the response

Additionally, you have several options for your return type, such as `ModelAndView`, an object containing both the model for the view and the view name, all contained in one object. You can even use a void return type if you have written directly to the response stream.

BE SMART: KEEP YOUR CODE SIMPLE AND EASY TO TEST

Remember to use tests as often as you can in any interview situation. Like all Java classes, it is advisable to keep your controllers simple and easily testable. It can be quite difficult to create testable controller methods if you are writing directly to the response output stream, and your method has a void return type.

How is the server-side logic kept separate from presenting the results of a request?

Views are how the work done by the controller, represented by the model, is displayed.

The view should simply be displaying the output; the controller should have done all meaningful work before passing the model to the view. This includes operations such as sorting lists or filtering any unwanted content from a data structure.

In essence, most view technologies that work with Spring follow a very similar pattern: They work as templates, with placeholders for dynamic data. Spring integrates tightly with several implementations so that their own integration with Spring's controllers stays the same.

The usual format is for the controller to populate a mapping, consisting of a name to either a "bean" or data structure. For the bean, the properties, defined on the Java class by `getxxx` methods, are then directly accessible in the view technology. For data structures, such as a `Map` or `List`,

the template provides ways to iterate over the collection and access each element, again as a bean. Listing 16-24 is a very simple controller, and Listing 16-25 is a possible view for displaying that data. The view implementation here is called Freemarker.

LISTING 16-24: A simple controller for use with Listing 16-25

```
@RequestMapping("/planets")
public String createSimpleModel(final Model model) {

    model.addAttribute("now", new Date());

    final Map<String, String> diameters = new HashMap<>();
    diameters.put("Mercury", "3000 miles");
    diameters.put("Venus", "7500 miles");
    diameters.put("Earth", "8000 miles");

    model.addAttribute("diameters", diameters);

    return "diameters";
}
```

LISTING 16-25: Using a view template to display the output from a controller

```
<html>
<head>
    <title>Planet Diameters</title>
</head>
<body>
<#list diameters?keys as planet>
    <b>${planet}</b>: ${diameters[planet]}<br/>
</#list>
This page was generated on <i>${now?datetime}</i>
</body>
</html>
```

The controller has added two elements to the model: `now`, a representation of the current time, and `diameters`, the map of planet names to a description of their diameter.

Listing 16-25 is written as HTML, with slots for the dynamic content to fit. This has several advantages, including a clear separation between the server-side processing and the front-end display. Also, notice that the template is not restricted to HTML: Anything at all could have been displayed, perhaps an XML or comma-separated values (CSV) document, or even a template for use in an e-mail.

The `<#list>` tag is known as a macro; Freemarker has many of these for different situations. It is being used here to iterate through the keys in the `diameters` map. Other examples include `bool?string("yes", "no")`, which is used to map the value `bool` onto one of two strings; `yes` or `no`, and `number?round`, which will display a rounded version of the value `number`.

How can Spring be configured to work with different view template engines?

Listing 16-25 uses the Freemarker engine. The controller in Listing 16-24 references the view by the name `diameters`. The mapping from name to template file is set up in the application context, and is highly dependent on the implementation of `ViewResolver`. For this application, which was originally set up in Listing 16-21, the view resolver was set as:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver">
    <property name="prefix" value="" />
    <property name="suffix" value=".ftl" />
</bean>
```

This specifies that the view template files are stored in the default location, and have the file extension `.ftl`. When using Freemarker, the `diameters` view should be found on the classpath as `/WEB-INF/freemarker/diameters.ftl`.

A similar, older view technology is Velocity, and Spring's view resolver bean uses similar bean properties:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.velocity.VelocityViewResolver">
    <property name="prefix" value="" />
    <property name="suffix" value=".vm" />
</bean>
```

Convention follows that Velocity templates have a `.vm` suffix.

SUMMARY

Whatever parts of Spring you do and do not use, the core concept for most approaches is the use of the application context. Understanding dependency injection and how components fit together is extremely important, and any interview involving Spring will cover Spring beans, how they are created, and how they interact among themselves.

This chapter has not been an exhaustive look at the Spring Framework. Do take the time to write some code with Spring, and see how the concepts here translate to other parts, such as how to interact with Hibernate using the `HibernateTemplate`, or how the application context is used in more complicated scenarios with Spring Integration.

Make sure that anything you write in Spring is tested, especially if you are configuring your applications in XML. Spring gives you good support for creating comprehensive tests. Eclipse or IntelliJ have good plug-ins for Spring; they can often highlight any errors in Spring configuration straight away—this may not always be apparent until you run the code and attempt to construct the application context.

Chapter 17 covers Hibernate, a framework that is often paired with Spring. Hibernate enables you to convert database tables directly into a Java object, and vice-versa. The chapter looks at the basics of Hibernate's different approaches that you can take to configure and use it.

17

Using Hibernate

Hibernate is a tool built specifically for managing the mapping between Java objects and database tables, usually referred to as *Object/Relational Mapping*, or *ORM*.

This chapter builds upon a sample database schema for a box office application. The application manages tickets for customers for different events, at different venues. Figure 17-1 shows the structure of the database.

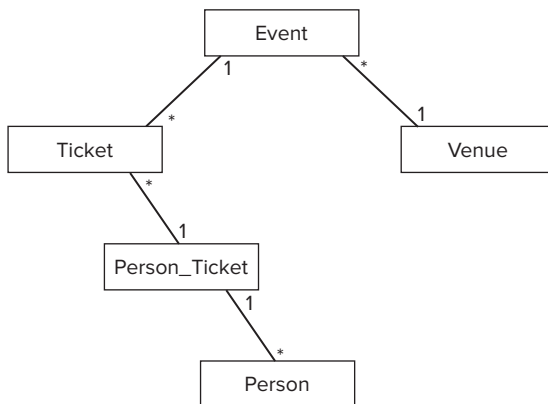


FIGURE 17-1

USING HIBERNATE

How do you create Java objects using Hibernate?

Hibernate is a tool for mapping database relations to objects, so it makes sense to have domain objects for these entities. The classes are simple *Plain Old Java Objects*, or *POJOs*, and they simply match their corresponding database table. For instance, Listing 17-1 shows the class for the `people` table.

LISTING 17-1: The Person class

```
public class Person {

    private int id;
    private String name;
    private String address;

    public Person() {
        // default no-arg constructor
    }

    public Person(int id, String name, String address) {
        this.id = id;
        this.name = name;
        this.address = address;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    @Override
    public String toString() {
        return "Person{" +
            "name='" + name + '\'' +
            ", address='" + address + '\'' +
            '}';
    }

    // equals and hashCode omitted
}
```

Note the name of the class: `Person`. This refers to the `people` table. The convention is for database tables to have the pluralized form of the word, and the class definition to have the singular form. The class represents a single person.

A couple of other points to note here are that you must include a default, no-arg constructor. When Hibernate is building objects for you, it creates the objects using this constructor. If there were no default, it would be simply not possible for Hibernate to decide which constructor to use. Hibernate will populate the instance using the setter methods, so you will need a default constructor, and the fields of the `Person` class must be non-final.

You need to specify a mapping between the class and the table. Listing 17-2 shows this for the `Person` class.

LISTING 17-2: The mapping for the `Person` entity

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.wiley.acinginterview.chapter17">

    <class name="Person" table="PEOPLE">
        <id name="id" column="ID"/>
        <property name="name"/>
        <property name="address"/>
    </class>

</hibernate-mapping>
```

The naming convention for this file is `<Entity>.hbm.xml`, so this file would be called `Person.hbm.xml`. For this simple example, the fields of the class match the columns of the database table, so this mapping merely names the properties of the `Person` entity, and describes that the `id` field works as the unique identifier for the different objects.

You also need to tell Hibernate what database you are using and its location. By convention, this is stored in the root of the classpath, in a file called `hibernate.cfg.xml`. This file also contains a reference to the mapping file shown in Listing 17-2. Listing 17-3 shows a simple configuration file.

LISTING 17-3: A sample `hibernate.cfg.xml` file

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="connection.driver_class">
            com.mysql.jdbc.Driver
```

continues

LISTING 17-3 *(continued)*

```

    </property>
    <property name="connection.url">
        jdbc:mysql://localhost:3306/ticketoffice
    </property>
    <property name="connection.username">nm</property>
    <property name="connection.password">password</property>

    <property name="dialect">
        org.hibernate.dialect.MySQLDialect
    </property>

    <property name="show_sql">true</property>

    <mapping
        resource="com/wiley/javainterviewsexposed/chapter17/Person.hbm.xml"/>
</session-factory>
</hibernate-configuration>

```

This file is where you define the properties about the connection to the database, such as the size of your connection pool, or if you want to use a cache.

If the rows in the people table looked like this:

```
mysql> select * from people;
+-----+-----+-----+
| id | name           | address           |
+-----+-----+-----+
| 1 | Eric Twinge    | 29, Acacia Avenue |
| 2 | Dominic Taylor | 74A, High Road    |
| 3 | Alice Smith    | 1, Pine Drive     |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

Listing 17-4 shows how your configuration is then used to extract rows from that table into Person objects.

LISTING 17-4: Querying the database with Hibernate

```

@Test
public void retrievePeople() {
    final SessionFactory sessionFactory =
        new Configuration()
            .configure()
            .buildSessionFactory();

    final Session session = sessionFactory.openSession();
    final List<Person> actual = session
        .createQuery("from Person")

```



```

        .list();

    final List<Person> expected = Arrays.asList(
        new Person(1, "Eric Twinge", "29, Acacia Avenue"),
        new Person(2, "Dominic Taylor", "74A, High Road"),
        new Person(3, "Alice Smith", "1, Pine Drive"));

    assertEquals(expected, actual);
}

```

Notice, too, that once the `Person` objects have been populated by Hibernate, there is no dependency on Hibernate or the database; they are just regular POJOs.

How do you insert data into a database using Hibernate?

Inserting data into a database using Hibernate is just as simple. Listing 17-5 inserts data using Hibernate. Verification that the row is actually in the database is done using regular SQL.

LISTING 17-5: Persisting data using Hibernate

```

@Test
public void storePerson() throws SQLException {
    final Session session = sessionFactory.openSession();
    final Transaction transaction = session.beginTransaction();

    final Person newPerson =
        new Person(4, "Bruce Wayne", "Gotham City");
    session.save(newPerson);
    transaction.commit();

    final Connection conn = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/ticketoffice",
        "nm",
        "password");

    final Statement stmt = conn.createStatement();

    final ResultSet rs = stmt.executeQuery(
        "select name from people where id = 4");
    assertTrue(rs.next());
    assertEquals("Bruce Wayne", rs.getString(1));

    rs.close();
    stmt.close();
    conn.close();
}

```

By simply calling `save` on the Hibernate session with a `Person` object, that `Person` object is persisted to the database.

Note that this code is still dependent on the `id` field of the `Person` object being unique. Trying to insert another `Person` object with an `id` of 4 will fail. A couple of small changes will make your code less dependent on any persistence mechanism.

If your table is set to automatically generate identifiers, and Hibernate is aware of this, you do not need to supply the `id` when persisting objects. Listing 17-6 shows the DDL statement for a MySQL database for the `people` table.

LISTING 17-6: An auto-incrementing identifier

```
CREATE TABLE people (  
    id INTEGER AUTO_INCREMENT PRIMARY KEY,  
    name VARCHAR(100),  
    address VARCHAR(100)  
);
```

The `AUTO_INCREMENT` attribute will create the identifier for you.

You also need to provide a generator in your `Person.hbm.xml` Hibernate mapping, as shown in Listing 17-7.

LISTING 17-7: Adding a generator to the Hibernate mapping

```
<class name="Person" table="PEOPLE">  
    <id name="id" column="ID">  
        <generator class="native"/>  
    </id>  
    <property name="name"/>  
    <property name="address"/>  
</class>
```

The attribute chosen here is `native`; that is, it is expected that the database will automatically insert an identifier. Other generator classes exist, such as one for creating UUIDs or allowing Hibernate to keep track of the identifiers itself.

With these changes, the generated SQL will change. If you have kept the `show_sql` property switched on in your `hibernate.cfg.xml` file, you will see the following log line:

```
Hibernate: insert into PEOPLE (name, address) values (?, ?)
```

There is no reference to the `id` column. This means that you do not need an `id` field in your `Person` class at all, as shown in Listing 17-8.

LISTING 17-8: A cleaner Person class

```
public class Person {  
  
    private String name;  
    private String address;  
  
    ... the rest as before, without the id getters and setters
```

This has resulted in a clearer domain object, with no notion at all of persistence.

How do you join database tables with Hibernate?

When trying to implement the schema provided in Figure 17-1, having the `Person` class on its own isn't very useful. The diagram stipulates that the relationship between tickets and people is *many-to-many*, and as such, a join table is used.

For the following examples, the data in `tickets` and `person_ticket` is as follows:

```
mysql> select * from tickets;
+----+-----+
| id | event_id |
+----+-----+
| 1 | 1 |
| 2 | 1 |
| 3 | 2 |
| 4 | 2 |
| 5 | 3 |
| 6 | 3 |
| 7 | 4 |
| 8 | 5 |
| 9 | 5 |
| 10 | 6 |
+----+-----+
10 rows in set (0.00 sec)
```

```
mysql> select * from person_ticket;
+-----+-----+
| person_id | ticket_id |
+-----+-----+
| 1 | 1 |
| 2 | 4 |
| 1 | 7 |
| 3 | 10 |
+-----+-----+
4 rows in set (0.00 sec)
```

You can see here that one person has tickets to many events, (`person_id` 1), and events can have more than one ticket (`event_ids` 1, 2, 3, and 5).

It would be useful to modify the `Person` class to hold a collection of tickets. That way, a person retrieved from the database will hold a list of `Ticket` objects, which may be useful if the person ever needs to ask the ticket office which events he has tickets for.

You will need to create a `Ticket` class. It has not been written out here, because there is nothing unexpected in this class. It will hold two fields—`id` and `eventId`—and will have a no-arg constructor, as well as getters and setters for all the fields. Listing 17-9 shows two Hibernate mapping files: a new `Ticket.hbm.xml` and a modified `Person.hbm.xml`.

LISTING 17-9: Hibernate mappings for associating tickets and people

```
<hibernate-mapping package="com.wiley.javainterviewsexposed.chapter17">

    <class name="Ticket" table="TICKETS">
        <id name="id" column="ID"/>
        <property name="eventId" column="event_id"/>
    </class>

</hibernate-mapping>

<hibernate-mapping package="com.wiley.javainterviewsexposed.chapter17">

    <class name="Person" table="PEOPLE">
        <id name="id" column="ID">
            <generator class="native"/>
        </id>
        <property name="name"/>
        <property name="address"/>

        <set name="tickets" table="PERSON_TICKET" cascade="all">
            <key column="person_id"/>
            <many-to-many column="ticket_id"
                        unique="true"
                        class="Ticket"/>
        </set>

    </class>

</hibernate-mapping>
```

The `Ticket` Hibernate mapping is pretty much as you would expect. The only new piece of information is that you need to map the `eventId` field to the `event_id` column—all the examples previously had the exact same names. This mapping enables you to keep the appropriate naming conventions for both Java and SQL.

The `Person` Hibernate mapping holds one new piece of information, a `set` for the `tickets` field. This references the `person_ticket` table. The `key-column` tag instructs Hibernate how to perform its SQL join—the `person_id` column of `person_ticket` is the join criteria, and the `many-to-many` tag references the identifier of the `Ticket` entity.

You need to reference the new `Ticket.hbm.xml` file in your `hibernate.cfg.xml` Hibernate configuration file, and then you have one more change to make: You need to add a collection field to your `Person` class, as shown in Listing 17-10.

LISTING 17-10: Additions to the Person class

```
public class Person {

    private int id;
    private String name;
```

```

private String address;
private Set<Ticket> tickets;

public Person() {
    // default no-arg constructor
}

public Person(int id,
              String name,
              String address,
              Set<Ticket> tickets) {
...

```

The rest of the class is as expected. The `tickets` field has appropriate getters and setters, the `equals` and `hashCode` methods take the collection into account, and the `toString` method makes sure to include the values in the set.

Listing 17-11 shows an updated test confirming that retrieving `Person` objects now includes a set of tickets.

LISTING 17-11: Testing joining entities using Hibernate

```

@Test
public void retrievePeople() {
    final SessionFactory sessionFactory =
        new Configuration()
            .configure()
            .buildSessionFactory();

    final Session session = sessionFactory.openSession();
    final List<Person> actual = session
        .createQuery("from Person")
        .list();

    final Set<Ticket> ericTickets = new HashSet<Ticket>(2) {{
        add(new Ticket(1, 1));
        add(new Ticket(7, 4));
    }};

    final Set<Ticket> dominicTickets = new HashSet<Ticket>(1) {{
        add(new Ticket(4, 2));
    }};

    final Set<Ticket> aliceTickets = new HashSet<Ticket>(1) {{
        add(new Ticket(10, 6));
    }};

    final List<Person> expected = Arrays.asList(
        new Person(1, "Eric Twinge", "29, Acacia Avenue", ericTickets),
        new Person(2, "Dominic Taylor", "74A, High Road", dominicTickets),
        new Person(3, "Alice Smith", "1, Pine Drive", aliceTickets));

    assertEquals(expected, actual);
}

```

Of course, working with just identifiers isn't much use. The pattern to include `Tickets` on the `Person` class can be repeated to include `Events` on the `Ticket` class, but this time the relationship is many-to-one: A ticket can only be for one event, though a single event will clearly sell more than one ticket.

You must create both a new `Event` class containing the fields from the event table, and a vanilla `Events.hbm.xml` Hibernate mapping to map columns to class fields. Modify the `Ticket.hbm.xml` file to have a many-to-one field, which matches with the event table, as shown in Listing 17-12.

LISTING 17-12: A many-to-one relationship

```
<hibernate-mapping package="com.wiley.acinginterview.chapter17">

    <class name="Ticket" table="TICKETS">
        <id name="id" column="ID"/>
        <many-to-one name="event" class="Event"
            column="event_id" cascade="all" not-null="true"/>
    </class>

</hibernate-mapping>
```

Listing 17-13 shows the change to the test, now that the `Ticket` class constructor takes an `int id` and an `Event` object.

LISTING 17-13: Testing the many-to-one relationship

```
final Set<Ticket> dominicTickets = new HashSet<Ticket>(1) {{
    add(new Ticket(4,
        new Event(
            2,
            1,
            "The Super Six",
            new DateTime(2013, 8, 1, 0, 0).toDate())));
}};
```

The test should still pass; it will populate the `Ticket` objects with `Events` equal to those defined for the test.

HIBERNATE AND DATES

The event table has a `DATE` data type for when the event is taking place. Conveniently, this automatically maps to a `java.util.Date` object. For conciseness in the test in Listing 17-13, the `eventDate` object is created using the Joda Time library, and converted to a `java.util.Date` object. For more on Joda Time and other helpful Java libraries, see Chapter 18.

How do you use HQL?

HQL is Hibernate's query language. You have already seen an example in Listing 17-4; the line `session.createQuery("from Person").list()` uses the string `from Person` to pull all `Person` entities from the database.

The language is similar in some respects to SQL; you request data from somewhere, which may need to meet some criteria. Listing 17-14 shows some HQL to retrieve a single user.

LISTING 17-14: Being specific with HQL

```
@Test
public void hqlSinglePerson() {
    final Session session = sessionFactory.openSession();
    final List<Person> singleUser = session
        .createQuery("from Person where name like 'Dominic%'")
        .list();
    assertEquals(1, singleUser.size());
}
```

The HQL here, `"from Person where name like 'Dominic%'"`, limits the data to only people whose name starts with Dominic. For the data set for the test, only one person meets those criteria.

If you were more interested in `Tickets` that a particular `Person` had, you can query for those entities instead, as shown in Listing 17-15.

LISTING 17-15: Querying for fields on an entity

```
@Test
public void hqlSpecificTickets() {
    final Session session = sessionFactory.openSession();
    final List<Ticket> actual = session.createQuery(" " +
        "select person.tickets " +
        "from Person person " +
        "where person.name = 'Eric Twinge'").list();

    assertEquals(2, actual.size());
}
```

Because this query is retrieving a specific field from an entity, you need to start your query with `select`.

Listing 17-16 shows how to find the events a particular person was visiting.

LISTING 17-16: Extracting data from multiple levels of Hibernate objects

```

@Test
public void hqlVenuesForPerson() {
    final Session session = sessionFactory.openSession();
    final List actual = session.createQuery("" +
        "select event.eventName " +
        "from Person person " +
        "inner join person.tickets as tickets " +
        "inner join tickets.event as event " +
        "where person.name = 'Eric Twinge'").list();

    assertEquals(Arrays.asList("The Super Six", "David Smith"), actual);
}

```

This test retrieves *strings*, not a special domain object managed by Hibernate. The HQL query reads naturally for someone who has some experience with SQL.

What performance impact can Hibernate have?

By default, Hibernate entities are retrieved *eagerly*; that is, when you retrieve an object from a database, any dependent objects are also retrieved at the same time. In the case of the ticket office, retrieving a `Person` object will also retrieve its relevant `Ticket` objects and `Event` objects.

This can be quite an expensive, time-consuming operation. For a real ticketing system, which may hold thousands of events, each of which can have tens of thousands of tickets, retrieving all of these objects does not make much sense.

Thankfully, Hibernate enables you to *lazily* load objects; that is, the query will happen if and when it needs to. Listing 17-17 shows how this is done for the `Ticket` object. If you set tickets and events to be lazily loaded, then when you retrieve `Person` objects, their tickets and events will only be loaded from the database if you try to access the `Ticket` set reference or the `Event` reference.

To make an object lazily loaded, you need to add a `lazy="true"` attribute to the relevant property in the Hibernate mapping.

LISTING 17-17: Lazily loading Events

```

<hibernate-mapping package="com.wiley.acinginterview.chapter17">

    <class name="Ticket" table="TICKETS">
        <id name="id" column="ID"/>
        <many-to-one name="event"
            class="Event"
            column="event_id"
            cascade="all"
            not-null="true"
            lazy="true"/>
    </class>
</hibernate-mapping>

```



```

    </class>

</hibernate-mapping>

```

To prove this is actually happening, whenever you access the `Ticket` object of a person, you will see Hibernate log the query if you still have the `show_sql` property switched on.

However, it is possible to go one better in proving this. By keeping a count of when `Event` objects are created, you can determine that they are being created at the correct time. Listing 17-18 shows some *temporary* changes you could make to the `Event` class.

LISTING 17-18: Counting instantiations of a Hibernate class

```

public class Event {

    private int id;
    private int venueId;
    private String eventName;
    private Date eventDate;

    public static AtomicInteger constructedInstances =
                                                new AtomicInteger(0);

    public Event() {
        constructedInstances.getAndIncrement();
    }

    ... the rest of the class as before

```

If the `AtomicInteger` class is unfamiliar to you, it is covered in Chapter 11, but in a nutshell, it is a holder for an integer that can be accessed for reading and writing in a thread-safe way.

By adding a static counter, this can be referenced globally and checked at the appropriate time. It is worth emphasizing that this is not good design, or particularly safe code, but it is useful here to prove the point.

You only need to increment the counter in the no-arg constructor; because this is the only constructor Hibernate will use when constructing `Events`.

Listing 17-19 shows how to determine that the `Events` were lazily loaded.

LISTING 17-19: Determining lazy instantiation of Hibernate objects

```

@Test
public void lazyLoading() {
    final SessionFactory sessionFactory =
        new Configuration()
            .configure()
            .buildSessionFactory();

    final Session session = sessionFactory.openSession();

```

continues

LISTING 17-19 *(continued)*

```
Event.constructedInstances.set(0);
final Person person = (Person) session.createQuery("" +
    "select person " +
    "from Person person " +
    "where person.name = 'Eric Twinge'").uniqueResult();

assertEquals(0, Event.constructedInstances.get());

final Set<Ticket> tickets = person.getTickets();

for (Ticket ticket : tickets) {
    System.out.println("Ticket ID: " + ticket.getId());
    System.out.println("Event: " + ticket.getEvent());
}

assertEquals(2, Event.constructedInstances.get());
}
```

You should note several points about this test. First, the `constructedInstances` variable is reset to zero before calling Hibernate, even though it is set to zero when initialized in the class. This is because Hibernate would have re-created these objects as the test was starting, to ensure that the mappings were correct.

If the `System.out.println` was removed, and the retrieved `Person` object was never referenced, it is possible that the compiler would remove the retrieval completely, because the `Person` object would never have been used.

After the `Person` object has been retrieved, the counter should still be zero, even though Hibernate has collected the `Person` from the database, and, it seems, the `Ticket` set is accessible.

Once the `Tickets` have been retrieved and their `Event` object referenced, the `Events` are created. This is verified by the fact that the counter has been incremented by the `Event` constructor.

You should be able to see that the lines printed with the test are intertwined with the Hibernate logging to extract the `Event` details from the database.

If you remove the `lazy` attribute from the `Tickets.hbm.xml` mapping, the test will fail, as you would expect.

SUMMARY

Hibernate is an extremely useful tool to aid in persistence of your domain objects. You should not treat it as a silver bullet; any experienced Java developer must be able to understand and write SQL. If you ever need to debug or understand how the objects are being mapped from the database into rich objects, Hibernate can provide logging to show exactly what SQL it creates to interact with the database.

It is often said that you should treat Hibernate as a *framework* rather than a *library*. That is, you need to develop your application around the way Hibernate works, rather than mixing the Hibernate JAR into your application and expecting it to play nicely with your already-established database schema and mature application. You may find that if you stray from the patterns provided by Hibernate for access and retrieval, you may have some headaches in getting Hibernate to understand what you require.

That said, Hibernate is extremely valuable in getting projects moving quickly, removing any worry about database interaction and different vendor dialects. For the ticket office schema built in this chapter, Hibernate was extremely flexible in working with the provided schema. Early in the chapter, when only the `Person` class existed, Hibernate was content to not try to understand the relationship between `People` and `Tickets`, until it was made explicit. Notice, too, that for this application, no concept of a venue was ever created, even though the table existed, had relevant data, and even had a foreign key reference in the `event` table.

The next chapter covers some functionality missing from Java's standard libraries, and looks at some common useful libraries and their use.

18

Useful Libraries

This chapter examines some utilities from three commonly used Java libraries: Apache Commons, Guava, and Joda Time. All three libraries attempt to provide extra, useful functionality that the developers believed was missing from the Standard Java APIs, but was common enough to be written into a reusable library.

What kinds of problems these libraries and their functions alleviate can often be asked as interview questions, but it is also important for experienced developers to know what is implemented in common libraries so that they do not need to reinvent the wheel every time they come across the same pattern.

For any mature project, it is not uncommon for all three of these libraries to appear in some way, but many more libraries are available than just the three shown here.

REMOVING BOILERPLATE CODE WITH APACHE COMMONS

How do you properly escape Strings?

The Commons Lang library complements the functionality found in the `java.lang` package, including several libraries for manipulating Strings.

Many systems work over many platforms and interfaces. It is common to see a Java server that serves HTML pages and receives input from HTML forms, or you can use a Java application to read and write a comma-separated values file. The `StringEscapeUtils` class provides the facility to represent a String in that environment. Listing 18-1 shows an example of how you could use the `StringEscapeUtils` class to provide some HTML formatting.

LISTING 18-1: Escaping Strings with Apache Commons

```

@Test
public void escapeStrings() {
    final String exampleText = "Left & Right";
    final String escapedString =
        StringEscapeUtils.escapeHtml4(exampleText);
    assertEquals("Left & Right", escapedString);
}

```

The class can also escape Strings for JavaScript, XML, and Java itself. This can be useful if you ever need to write Java source code programmatically.

How do you write an `InputStream` to an `OutputStream`?

The Apache Commons IO library has several operations for streams that significantly reduce the amount of boilerplate code you need to write.

One common pattern is the need to connect an `InputStream` to an `OutputStream`. For example, you may want to write the contents of an incoming HTTP request straight to disk for a file upload form or something similar.

Listing 18-2 shows the `copy` method on the `IOUtils` class.

LISTING 18-2: Connecting an `InputStream` to an `OutputStream`

```

@Test
public void connectStreams() throws IOException {
    final String exampleText = "Text to be streamed";
    final InputStream inputStream =
        new ByteArrayInputStream(exampleText.getBytes());

    final OutputStream outputStream = new ByteArrayOutputStream();

    IOUtils.copy(inputStream, outputStream);

    final String streamContents = outputStream.toString();
    assertEquals(exampleText, streamContents);
    assertNotSame(exampleText, streamContents);
}

```

The `assertNotSame` method confirms that a brand new String was constructed, and that the `copy` method transferred each byte to the `OutputStream`.

Listing 18-2 used a `ByteArrayInputStream` and `ByteArrayOutputStream` to demonstrate this functionality, but both streams were declared against the interface. In practice, you will more than likely use different streams and rely on reading from or writing to a third party, such as the disk or the network interface.

The `copy` method on the `IOUtils` class is heavily overloaded to give you as much flexibility as possible in copying a source of bytes from one location to another. You can use it to connect `Readers` to `Writers`, to connect a `Reader` to an `OutputStream`, or to connect an `InputStream` to a `Writer`.

Be aware, too, that the `toString` method on an `OutputStream` will not necessarily give the contents of the stream, because these could have been written to disk, or the network, or some other irretrievable place. Some `OutputStream` implementations do not override `toString` at all.

`IOUtils` provides a way to consume an `InputStream` to convert it to a `String`, as shown in Listing 18-3.

LISTING 18-3: From an `InputStream` to a `String`

```
@Test
public void outputStreamToString() throws IOException {
    String exampleText = "An example String";
    final InputStream inputStream =
        new ByteArrayInputStream(exampleText.getBytes());

    final String consumedString = IOUtils.toString(inputStream);

    assertEquals(exampleText, consumedString);
    assertNotSame(exampleText, consumedString);
}
```

Again, the `toString` method is heavily overloaded, and can convert `Readers`, too.

How can you split an `OutputStream` into two streams?

Similar to the `tee` command in UNIX, Apache Commons provides a `TeeOutputStream`; its constructor takes two `OutputStream` instances, and writes to both. This can be useful for auditing output from an application, such as logging responses. Listing 18-4 shows an example.

LISTING 18-4: Splitting an `OutputStream`

```
@Test
public void outputStreamSplit() throws IOException {
    final String exampleText = "A string to be streamed";
    final InputStream inputStream = IOUtils.toInputStream(exampleText);

    final File tempFile = File.createTempFile("example", "txt");
    tempFile.deleteOnExit();
    final OutputStream stream1 = new FileOutputStream(tempFile);
    final OutputStream stream2 = new ByteArrayOutputStream();

    final OutputStream tee = new TeeOutputStream(stream1, stream2);

    IOUtils.copy(inputStream, tee);

    final FileInputStream fis = new FileInputStream(tempFile);
```

continues

LISTING 18-4 *(continued)*

```

    final String stream1Contents = IOUtils.toString(fis);
    final String stream2Contents = stream2.toString();

    assertEquals(exampleText, stream1Contents);
    assertEquals(exampleText, stream2Contents);
}

```

Listing 18-4 writes to only one location, but the `TeeOutputStream` takes care of writing the data to both streams.

This code uses the Apache Commons IO extensively. As well as the `TeeOutputStream` class under test, the test text is turned into an `InputStream` using `IOUtils.toInputStream`; the data is copied between streams using the previously seen `copy` method; and the `FileInputStream` is consumed with `IOUtils.toString`. A useful exercise would be to rewrite this test using only the standard Java libraries, and see just how much extra code you need to write.

DEVELOPING WITH GUAVA COLLECTIONS

Guava is a set of libraries originally developed by Google for use in its own Java projects. It concentrates on collections, I/O, and math, as well as some crossover with the Apache Commons libraries. It can be seen as an extension to the data structures included as part of the Java Collections API, which was covered in Chapter 5.

Are any types of collections missing from Java's standard library?

Guava adds a collection called a `Multiset`. It enables you to have a set that allows multiple, equal elements. Additional methods on the `Multiset` class enable you to count the number of a particular element, and to turn it into a regular set. Similar to the Collections library, several `Multiset` implementations exist, such as a `HashMultiset`, a `TreeMultiset`, and a `ConcurrentHashMultiset`.

Listing 18-5 shows `Multiset` in use.

LISTING 18-5: Using a Multiset

```

@Test
public void multiset() {
    final Multiset<String> strings = HashMultiset.create();

    strings.add("one");
    strings.add("two");
    strings.add("two");
    strings.add("three");
    strings.add("three");
}

```



```

        strings.add("three");

        assertEquals(6, strings.size());
        assertEquals(2, strings.count("two"));

        final Set<String> stringSet = strings.elementSet();

        assertEquals(3, stringSet.size());
    }

```

This keeps a running total of the count of each element. It can be helpful to think of a `Multiset` as a `Map` between that element and an integer to keep the count. Though it is definitely possible to implement the notion of a `Multiset` this way, the Guava implementation is more powerful because the `size` method returns the element count, whereas the `Map` implementation returns the number of unique elements. Additionally, an `Iterator` over the `Map` returns each element only once, regardless of how many are in the collection.

A `Multimap` is similar to a `Multiset`—it is possible to have a key appear more than once in the `Map`.

A common pattern when using a `Map` to store multiple values against a key is to store a mapping from a key to a `List`, or `Collection`, of values. You then often see the following pseudocode when attempting to add value against a particular key:

```

method put(key, value):
    if (map contains key):
        let coll = map(key)
        coll.add(value)
    else:
        let coll = new list
        coll.add(value)
        map.put(key, coll)

```

This is brittle, prone to errors, and you also need to make sure the code is thread-safe. Guava's `Multimap` takes care of this for you.

Listing 18-6 shows an example of its use.

LISTING 18-6: A `Multimap` address book

```

@Test
public void multimap() {
    final Multimap<String, String> mapping = HashMultimap.create();

    mapping.put("17 High Street", "Alice Smith");
    mapping.put("17 High Street", "Bob Smith");
    mapping.put("3 Hill Lane", "Simon Anderson");

    final Collection<String> smiths = mapping.get("17 High Street");
    assertEquals(2, smiths.size());
    assertTrue(smiths.contains("Alice Smith"));
    assertTrue(smiths.contains("Bob Smith"));

    assertEquals(1, mapping.get("3 Hill Lane").size());
}

```

The `get` method does not return an instance of the value type of the map; it returns a `Collection` of that type.

The `Multimap` differs from Java's `Map` interface in several ways. The `get` method never returns `null`—when getting a nonexistent key, an empty collection will be returned. Another difference is that the `size` method returns the number of entries, not the number of keys.

Similar to the `Multiset`, you can convert your `Multimap` into a Java `Map` using the `asMap` method. This returns `Map<Key, Collection<Value>>`.

Another type of map provided by Guava is the `BiMap` interface. This provides a two-way lookup: Keys can look up values, and values can look up keys. Implementing this using Java's `Map` is particularly tricky; you need to keep two maps, one for each direction, and ensure that a particular value is never duplicated, regardless of which key it is associated with.

Listing 18-7 shows an example of mapping stocks to their company name.

LISTING 18-7: Using a `BiMap`

```
@Test
public void bimap() {
    final BiMap<String, String> stockToCompany = HashBiMap.create();
    final BiMap<String, String> companyToStock =
        stockToCompany.inverse();

    stockToCompany.put("GOOG", "Google");
    stockToCompany.put("AAPL", "Apple");
    companyToStock.put("Facebook", "FB");

    assertEquals("Google", stockToCompany.get("GOOG"));
    assertEquals("AAPL", companyToStock.get("Apple"));
    assertEquals("Facebook", stockToCompany.get("FB"));
}
```

The `inverse` method is a reference to the original `BiMap`, so any additions in the originally created map are reflected in the inverted map, and vice versa.

How do you create an immutable collection?

The `Collections` utility class in the Java Collections API provides some utility methods for creating unmodifiable collections, as shown in Listing 18-8.

LISTING 18-8: Creating an unmodifiable collection

```
@Test
public void unmodifiableCollection() {
    final List<Integer> numbers = new ArrayList<>();
    numbers.add(1);
    numbers.add(2);
```

```

    numbers.add(3);

    final List<Integer> unmodifiableNumbers =
        Collections.unmodifiableList(numbers);

    try {
        unmodifiableNumbers.remove(0);
    } catch (UnsupportedOperationException e) {
        return; // test passed
    }

    fail();
}

```

This can be useful because any reference to `unmodifiableNumbers` simply cannot mutate the underlying list. If references to the original list still exist, the collection referenced in the unmodifiable collection can change, as shown in Listing 18-9.

LISTING 18-9: Changing the underlying values of an unmodifiable collection

```

@Test
public void breakUnmodifiableCollection() {
    final List<Integer> numbers = new ArrayList<>();
    numbers.add(1);
    numbers.add(2);
    numbers.add(3);

    final List<Integer> unmodifiableNumbers =
        Collections.unmodifiableList(numbers);

    assertEquals(Integer.valueOf(1), unmodifiableNumbers.get(0));

    numbers.remove(0);

    assertEquals(Integer.valueOf(2), unmodifiableNumbers.get(0));
}

```

Guava focuses on providing utilities to create unmodifiable collections that are also immutable. Listing 18-10 shows how this works.

LISTING 18-10: Creating immutable collections

```

@Test
public void immutableCollection() {
    final Set<Integer> numberSet = new HashSet<>();
    numberSet.add(10);
    numberSet.add(20);
    numberSet.add(30);

    final Set<Integer> immutableSet = ImmutableSet.copyOf(numberSet);

    numberSet.remove(10);
}

```

continues

LISTING 18-10 *(continued)*

```

        assertTrue(immutableSet.contains(10));

        try {
            immutableSet.remove(10);
        } catch (Exception e) {
            return; // test passed
        }

        fail();
    }
}

```

Of course, this is very different from the unmodifiable approach, because this will make a *full copy of the collection*, so if you intend to use this approach, you should be aware of any memory constraints that could become a concern.

The unmodifiable approach from the Java Collections API simply delegates to the underlying collection, but intercepts calls to any mutating methods and throws an exception instead. This means that any optimizations that are made for thread safety on the underlying collection are still part of the unmodifiable approach, even though an unmodifiable collection cannot be mutated, and is therefore thread-safe.

Because using Guava's `copyOf` returns a copied collection, it is optimized for immutability; it does not need to ensure thread safety when accessing the elements. They simply cannot change.

Is it possible to create an `Iterator` over several `Iterators`?

Making your own “iterator of iterators” can be especially difficult to do by hand. You need to keep a reference to your most recent iterator, and when calling `hasNext` or `next` you may need to move to the next iterator, or even beyond that, if the next iterator has no elements at all.

Being asked to make this implementation can be a common interview question, but it is also provided out of the box by Guava, as shown in Listing 18-11.

LISTING 18-11: Iterating over iterators

```

@Test
public void iterators() {
    final List<Integer> list1 = new ArrayList<>();
    list1.add(0);
    list1.add(1);
    list1.add(2);
    final List<Integer> list2 = Arrays.asList(3, 4);
    final List<Integer> list3 = new ArrayList<>();
    final List<Integer> list4 = Arrays.asList(5, 6, 7, 8, 9);

    final Iterable<Integer> iterable = Iterables.concat(
        list1, list2, list3, list4);
}

```

```

    final Iterator<Integer> iterator = iterable.iterator();

    for (int i = 0; i <= 9; i++) {
        assertEquals(Integer.valueOf(i), iterator.next());
    }

    assertFalse(iterator.hasNext());
}

```

The `Iterables.concat` method takes implementers of the `Iterable` interface.

Can you find the intersection of two Sets?

Surprisingly, many operations you would expect to see on the `Set` interface are not there: There is no clear way when dealing with `Sets` to perform a union, subtraction, or difference without modifying the provided `Set`. The `Collection` interface provides the `addAll` method to perform a union, `removeAll` for difference, and `retainAll` for intersection. All of these methods modify the collection on which you called the method.

An alternative is to use the `Sets` class from Guava. This is a class with several static methods for manipulating sets. Listing 18-12 shows how it is used.

LISTING 18-12: Set operations with Guava

```

@Test
public void setOperations() {
    final Set<Integer> set1 = new HashSet<>();
    set1.add(1);
    set1.add(2);
    set1.add(3);

    final Set<Integer> set2 = new HashSet<>();
    set2.add(3);
    set2.add(4);
    set2.add(5);

    final SetView<Integer> unionView = Sets.union(set1, set2);
    assertEquals(5, unionView.immutableCopy().size());

    final SetView<Integer> differenceView = Sets.difference(set1, set2);
    assertEquals(2, differenceView.immutableCopy().size());

    final SetView<Integer> intersectionView =
        Sets.intersection(set1, set2);
    set2.add(2);
    final Set<Integer> intersection = intersectionView.immutableCopy();
    assertTrue(intersection.contains(2));
    assertTrue(intersection.contains(3));
}

```

The union, difference, and intersection methods all return a `SetView` object, which is reliant on the underlying sets. `SetView` has a method that can return an immutable set of the elements that meet that `SetView`'s requirements. `SetView` is a view over the sets, so if the underlying sets change, the properties of `SetView` may change, too. This is what happened with the `intersection` call in Listing 18-12: after that intersection call, another element was added to the `set2` object, and that value was in `set1`, so it was to be returned in the intersection of the two sets.

Utility classes for the core Java Collections interfaces include `Maps`, `Sets`, and `Lists`, and utility classes for the Guava interfaces include `Multisets` and `Multimaps`.

USING JODA TIME

Joda Time is intended as a replacement for Java's `Date` and `Calendar` classes. It provides much more functionality, and is generally considered easier to use. Java 8 will provide a new and improved date and time API based on Joda Time.

How do you use the `DateTime` class?

The `Calendar` class has traditionally been used for working with dates in Java. It has a strange access pattern, and can be slightly uncomfortable for newcomers to use. Listing 18-13 shows a simple operation that subtracts a month from the current date using the `Calendar` class.

LISTING 18-13: Using Java's `Calendar` class

```
@Test
public void javaCalendar() {
    final Calendar now = Calendar.getInstance();
    final Calendar lastMonth = Calendar.getInstance();
    lastMonth.add(Calendar.MONTH, -1);

    assertTrue(lastMonth.before(now));
}
```

The `add` method takes two integer parameters, and unless you are familiar with the API, you may not realize that the first parameter is intended to be a constant, one of a many special constant integers on the `Calendar` class.

This is partially a hangover from before Java 5, before it was possible to have enumerated types.

The `DateTime` class replaces Java's `Calendar` class, and provides many clearer methods for retrieving and modifying a representation of time and date. Listing 18-14 shows how to use `DateTime`.

LISTING 18-14: Using the `DateTime` object

```
@Test
public void dateTime() {
```

```

final DateTime now = new DateTime();
final DateTime lastWeek = new DateTime().minusDays(7);

assertEquals(now.getDayOfWeek(), lastWeek.getDayOfWeek());
}

```

The `DateTime` class has many methods for getting the appropriate information about a date, such as the day of the month, the week of the year, and even to check for a leap year.

`DateTime` objects are immutable, so any “mutating” operation, such as `minusDays` in Listing 18-14, actually returns a new `DateTime` instance. Similar to `String`, if you do call any mutating methods remember to assign the result to a variable!

How can you work with Java’s `Date` object when using Joda Time?

When working with a third-party or legacy API, you may be constricted to using `Date` objects as method parameters or return types. Joda Time tries to make it as easy as possible to interact with Java’s `Date` and `Calendar` objects. The `DateTime` class can take both `Date` and `Calendar` instances as constructor parameters, and it has a `toDate` method that returns the `DateTime` representation as a `Date`. Listing 18-15 shows taking a `Calendar` instance, converting it to a `DateTime`, modifying the time, and converting it to a Java `Date`.

LISTING 18-15: Converting to and from a `DateTime` object

```

@Test
public void withDateAndCalendar() {
    final Calendar nowCal = Calendar.getInstance();
    final DateTime nowDateTime = new DateTime(nowCal);

    final DateTime tenSecondsFuture = nowDateTime.plusSeconds(10);

    nowCal.add(Calendar.SECOND, 10);
    assertEquals(tenSecondsFuture.toDate(), nowCal.getTime());
}

```

What is the difference between a duration and a period in Joda Time?

The representation of amounts of time is a feature that is not part of Java’s `Date` and `Calendar` classes, and it is the simplicity of working with these in Joda Time that makes the library so popular.

A `DateTime` instance represents an *instant*; that is, a measurement of time exact to the millisecond. Joda Time also provides APIs for working with an amount of time between two instants.

A *duration* is an amount of time in milliseconds. If you add a duration to a `DateTime`, you will get a new `DateTime` instance. Listing 18-16 shows how this works.

LISTING 18-16: Using a duration

```

@Test
public void duration() {
    final DateTime dateTime1 = new DateTime(2010, 1, 1, 0, 0, 0, 0);
    final DateTime dateTime2 = new DateTime(2010, 2, 1, 0, 0, 0, 0);

    final Duration duration = new Duration(dateTime1, dateTime2);

    final DateTime dateTime3 = new DateTime(2010, 9, 1, 0, 0, 0, 0);
    final DateTime dateTime4 = dateTime3.withDurationAdded(duration, 1);

    assertEquals(2, dateTime4.getDayOfMonth());
    assertEquals(10, dateTime4.getMonthOfYear());
}

```

The `Duration` instance is calculated as the difference between two `DateTime` instances; in this case midnight January 1, 2010, and midnight February 1, 2010. Internally, this is represented as a number of milliseconds, and in this case it will be equal to the number of milliseconds in 31 days.

This duration is then added, once, to another `DateTime`, representing midnight on September 1, 2010, and a new `DateTime` is returned. Because September has 30 days, the `DateTime` returned represents October 2, 2010.

With the irregularity of calendars—differing lengths of months and even years—you may still want to perform operations such as “the same day of the month, in two months’ time,” or “the same date next year.” You cannot simply add 60 or so days for two months and get the same date next year. You need to check if there is a February 29 between now and then, and add one if so.

The `Period` class removes all of these headaches for you. A *period* is a quantity of time; it is relative to the starting date, and can only be resolved once the start date is known. Listing 18-17 shows how this works.

LISTING 18-17: Using a period

```

@Test
public void period() {
    final DateTime dateTime1 = new DateTime(2011, 2, 1, 0, 0, 0, 0);
    final DateTime dateTime2 = new DateTime(2011, 3, 1, 0, 0, 0, 0);

    final Period period = new Period(dateTime1, dateTime2);

    final DateTime dateTime3 = new DateTime(2012, 2, 1, 0, 0, 0, 0);
    final DateTime dateTime4 = dateTime3.withPeriodAdded(period, 1);

    assertEquals(1, dateTime4.getDayOfMonth());
    assertEquals(3, dateTime4.getMonthOfYear());
}

```


The `Period` in Listing 18-17 was calculated as the interval between February 1, 2011 and March 1, 2011. The `Period` was then applied to February 1, 2012, giving a new `DateTime` for March 1, 2012. A `Period` inspects the difference in each field between two `DateTime` instances, that is, the difference in months, days, hours, and so on. Although the number of days between the first two dates was 28 days, it was represented as a period of “one month.” When this period is applied to a leap year, that extra day is still part of that “one month,” so March 1, 2012 was returned. Had this interval been a `Duration` rather than a `Period`, the `DateTime` instance returned would have been the last day of February.

How do you convert a specific date into a human-readable representation?

As part of the Java API, the `SimpleDateFormat` class is commonly used to construct custom, human-readable dates. You provide a template for the date, and when given a `Date` instance, it returns a `String` representation of that date. Listing 18-18 shows this in use.

LISTING 18-18: Using `SimpleDateFormat`

```
@Test
public void simpleDateFormat() {
    final Calendar cal = Calendar.getInstance();
    cal.set(Calendar.MONTH, Calendar.JULY);
    cal.set(Calendar.DAY_OF_MONTH, 17);

    final SimpleDateFormat formatter =
        new SimpleDateFormat("'The date is 'dd MMM'");

    assertEquals(
        "The date is 17 July",
        formatter.format(cal.getTime()));
}
```

The format string is hard to use, and can often lead to confusion: Lowercase `m` represents milliseconds and uppercase `M` represents months, for instance. Plaintext must be surrounded by single quotes. Any nontrivial case for using the `SimpleDateFormat` class is often accompanied by some time with the Java API documentation and some fiddling to get the format exactly right. Also, the `SimpleDateFormat` class is not thread-safe as the format `String` can change after the object has been created. `DateTimeFormatter` is immutable, which implies thread safety, as once it has been created it can never be changed.

The `DateTimeFormatter` can parse similar `Strings` that you would give to the `SimpleDateFormat` class, but Joda Time also provides a builder class, giving clearer meaning to each element of the string you are building. Listing 18-19 shows how to construct the same `String` as that in Listing 18-18.

LISTING 18-19: Formatting dates with Joda Time

```
@Test
public void jodaFormat() {
    final DateTime dateTime = new DateTime()
        .withMonthOfYear(7)
        .withDayOfMonth(17);

    final DateTimeFormatter formatter = new DateTimeFormatterBuilder()
        .appendLiteral("The date is ")
        .appendDayOfMonth(2)
        .appendLiteral(' ')
        .appendMonthOfYearText()
        .toFormatter();

    assertEquals("The date is 17 July", formatter.print(dateTime));
}
```

SUMMARY

Java is mature enough that it has a large number of robust, well-supported libraries, from full application frameworks such as Tomcat, Spring, and Hibernate, all the way through to lower-level libraries that make the day-to-day work of a developer a little easier.

If you are writing the same pattern in code repeatedly, you may find that someone has encapsulated it into a library. When a well-known, established library has the same functionality as whatever you have written, it is often better to use the library. The chances are it will have been well tested, both with unit tests and also out in production of many other applications.

Even better: If you find you're writing the same code over and over again, and there is no library for it, you could write one yourself. Make it open source, publish it on GitHub, and maintain it. Potential employers love to see interview candidates who have written and maintained open source code, and it also gives them a chance to examine how you write code outside of an interview environment.

The next chapter steps away from writing Java code and looks at how to build nontrivial Java projects using build tools such as Maven and Ant.

19

Developing with Build Tools

There is much more to creating a usable application than writing Java source code. Applications comprise many different artifacts, XML configuration, and environment-specific properties, and even images or embedded code for other languages.

This chapter looks at two applications that focus on pulling all your project's resources together into a coherent bundle. Maven and Ant are possibly the two most-used and best-understood build tools for Java projects.

Any non-trivial professional application will need something other than the Java compiler to create distributable applications—sometimes you can use a simple shell script—but the chances are that something will be Maven or Ant.

BUILDING APPLICATIONS WITH MAVEN

What is Maven?

Maven is an all-encompassing build tool. It is used to compile, test, and deploy Java projects.

It has been designed to favor convention over configuration. For the most part, builds for Java applications can be defined in a very similar manner, and if you define the layout of your source code in the same way, Maven knows where to find all the resources it needs and can perform many different build tasks.

Maven has built-in defaults for most settings, so whether your application will be deployed as a *Web Archive* (WAR) file or as a JAR file, Maven has defined tasks for how to build these properly.

A non-trivial Java application will have dependencies on many other libraries, such as unit-testing frameworks like JUnit or TestNG, or utility libraries such as Apache Commons or Google's Guava. Maven provides a facility for defining these dependencies, and you can download these JARs directly from a repository such as www.maven.org on the Internet, or perhaps from a repository set up local to your organization such as Artifactory or Nexus.

Maven's plug-in system enables you to add some more specific operations to your build. Similar to the dependency setup, these plug-ins are served remotely, and Maven has the capability to find these at build time and perform whatever operation the plug-in provides. An example of a common plug-in is `maven-jetty-plugin`, which runs the current build as a web application. This plug-in finds the appropriate `web.xml` file by assuming your project is following the convention that the file lives in the `src/main/webapp/WEB-INF` directory. Another such plug-in is the `maven-release-plugin`, which performs complex operations in conjunction with source control such as Subversion and Git to cut code for a release build and change the version number of the application, ready for the next development iteration.

A specific unit of work to run is called a *goal*. Specific goals are bound to the phases of the build life cycle, such as `compile`, `test`, and `install`. For example, when you run `mvn compile`, you are actually running the `compile` phase of the `compiler` plug-in. If your project has not changed the default behavior of the `compile` phase, you can achieve exactly the same effect by running `mvn compiler:compile`.

How do you structure your project for use in Maven?

Figure 19-1 displays the default directory structure for a Maven build.

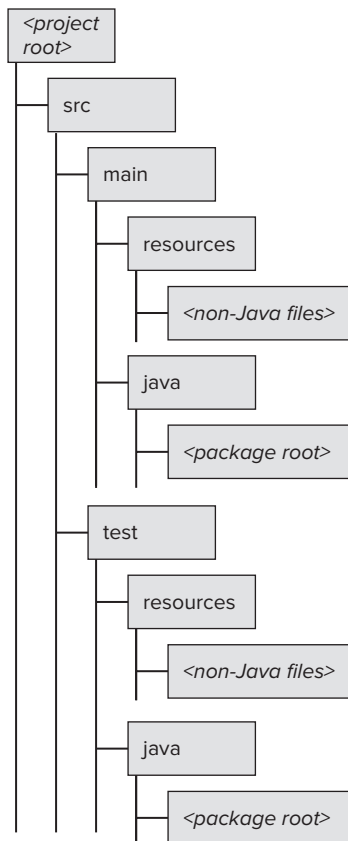


FIGURE 19-1

If you are new to Maven, what might surprise you here is that there are several *source roots* in the project. All of your production package structure lives under the `src/main/java` directory (if you have other languages, such as Scala, these would be housed in `src/main/scala`). Any classes living in this structure will be on your classpath at run time.

You also have the opportunity to add other files to the classpath; these live in the `src/main/resources` directory. Typically you would place any configuration here, such as your Spring application context configuration, as well as any properties files.

You can access any files in either of these directories at run time by calling `getClass()` `.getResource(name)`.

You also can add test code, such as unit tests and test-specific configuration. These are held in `test/main/java` and `test/main/resources`, respectively. When Maven runs any tests as part of its build, the classes and files in these locations are also added to the classpath. These are kept separate so that they are not bundled with any built artifact for production deployment.

What is the life cycle of a Maven build?

A Maven build consists of a chain of *phases*, called the *life cycle*. Each phase is bound to a specific goal. Each phase is dependent on the ones before—should a specific goal fail for whatever reason, the whole build fails. Figure 19-2 shows the basic phases for a build.

Any built artifacts are put in a directory called `target`, at the project root. The `clean` goal removes any files from a previous build by deleting this `target` directory. This is not run before any subsequent phases, unless you specifically initiate this build phase with `mvn clean` or define it to run always in your Project Object Model (POM).

The `validate` phase checks that the `pom.xml` file is a correct build file, checking that it conforms to the XML specification for Maven's build files. If your `pom.xml` is not even correct XML—for instance, if you are missing any closing tags, or your tags are not properly nested—then your build will fail before the Maven application is even properly initialized, displaying an error about a non-parseable POM or something similar.

The `compile` phase pulls in any defined dependencies and performs compilation of your code, building any class files into the `target/classes` directory.

The next phase is `test`. This compiles the classes under the test directory and runs any tests code here, such as unit or integration tests. By default, any failing unit tests will cause the build to fail.

The `package` phase is run after a successful test phase. This creates an artifact, such as a WAR or JAR file. This file is stored in the root of your `target` directory. Note that this is not a standalone file; by default it does not have any dependent libraries built in, so the dependencies still need to be included on the classpath. The goal that is bound to this phase is dependent on the packaging tag: If it is set to `jar`, the `jar:jar` goal is run; if it is set to `war`, the `war:war` goal is run.

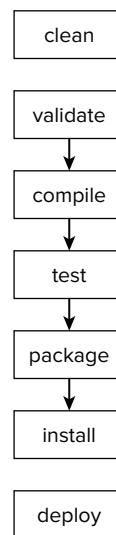


FIGURE 19-2

The `install` step publishes the built artifact to your local maven repository. This is usually held in the directory `$HOME/.m2/repository`. If you have more than one Maven build running locally, your new artifact is now available to those builds.

The final step is `deploy`. This step usually needs a small amount of extra configuration, defining exactly where to deploy the finished artifact. This deployment is usually to an artifact repository such as Artifactory or Nexus.

These repositories are typically where you would download dependent artifacts for your own build, but these repositories often act as a holding place for installing builds onto full test environments or a live environment.

The life-cycle phases here are the most basic steps; several more, such as `generate-test-sources`, `pre-integration-test`, and `post-integration-test`, can be used to provide more granularity to a build.

How is a specific application build defined in Maven?

The definition of a Maven build is held in a file called `pom.xml`, which lives in the root directory of the project. Maven uses this file as the instructions for how to build your application. Listing 19-1 shows a very simple example.

LISTING 19-1: A simple Maven build file

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.wiley</groupId>
  <artifactId>acing-java-interview-example-pom</artifactId>
  <version>1.0</version>
  <packaging>jar</packaging>

  <dependencies>
    <dependency>
      <groupId>commons-io</groupId>
      <artifactId>commons-io</artifactId>
      <version>1.4</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>3.0.0.RELEASE</version>
    </dependency>

    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.8.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

```

        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <version>1.9.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.7</source>
                <target>1.7</target>
                <showDeprecation>true</showDeprecation>
                <showWarnings>true</showWarnings>
                <fork>true</fork>
            </configuration>
        </plugin>
    </plugins>
</build>
</project>

```

The POM can be broadly split into three sections: the project definition, the dependencies for a build, and any build-specific plug-ins.

An exact artifact can be specified by the `groupId`, `artifactId`, and `version`. These are called *coordinates*, or often casually referred to as GAV. If your project is a library for use in other projects, those other projects will use these attributes to define their dependency on this build. The `build` tag here informs Maven what type of artifact to build in the `package` life-cycle step. Valid steps include `war` and `jar`, or even `maven-plugin` if you write your own extension to Maven. The `groupId` usually refers to your own organization, and the `artifactId` is your application name.

The `dependencies` tag is where you would define any dependent libraries for your build. This can often be other libraries built by your own organization, perhaps available on an intranet repository, or a third-party library. The exact `groupId`, `artifactId`, and `version` are usually found in the project's documentation, usually on the project website. Alternatively, a Google search for popular libraries or Maven repository search engines such as `mvnrepository.com` can be useful, though the usual caveats should be clear for downloading any unknown executable code from the Internet.

Dependent artifacts can have different scopes. These define how they are used within the project, and when they are pulled into the build. Listing 19-1 had the scopes for `junit` and `mockito-core` as `test`. This means they are available while building the test source, which is by default any code under the `src/test` root, and also for running the tests. The `test` scoped dependencies are not available to the main source, by default under `src/main`.

That is, `test` scoped artifacts are specific for testing, and are not necessary for the main code of the application.

Not providing a specific scope defaults to `compile`. For this scope, an artifact is resolved for the compilation step, and is also provided to the classpath as a runtime dependency.

If a library is needed at compile time, but not at run time, this has the scope `provided`. The Servlet API JAR is a common example of this: At run time, the JAR is not provided by Maven but by the application container, but you still need the class definitions to compile your source in the first place.

If you have a specific library that is not available through a Maven repository, you can provide local filesystem libraries with the `system` scope. You should use this only as a last resort; the use of Maven repositories means that your build should be reproducible across any number of machines—yours, your colleague’s, your build server, and so on. Having local-only libraries can lead to a confusing dependency nightmare later on in your build cycle.

The `build` tag of the POM defines any build plug-ins, and any specific configuration for those plug-ins. Listing 19-1 defined some specific configuration for the `maven-compiler-plugin`, used in the compile stage, defining that the compiler should expect Java 7 source, and can output Java 7-specific class files.

As with the `maven-compiler-plugin`, any of the build defaults can be altered using this build section to define how a particular step should behave. One common pattern is to separate the tests: The unit tests are run as part of the `test` step, and the expensive, slow integration tests are run later after the package step, during the `integration-test` step. Some projects use a naming convention to differentiate unit and integration tests: Any class ending in `IntegrationTest.java` will be filtered out of the `test` step and run in the `integration-test` step instead. Listing 19-2 shows how you can do this.

LISTING 19-2: Separating unit and integration tests

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <excludes>
      <exclude>/**/*.IntegrationTest.java</exclude>
    </excludes>
  </configuration>
  <executions>
    <execution>
      <id>integration-test</id>
      <goals>
        <goal>test</goal>
      </goals>
      <phase>integration-test</phase>
      <configuration>
        <excludes>
          <exclude>none</exclude>
        </excludes>
        <includes>
          <include>/**/*.IntegrationTest.java</include>
        </includes>
      </configuration>
    </execution>
  </executions>
</plugin>
```


The `maven-surefire-plugin` is the default plug-in for running unit tests; this is providing specific configuration to the plug-in, instructing it what to run and when.

If you do not need any specific configuration when using a particular plug-in, you do not need to define the plug-in at all. For example, when you run `mvn jetty:run` using the `maven-jetty-plugin`, this plug-in will look in `src/main/webapp/WEB-INF` for the `web.xml` file, and start a web application with that configuration. Of course, if your application holds the `web.xml` file somewhere else, you would need to provide the appropriate configuration to allow the plug-in to find it.

How can you share built artifacts between applications?

If you were to create a reusable component, you may want to reuse that component between projects. One such example would be to provide a high-level API to connect to a specific service within your organization, such as an API for connecting to a high-score service for several games, or providing real-time trading prices across several applications.

If you have a Maven repository configured for your organization, you need to configure your POM to check this repository for when it resolves dependencies, and also for publishing your successful builds. Listing 19-3 shows a snippet of a `pom.xml` file configured for a custom Maven repository. This is a top-level tag within the `<project>` tag.

LISTING 19-3: Configuring a repository for deployment

```
<distributionManagement>
  <repository>
    <id>deployment</id>
    <name>Internal Releases</name>
    <url>
      http://intranet.repo.com/content/repositories/releases/
    </url>
  </repository>
  <snapshotRepository>
    <id>deployment</id>
    <name>Internal Snapshots</name>
    <url>
      http://intranet.repo.com/content/repositories/snapshots/
    </url>
  </snapshotRepository>
</distributionManagement>
```

Note that two endpoints are configured: one for releases and one for snapshots. The differentiation between the two is important. For active development, the `version` tag will have a `-SNAPSHOT` suffix. This indicates that the artifact is still under development and may not be stable. When the `deploy` goal is run, if the build is a *snapshot build*, this will be deployed to the `snapshotRepository` URL.

For more stable builds, which are feature complete and pass your application's full testing criteria, such as User Acceptance Testing or exploratory QA testing, the version will have the `-SNAPSHOT` suffix removed, so when the `deploy` goal is run, this will be deployed to the URL inside the `repository` tag.

Some continuous delivery systems are set up to automatically release any new artifacts deployed to this URL. The `releases` URL usually defines that an artifact is ready for release.

Of course, you need to make sure your organization's local Maven repository has the correct permissions configured so you can deploy artifacts to these URLs.

How can you bind different goals to a specific Maven phase?

Although the default build model may be enough for your application, for a significantly advanced project, you will probably find that you want to do something different than the usual.

For example, in the `package` phase when the `packaging` tag is set to `jar`, this, by default, creates a JAR containing only the code from your project and no dependencies. If you had a class with a `main` method inside this JAR, you would need to create a command-line application manually, with reference to all dependent library JARs to run your application.

Luckily, this problem is so common that a plug-in has been made for this situation. The `maven-assembly-plugin` has the capability to create a JAR with dependencies. The resultant artifact will be much larger because it contains all the class files from your dependent JARs, but will be runnable as a single file. This is especially useful as a solution for technical interviews or tests, where you may be asked to write some Java code to solve a particular problem and submit a runnable application as well as source code. This Maven plug-in is a particularly neat and professional way to submit any runnable application to a potential employer; they will not have to waste any time getting your application to run because all the dependencies are built in. Listing 19-4 shows some sample configuration for using this plug-in.

LISTING 19-4: Changing the behavior of a specific phase

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.4</version>
  <configuration>
    <finalName>standalone-application</finalName>
    <descriptorRefs>
      <descriptorRef>jar-with-dependencies</descriptorRef>
    </descriptorRefs>
    <archive>
      <manifest>
        <mainClass>
          com.wiley.javainterviewsexposed.chapter19.MainClass
        </mainClass>
      </manifest>
    </archive>
  </configuration>
  <executions>
    <execution>
      <id>make-assembly</id>
      <phase>package</phase>
```

```

        <goals>
          <goal>single</goal>
        </goals>
      </execution>
    </executions>
  </plugin>

```

The plug-in has some configuration that is specific to making it work for your needs. The `finalName` tag defines exactly how your final JAR will be named.

The `executions` section defines what exactly to run and when to run it. The `<goal>single</goal>` tag defines that the `single` goal defined in the `maven-assembly-plugin` is to be run. The `single` goal creates a JAR that can be run as a command-line application. This is bound to the `package` phase. This could easily have been a different phase, perhaps `post-integration-test`, when you know that your packaged artifact has passed all of your integration tests.

With this plug-in defined in your `pom.xml`, if you run `mvn package`, you will find two JARs in your target directory: the original JAR as part of the default Maven build, and a second, called `standalone-application-with-dependencies.jar`. It should be possible to run the class defined as `mainClass` with this command:

```
java -jar standalone-application-with-dependencies.jar
```

ANT

How do you set up a build with Ant?

Ant does not follow the same model of convention over configuration like Maven does. As such, you need to explicitly specify every step of your build. There is no default directory structure. Although this may appear to be more work, and often it is, you have ultimate flexibility for how your build operates and what exactly needs to happen for your application to be built successfully.

Ant works with two main concepts, *targets* and *tasks*. A target is quite similar to Maven's idea of a goal: It is a high-level action that your build performs, such as compiling code, running tests, or packaging your application into a JAR.

Tasks are the actual actions that comprise a target. For instance, there is a `javac` task to compile your code, a `java` task to run an application, and even a `zip` task to bundle a selection of files into a zip file. You can even create your own tasks.

When you run an Ant build from the command line, the tool automatically looks for a `build.xml` file. You do have the opportunity to specify a different name if you want. Listing 19-5 shows an example `build.xml` file, which compiles the project and then bundles the files into a JAR.

LISTING 19-5: An example Ant build file

```

<project name="ExampleAntBuildFile" default="package-jar" basedir=".">

  <target name="init">
    <mkdir dir="ant-output"/>
    <mkdir dir="ant-test-output"/>
  </target>

  <target name="compile" depends="init">
    <javac srcdir="src/main/java" destdir="ant-output"/>
    <javac srcdir="src/test/java" destdir="ant-test-output"/>
  </target>

  <target name="package-jar" depends="compile">
    <jar jarfile="Example.jar" basedir="ant-output"/>
  </target>

</project>

```

This build has three targets, which correspond to the high-level tasks required for this application. You can run this build by simply typing `ant` on the command line. This will attempt to run the build for the default target (the target `package-jar`) to build the JAR.

Notice that each task can specify a dependency. If one or more dependencies are specified, Ant will execute those targets before continuing. If a target fails, such as on a compilation error or a unit test failure, the build will stop.

A target can constitute many different tasks. The `compile` task contains two executions of the `javac` task—one for the production code and one for test code. Each compilation is given a separate output because only the production classes are needed for the `package-jar` target.

When you run an Ant build, you can explicitly specify the target you want to run. The build will stop when this target and any targets it depends on are finished. Running `ant compile` stops after creating the output directories and compiling the source files.

Is it possible to integrate Ant and Maven together?

Within a Maven build is a plug-in called the `maven-antrun-plugin`, which enables you to run Ant tasks as part of a Maven build. One common use for this is to run an operating system command or shell script using Ant's `exec` target. There is no equivalent within the Maven world to do this. Listing 19-6 is an example of how this works.

LISTING 19-6: Using Ant tasks within Maven

```

<plugin>
  <artifactId>maven-antrun-plugin</artifactId>
  <version>1.7</version>
  <executions>

```

```

<execution>
  <phase>deploy</phase>
  <configuration>
    <tasks>
      <exec dir="scripts"
        executable="scripts/notify-chat-application.sh"
        failonerror="false">
      </exec>
    </tasks>
  </configuration>
  <goals>
    <goal>run</goal>
  </goals>
</execution>
</executions>
</plugin>

```

This has been attached to Maven's `deploy` phase, and you can imagine that this script could send an alert to the team's chat server that a deployment is about to happen.

SUMMARY

Java has several mature and stable products for helping you turn your source code into a useful, executable binary artifact. For anything more than a simple program, using a build tool such as Maven or Ant will help you produce professional, standard builds in a relatively short amount of time.

Maven is a fantastic way to get a project running very quickly. By following convention, you need very little input to get a build working. It also has the added benefit that any new colleagues or team members who are already experienced with Maven can get up to speed with your project very quickly, because the build will work in the same way as other projects.

Ant has the ultimate flexibility. For projects that do not or cannot follow convention, Maven can often be problematic to get working. Although an Ant build may need more supervision, your builds will often be faster and much better understood, because the build will only be performing tasks that are relevant to your application.

Whichever build technology you decide to use, making the decision to actually use a build tool is the best decision of all. Many projects are often blighted with bespoke operating system scripts, which maybe only one or very few team members understand properly. Both Maven and Ant are well supported within the Java community.

The final chapter on the book focuses on the use of Java in writing mobile phone applications for the Android operating system.

20

Android

Increasingly, more and more mobile devices are being sold as sales of laptops and desktop computers have declined. While mobile websites perform reasonably well, native mobile apps still may yield superior user experience. Additionally, with the decline of J2ME, Android has become the preferred way to run Java code on mobile devices. In short, Android development skills are important for any developer to be familiar with because there may be no desktop to run them on someday, and right now a good portion of the users for any product may be mobile users.

Fortunately, Android programming skills are a specialization of Java skills. Hence, a Java developer can transition smoothly to an Android developer. Android apps are written in the same programming language as Java programs, but they run under Android OS and make heavy use of the Android SDK in addition to a standard Java SDK. Another difference is that Java programs can run on many different platforms, but they most likely run on powerful desktops or servers. In contrast, devices that run Android apps have limited computational resources, but a rich set of sensing capabilities. They are also highly mobile and keep a near constant Internet connection. Thus, Android apps are like Java programs, but they take full advantage of the device's capabilities while running within the constraints of the Android OS.

This chapter reviews major concepts from the Android SDK and shows the code that apps need to accomplish common tasks. It does not cover all of the Android SDK, but instead aims to focus on the most commonly used parts. Specifically, this chapter covers the following:

- The components of an Android app, how they communicate, and what they are for
- Using layouts to create various user interfaces and organizing them with `Fragments`
- Storing different kinds of data
- Using Location Services as an example of accessing hardware on Android devices

NOTE *This source code for this book includes an Android App that executes some of the example code in this chapter.*

BASICS

Android apps are made up of several types of objects and communicate using `Intents`. As an Android developer, you need to be familiar with each of these components and what they are used for.

Components

What are the main components of Android apps and what are their main functions?

The four main components in Android are `Activity`, `Service`, `BroadcastReceiver`, and `ContentProvider`. The first three communicate with each other using `Intents`, and apps use `ContentResolvers` to communicate with `ContentProviders`. Each component has different uses, and all follow their own life cycle controlled by Android.

What is an `Activity`?

An `Activity` represents a screen with a user interface. Each `Activity` runs independently, and internal and external apps can start them directly to control how an app changes screens. For example, an app might start up an `Activity` to display a list of recipes, and then another `Activity` to show the recipe details. A third search `Activity` might also activate the same recipe detail `Activity` when the user selects a search result.

What is a `BroadcastReceiver`?

A `BroadcastReceiver` receives system-wide or internally broadcasted `Intents`. `BroadcastReceivers` might show a notification, activate another component, or perform other short work. An app might use a `BroadcastReceiver` to subscribe to many system broadcasts, such as acting on startup or when the network connection changes.

What is a `Service`?

A `Service` represents long-running or persistent code that runs in the background. For example, an app might run a `Service` that executes a recipe backup task, so that the user can continue to do other things within the app and even switch apps without interrupting the backup. Another `Service` might run continuously, recording the user's location every few moments for later analysis.

What is a `ContentProvider`?

A `ContentProvider` stores shared data between components and makes it available through an API. Apps use it to share data publicly and also sometimes to share data privately within an app. One example is the `ContentProvider` Android exposes to give access to a user's contacts. `ContentProviders` have the advantage of abstracting away the details of the storage mechanism used from the method of reading and writing the data. They also encapsulate data access. You can change the internal method of storage without changing the external interface.

Intents

How do Android components communicate?

Android components can send `Intents` to activate each other and pass data between them. `Intents` specify what components Android should deliver them to and contain any necessary data. If an `Intent` specifies a class name it is called *explicit* because Android directs it to a particular class and if an `Intent` specifies an action string it is called *implicit* because Android directs them to any class that knows the action string.

How does an Activity send an Intent to another Activity?

To send an `Intent` to a particular `Activity`, an app first constructs an explicit `Intent` by specifying a class name. For example, in Listing 20-1 the code specifies that the `RecipeDetail` class should receive the `Intent` just created. Second, an app might add some data, such as a `recipeId`. Finally, the app calls `startActivity()` to send the `Intent`.

LISTING 20-1: Code to send an Intent containing String data

```
private void sendRecipeViewIntent(final String recipeId) {
    final Intent i = new Intent(this, RecipeDetail.class);
    i.putExtra(RecipeDetail.INTENT_RECIPE_ID, recipeId);
    startActivity(i);
}
```

To receive an `Intent`, an `Activity` can call its `getIntent()` method. The code in Listing 20-2 shows how `RecipeDetail` receives the recipe detail `Intent` that was sent from Listing 20-1 and extracts the `recipeId` from it.

LISTING 20-2: Receives an Intent and extracts a recipeId from it

```
public class RecipeDetail extends Activity {
    public static final String INTENT_RECIPE_ID = "INTENT_RECIPE_ID";

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

continues

LISTING 20-2 *(continued)*

```

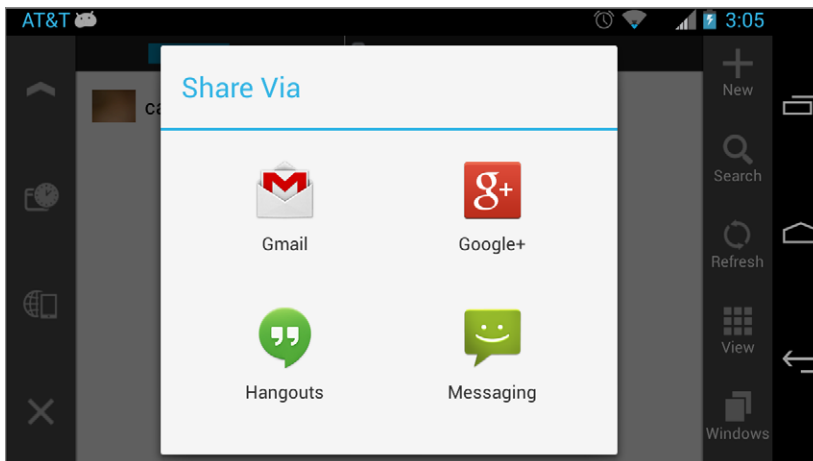
setContentView(R.layout.recipe_detail);
if (getIntent() != null) {
    if (getIntent().hasExtra(INTENT_RECIPE_ID)) {
        final String recipeId = getIntent().getStringExtra(
            INTENT_RECIPE_ID);
        // access database, populate display, etc...
        loadRecipe(recipeId);
    }
}
}
}

```

How does an app send common Intents?

One of the advantages of Android is that apps can easily communicate with each other using the Intent mechanism. Apps are free to define new Intents and allow any app to respond to them. This enables a high level of interaction between apps.

In particular, apps can use several common Intents provided by the Android OS. One such Intent is the share Intent. The code in Listing 20-3 creates and sends some text via the share Intent. When sent, the user sees a list like the one in Figure 20-1. As the figure shows, many apps might handle the Intent, and it is up to the user to decide which app will receive the Intent and do something with it. For example, users might elect to e-mail their text or send it via text message by selecting the e-mail or messaging icons.

**FIGURE 20-1**

LISTING 20-3: Sends a share Intent

```
private void sendShareIntent() {
    final Intent sendIntent = new Intent();
    sendIntent.setAction(Intent.ACTION_SEND);
    sendIntent.putExtra(Intent.EXTRA_TEXT, "Share Me!");
    sendIntent.setType("text/plain");
    startActivity(sendIntent);
}
```

How does an app allow an Activity to respond to an Intent?

When an app does not know the receiver of an Intent, it cannot use an explicit Intent to direct it. Thus, instead of naming a specific class as the code in Listing 20-1 does, apps specify an action string to send an implicit Intent. This more general approach allows any app that knows the action string to register to receive Intents that contain it. Action strings, such as `Intent.ACTION_SEND`, exist for common Intents, but apps can also create their own.

To receive such Intents, Activities need to register to receive them. They do so by adding an `IntentFilter` in the `AndroidManifest.xml` file as shown in Listing 20-4. The filter specifies that Android should send any Intents with the `android.intent.action.SEND` to `ShareIntentActivityReceiver`, shown in Listing 20-5. Apps can specify additional criteria within the `intent-filter` tag to further specify which Intents to receive.

LISTING 20-4: Manifest entry for making Android send any android.intent.action.SEND Intents to ShareIntentActivityReceiver

```
<activity android:name=".share.ShareIntentActivityReceiver"
    android:label="@string/share_intent_receiver_label">
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="text/*" />
    </intent-filter>
</activity>
```

LISTING 20-5: Receives the android.intent.action.SEND Intent

```
public class ShareIntentActivityReceiver extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        final String sharedText =
            getIntent().getStringExtra(Intent.EXTRA_TEXT);
    }
}
```

Activities

What is the Activity's life cycle?

Proper handling of the Activity life cycle is essential for creating an app that behaves as users expect. Respecting the Activity life cycle means that users can restart easily from where they left off within an app, and that any resources used by the app are appropriately released when the user has gone away to some other task. This makes apps easier to use and also minimizes battery usage. Any Android developer should be familiar with the documentation and flow diagram on this site: <http://developer.android.com/reference/android/app/Activity.html>. The documentation highlights several lifetimes between life cycle stages:

- **Entire lifetime**—From `onCreate()` to `onDestroy()`
- **Foreground lifetime**—From `onResume()` to `onPause()`
- **Visible lifetime**—From `onStart()` to `onStop()`

Additionally, when an app needs to save state, it calls `onSaveInstanceState()`; and when the app needs to restore state, it calls `onCreate()` with a `Bundle` containing the saved state. For example, if the user gets a phone call or changes screen orientation while using an app, Android calls `onSaveInstanceState()` and then calls `onCreate()` again after the phone call is over or the orientation change is complete. If the app has any state to preserve it can save it in the `onSaveInstanceState()` method. Following are several state-related questions to explore.

How does an app preserve dynamic state when the app temporarily is removed from view?

Dynamic state is a set of temporary values that are relevant while the app is running. Any dynamic state should be saved when the user switches Activities; otherwise, when the user returns to the Activity it may appear different when the UI is reset.

A good example of state that an app might need to save is the scroll position within a `ListView`. If users scroll down half the list, drill down to view one of the items, then come back to the list, they will expect the app to keep the list in the same position. To implement this, an app needs to save the scroll position before drilling down to the list detail. Listing 20-6 shows how the app saves the scroll position during `onSaveInstanceState()` and then restores it during `onCreate()`.

LISTING 20-6: Preserves the list scroll position and keeps a `BroadcastReceiver` registered only when the Activity is active

```
public class RecipeList extends ListActivity {
    private BroadcastReceiver receiver;

    public static final String BUNDLE_SCROLL_POSITION = "BUNDLE_SCROLL_POSITION";

    @Override
```

```

protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    final List<String> recipes = getRecipes();

    // make a string adapter, then add an onclick
    final ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
        android.R.layout.simple_list_item_1, recipes);
    setListAdapter(adapter);
    getListView().setOnItemClickListener(
        new AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> parent, View view,
                int position, long id) {
                final String recipeClicked = recipes.get(position);
                sendRecipeViewIntent(recipeClicked);
            }
        });

    if (savedInstanceState != null
        && savedInstanceState.containsKey(BUNDLE_SCROLL_POSITION)) {
        final int oldScrollPosition = savedInstanceState
            .getInt(BUNDLE_SCROLL_POSITION);
        scrollTo(oldScrollPosition);
    }
}

private List<String> getRecipes() {
    // get from a db, but for now just hard code
    final List<String> recipes = Arrays.asList("Pizza", "Tasty Minestrone",
        "Broccoli", "Kale Chips");
    return recipes;
}

private void sendRecipeViewIntent(final String recipeId) {
    final Intent i = new Intent(this, RecipeDetail.class);
    i.putExtra(RecipeDetail.INTENT_RECIPE_ID, recipeId);
    startActivity(i);
}

@Override
protected void onSaveInstanceState(Bundle outState) {
    // which row in the recipe is the user looking at?
    // save this so the app can rescroll to that location
    // when the user returns
    final int scrollPosition = getScrollPosition();
    outState.putInt(BUNDLE_SCROLL_POSITION, scrollPosition);
    super.onSaveInstanceState(outState);
}

private int getScrollPosition() {
    final int scrollPosition = getListView().getFirstVisiblePosition();
    return scrollPosition;
}

private void scrollTo(final int scroll) {

```

continues

LISTING 20-6 *(continued)*

```

        // scroll the list view to the given row
        getListView().setSelection(scroll);
    }

    @Override
    protected void onResume() {
        register();
        super.onResume();
    }

    @Override
    protected void onPause() {
        unregister();
        super.onPause();
    }

    private void register() {
        if (receiver == null) {
            receiver = new NewRecipeReceiver();
            final IntentFilter filter = new IntentFilter(
                "com.wiley.acinginterview.NewRecipeAction");
            this.registerReceiver(receiver, filter);
        }
    }

    private void unregister() {
        if (receiver != null) {
            this.unregisterReceiver(receiver);
            receiver = null;
        }
    }

    private class NewRecipeReceiver extends BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent) {
            // add to the recipe list and refresh
        }
    }
}

```

When should an app release resources or connections?

Apps typically release resources in the `onPause()` method and restore them during `onResume()`. For example, apps might start and stop `BroadcastReceivers` or start and stop listening for sensor data. Listing 20-6 shows how an app registers and then unregisters a `BroadcastReceiver` during `onPause()` and `onResume()`. Doing so ensures that the app will not get notified unnecessarily, and saves battery power by not updating the display of a user interface that is not visible.

When should an app persist data when the user is changing Activities?

Apps should store any persistent state, such as the edits made to a recipe, during `onPause()` and restore them `onResume()`. Apps should not use `onSaveInstanceState()` for this purpose because Android does not guarantee that it will call `onSaveInstanceState()` while shutting down an Activity.

BroadcastReceivers

How does an app use a `BroadcastReceiver` to receive a system-generated event?

A typical use of a `BroadcastReceiver` is to receive events generated by system functions, such as a network connection change. Implementation requires an `AndroidManifest` entry and a class to implement the `BroadcastReceiver`.

Listing 20-7 specifies an `AndroidManifest` entry that makes Android call `NetworkBroadcastReceiver` every time Android sends an `android.net.conn.CONNECTIVITY_CHANGE`. Apps can also register dynamically using code such as that shown in Listing 20-6.

LISTING 20-7: `AndroidManifest` parts needed to receive the connectivity change broadcast

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<receiver android:name=".receiver.NetworkBroadcastReceiver" >
    <intent-filter>
        <action android:name="android.net.conn.CONNECTIVITY_CHANGE"/>
    </intent-filter>
</receiver>
```

The code in Listing 20-8 implements a `BroadcastReceiver` and receives the `android.net.conn.CONNECTIVITY_CHANGE` broadcast. Because the app registered previously; Android calls `onReceive()` every time a network connection change occurs. An app might want to subscribe to many other events as well, such as the `BOOT_COMPLETED` event.

LISTING 20-8: Receives a network connection change broadcast

```
public class NetworkBroadcastReceiver extends BroadcastReceiver {
    private static final String TAG = "NetworkBroadcastReceiver";

    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d(TAG, "network connection change: " + intent.getAction());
    }
}
```

Services

How does an app execute code in the background?

Some processing could take a while. Also, by default, all processing occurs in the UI thread. Executing lengthy operations freezes the display and can result in an ANR (Application Not Responding) error, where the user forces the app to quit.

Because of these two reasons, an app might want to run the processing as a `Service` so that it executes persistently on its own thread. Doing so allows the user to run other apps instead of waiting for the processing to complete and prevents ANRs.

For example, a user might want to trigger a backup manually. If the user has a lot of data, he should not have to wait several minutes for the backup to finish. Using a `Service` enables the user to start the backup, and then do other things with the device while the backup is occurring.

Recipe backup is an example of one-shot processing where the `Service` does some work, and then shuts down. Because this is a common pattern, Android has a special `Service` called an `IntentService`. It handles starting a single worker thread to process each `Intent` it receives. It also handles shutting down the `Service` when it completes.

Listing 20-9 shows how to implement the `IntentService` and Listing 20-10 shows the required entry in the `AndroidManifest.xml` file. To start an `IntentService`, apps pass an `Intent` to `Context.startService()`. Android then passes the `Intent` to the `IntentService`.

LISTING 20-9: Backs up recipes using an `IntentService`

```
public class RecipeBackupService extends IntentService {
    private static final String TAG = "RecipeBackupService";

    private static final String BACKUP_COMPLETE = "BACKUP_COMPLETE";

    public RecipeBackupService() {
        // name service for debugging purposes
        super("RecipeBackupService");
    }

    @Override
    protected void onHandleIntent(Intent workIntent) {
        // TODO: backup recipes...
        Log.d(TAG, "Backup complete");

        // then communicate with app
        broadcastComplete();
    }

    private void broadcastComplete() {
        final Intent i = new Intent(BACKUP_COMPLETE);
        sendBroadcast(i);
    }
}
```


LISTING 20-10: AndroidManifest entry for the RecipeBackupService

```
<service android:name=".service.RecipeBackupService"/>
```

How might a Service communicate to an app?

As discussed previously in this chapter, an app can use `BroadcastReceivers` to receive system events. However, apps may use them internally as well. One useful way to use `BroadcastReceivers` is to have a `Service` communicate with an app by sending `Intents` to its `BroadcastReceivers`.

For example, the `RecipeBackupService` in Listing 20-9 sends a broadcast when the backup is completed. For this, it uses the code within the `broadcastComplete()` method.

How does an app create a persistently running Service?

Sometimes an app needs to have a `Service` that runs continuously. For example, a `Service` such as the one in Listing 20-11 runs continuously so it can maintain a consistent chat connection using a `ChatConnection` class. Because `CookingChatService` extends `Service` instead of `IntentService`, it needs to handle more of its life cycle.

One important aspect of the life cycle is how long the `Service` should run and what should happen if Android needs to stop it. To control this aspect, the `Service` has an `onStartCommand()` method that returns `START_REDELIVER_INTENT`. The value tells Android to restart the `Service` should it need to halt it. Apps use various other return values to control how the `Service` is restarted.

LISTING 20-11: Cooking Chat

```
public class CookingChatService extends Service {
    private static final String TAG = "CookingChatService";

    private ChatConnection connection;

    private final IBinder mBinder = new CookingChatBinder();

    @Override
    public void onCreate() {
        connection = new ChatConnection();
        super.onCreate();
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        connection.connect();
        // restart in case the Service gets canceled
        return START_REDELIVER_INTENT;
    }

    @Override
```

continues

LISTING 20-11 *(continued)*

```

    public IBinder onBind(Intent intent) {
        return mBinder;
    }

    public ChatConnection getConnection() {
        return connection;
    }

    @Override
    public void onDestroy() {
        Log.d(TAG, "Chat Service SHUTDOWN");
        super.onDestroy();
    }

    public class CookingChatBinder extends Binder {
        public CookingChatService getService() {
            return CookingChatService.this;
        }
    }
}

```

How does an app bind to a Service?

Using Intents to communicate with a Service works well when the data is simple, but sometimes it is more convenient to call methods directly on the Service. For this an app may *bind* to the Service.

For example, if a Service maintains a complicated connection object that has many methods, it might be easier for the Service to give the app access to that object through binding rather than having different Intents for each method in the connection object. Listing 20-12 shows how an Activity can bind to a Service, access its connection object, and call methods on it.

LISTING 20-12: Binds to the CookingChat Service

```

public class CookingChatLauncher extends Activity {
    private static final String TAG = "CookingChatLauncher";

    private boolean bound;

    private CookingChatBinder binder;

    private EditText message;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.chat_layout);
        findViewById(R.id.bt_send).setOnClickListener(

```

```

        new View.OnClickListener() {
            @Override
            public void onClick(View arg0) {
                sendMessage();
            }
        });
        message = (EditText) findViewById(R.id.et_message);
    }

    private void sendMessage() {
        if (bound && binder != null) {
            final String messageToSend = message.getText().toString();
            binder.getService().getConnection().sendMessage(messageToSend);
            Toast.makeText(this, "Message sent: " + messageToSend,
                Toast.LENGTH_SHORT).show();
        }
    }

    // Note: use onStart and onStop so that the app doesn't spend a lot
    // of time reconnecting, but still will release the resources should
    // app terminate
    @Override
    public void onStart() {
        if (!bound) {
            Log.d(TAG, "not bound, binding to the service");
            final Intent intent = new Intent(this, CookingChatService.class);
            bindService(intent, serviceConnection, Context.BIND_AUTO_CREATE);
        }
        super.onStart();
    }

    @Override
    public void onStop() {
        if (bound) {
            Log.d(TAG, "unbinding to service");
            unbindService(serviceConnection);
            bound = false;
        }
        super.onStop();
    }

    private ServiceConnection serviceConnection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName className, IBinder service) {
            Log.d(TAG, "service connected");
            binder = (CookingChatBinder) service;
            bound = true;
        }

        @Override
        public void onServiceDisconnected(ComponentName arg0) {
            bound = false;
        }
    };
}

```

USER INTERFACE

Arranging layouts can be confusing to beginning Android programmers, but intuitive for experts, who may skip the UI designer provided by Android IDEs and edit layout files directly. This section describes how to use the various layout mechanisms to create the user interfaces shown in Figure 20-2. It also discusses *Fragments*, which are essential for properly organizing a user interface.

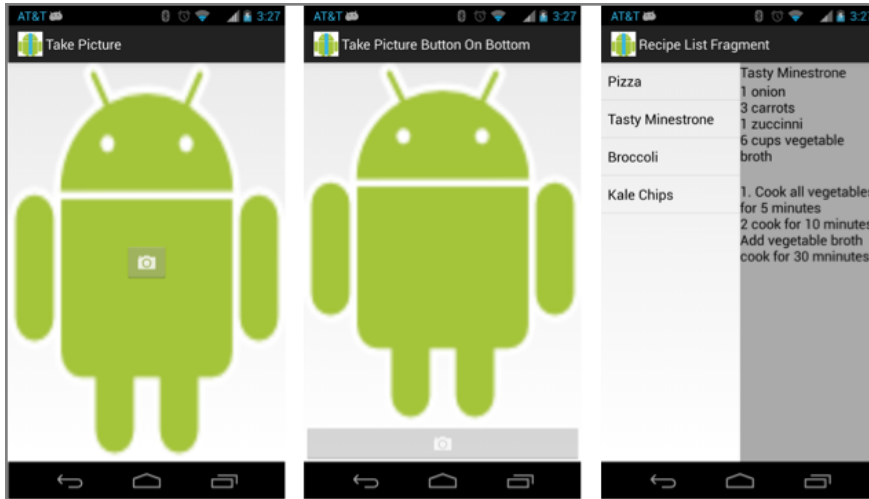


FIGURE 20-2

How does an app overlay one view on top of another?

Apps use a *FrameLayout* to position one view on top of another. All the views within a *FrameLayout* fill up their entire limits. Hence, when an app puts two views within a *FrameLayout*, the first one is on the bottom and the second is on top.

Listing 20-13 shows a *FrameLayout* that places a camera button on top of an *ImageView*. An image of the layout is shown on the left in Figure 20-2. In addition to setting the order of the views, the layout also needs to set the *layout_width* and *layout_height* attributes to make the *ImageView* take up the whole screen and the camera icon take up just the size of its image. It also needs to position the camera icon in the center of the *FrameLayout* by setting *layout_gravity*.

LISTING 20-13: A relative layout placing a camera button in front of an *ImageView*

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
```

```

        android:layout_height="match_parent">

        <ImageView
            android:id="@+id/iv_picture"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:scaleType="fitXY"
            android:src="@drawable/android_robot"/>

        <ImageButton
            android:id="@+id/bt_take_picture"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:src="@drawable/device_access_camera"/>

    </FrameLayout>

```

How do you put a button on the bottom of the screen below another view that takes up the remainder of the screen?

`RelativeLayout` is useful for positioning Views next to each other. Similar to the `FrameLayout`, all Views take up the entire space. `RelativeLayouts` have the extra feature of allowing apps to adjust how Views are positioned using relative parameters such as `layout_below`, `layout_toRightOf`, or `layout_align_parent_bottom`.

Several attributes must be set to achieve a layout where a button is on the bottom of the screen and another View takes up the remainder of the space. First, the layout anchors the button to the bottom by setting the `layout_alignParentBottom` attribute. Second, it sets the `android:layout_above` setting to position the `ImageView` above the button. Otherwise, the `ImageView` would take up the entire View like all Views in the `RelativeLayout` do and hide the button. Listing 20-14 shows how to implement it. An image of the layout is shown in the middle of Figure 20-2.

LISTING 20-14: Layout for button a button below another View

```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ImageView
        android:id="@+id/iv_picture"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_above="@+id/bt_take_picture"
        android:scaleType="fitXY"
        android:src="@drawable/android_robot"/>

    <ImageButton

```

continues

LISTING 20-14 *(continued)*

```

        android:id="@+id/bt_take_picture"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:src="@drawable/device_access_camera" />

</RelativeLayout>

```

How does an app distribute layout among components within the available space?

Sometimes an app needs to distribute Views within the available space. One way to achieve this is to divide up the space proportionally by using a `LinearLayout`, specifying the `weightSum`, and entering `layout_weight` attributes for each view within the `LinearLayout`. Listing 20-15 shows how to lay out two `Fragments` so they each contain 50 percent of the available space. The right side of Figure 20-2 shows how this layout looks.

LISTING 20-15: Layout for splitting available space between two views

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal"
    android:weightSum="100" >

    <fragment
        android:id="@+id/frag_recipe_list"
        android:layout_width="0dip"
        android:layout_height="match_parent"
        android:layout_weight="50"
        class="com.wiley.acinginterview.recipe.frag.RecipeListFragment" />

    <fragment
        android:id="@+id/frag_recipe_detail"
        android:layout_width="0dip"
        android:layout_height="match_parent"
        android:layout_weight="50"
        class="com.wiley.acinginterview.recipe.frag.RecipeDetailFragment" />

</LinearLayout>

```

Alternatively, the app could distribute space to a single column or row where each view takes up as much space as it needs. For example, an app might create a vertical `LinearLayout`, such as the one in Listing 20-16, which contains several `TextViews` with each taking up as much space as the text

they hold. The `LinearLayout` specifies that the orientation is vertical so the views inside will fill up a column. The recipe text in the right side of Figure 20-2 shows how the recipe detail view looks.

LISTING 20-16: Positions text views vertically in a column where each `TextView` is below the previous one and as big as is needed to hold its text

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:background="@android:color/darker_gray" >

    <TextView
        android:id="@+id/tv_recipe_id"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceMedium" />

    <TextView
        android:id="@+id/tv_ingredients"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceMedium" />

    <TextView
        android:id="@+id/tv_steps"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceMedium" />

</LinearLayout>
```

What is a **Fragment**?

A **Fragment** is a portion of a user interface, and it shares a similar life cycle to an **Activity**. The advantages of using **Fragments** include:

- Reduces complexity of the **Activity** class because each **Fragment** can handle its own state and life cycle
- Makes it easy to reconfigure a user interface. This feature is very helpful when apps have different layouts for different orientations or when apps need to show and hide parts of their user interfaces.

How do apps include a **Fragment** in a layout?

To use **Fragments**, an app may need several objects. An app potentially needs to get a reference to a **Fragment**, show it if it is not already shown, pass some data to it, and receive data from it.

To demonstrate these concepts, consider an app that has two `Fragment`s: one for a recipe list that appears on the left side of the screen and another for a recipe detail screen that appears on the right. The two `Fragment`s need to interact to display the recipe in the detail `Fragment` when the user clicks a recipe in the list. The right side of Figure 20-2 shows how this looks. An app like that requires several components.

First, an `Activity` creates a layout containing the `Fragment`s, and passes data between them, as shown in Listing 20-17.

LISTING 20-17: Activity that manages the `Fragment`s

```
public class RecipeViewAsFragments extends FragmentActivity implements
    RecipeActions {
    private RecipeDetailFragment recipeDetail;

    private RecipeListFragment recipeList;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.side_by_side);

        recipeDetail = (RecipeDetailFragment) getSupportFragmentManager()
            .findFragmentById(R.id.frag_recipe_detail);
        recipeList = (RecipeListFragment) getSupportFragmentManager()
            .findFragmentById(R.id.frag_recipe_list);
        hideRecipeDetail();
    }

    @Override
    public void showRecipe(String recipeId) {
        final FragmentTransaction ft = getSupportFragmentManager()
            .beginTransaction();
        ft.show(recipeDetail);
        ft.setCustomAnimations(android.R.anim.slide_in_left,
            android.R.anim.slide_out_right);
        ft.addToBackStack("show recipe");
        ft.commit();

        recipeDetail.loadRecipe(recipeId);
    }

    public void hideRecipeDetail() {
        final FragmentTransaction ft = getSupportFragmentManager()
            .beginTransaction();
        ft.hide(recipeDetail);
        ft.setCustomAnimations(android.R.anim.slide_in_left,
            android.R.anim.slide_out_right);
        ft.commit();
    }
}
```


Second, two similar `Fragment`s provide user interfaces for the recipe list and recipe detail functionality. Listing 20-18 shows the `RecipeListFragment`. It displays recipes in a `ListView` and receives the `RecipeActions` interface during the `onAttach()` method. `RecipeDetailFragment` has a similar implementation.

LISTING 20-18: Displays the recipe list

```
public class RecipeListFragment extends ListFragment {
    private RecipeActions recipeActions;

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        setRecipes();
    }

    /**
     * create the recipe list
     */
    public void setRecipes() {
        final List<String> recipes = getRecipes();
        final ArrayAdapter<String> adapter = new ArrayAdapter<String>(
            getActivity(), android.R.layout.simple_list_item_1, recipes);
        setListAdapter(adapter);
        getListView().setOnItemClickListener(
            new AdapterView.OnItemClickListener() {
                @Override
                public void onItemClick(AdapterView<?> parent, View view,
                    int position, long id) {
                    final String recipeClicked = recipes.get(position);
                    recipeActions.showRecipe(recipeClicked);
                }
            });
    }

    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        recipeActions = (RecipeActions) activity;
    }

    private List<String> getRecipes() {
        final List<String> recipes = Arrays.asList("Pizza", "Tasty Minestrone",
            "Broccoli", "Kale Chips");
        return recipes;
    }
}
```

Finally, the code also needs an interface to abstract away the concrete `Activity` and to enable communication between `Fragment` and `Activity` through a generic interface. Listing 20-19 shows the `RecipeActions` interface.

LISTING 20-19: Interface for the Activity to be used by its Fragments for communication to each other

```
public interface RecipeActions {  
    public void showRecipe(String recipeId);  
}
```

The Fragments are organized with the layout in Listing 20-15. In this case, the layout is a side-by-side layout of the `RecipeListFragment` on one side and the `RecipeDetailFragment` on the other.

How does an app manipulate a `Fragment`?

Once an app has a reference to a `Fragment`, it can animate, show, and hide it using `FragmentTransactions`. Optionally, an app can also add the `FragmentTransaction` to the backstack it needs to allow the user to hit the back button to undo the transaction. All these features reduce the code that apps need to manipulate sub-chunks of user interfaces. The `showRecipe()` method from Listing 20-17 has a `FragmentTransaction` that shows a new `Fragment` with a slide-in animation and adds the showing to the backstack. If the user hits the back button, the app undoes the transaction.

How does an `Activity` pass data to a `Fragment`?

If an `Activity`'s layout has multiple `Fragments`, it typically has the job of passing data between them. When an `Activity` has a reference to a `Fragment`, it can then call methods directly on them to pass data.

For example, in `RecipeViewAsFragments`, when the user clicks a recipe in the recipe list, the `RecipeViewAsFragments` passes the new recipe ID to the recipe detail `Fragment` by calling `loadRecipe()`.

How does a `Fragment` pass data to an `Activity`?

The Android documentation recommends an approach for allowing a `Fragment` to pass data back to the `Activity`. First, the `Activity` implements an interface, then in the `onAttach()` method, the app stores the reference to that interface for later use.

For example, in `RecipeViewAsFragments` the `RecipeActions` interface defines some methods the `Fragments` can use to coordinate. `RecipeListFragment`, shown in Listing 20-18, uses the `RecipeActions` interface to have `RecipeViewAsFragments` call the `RecipeDetails` when the user clicks a recipe.

PERSISTENCE

Android offers various options to allow an app to persist different kinds of data. An app might use different mechanisms for different purposes, so Android developers have to know which kinds of persistence methods are available and how to apply them properly.

How does an app persist name/value pairs?

Name/value pairs are a convenient storage mechanism, and as such, Android provides a special system called `SharedPreferences` to help apps store and retrieve them.

Apps save and retrieve `SharedPreferences` using the `SharedPreferences` API. Listing 20-20 shows how an app can access `SharedPreferences`, write a value, and then read the value just written.

LISTING 20-20: Reads and writes `SharedPreferences`

```
public class SharedPreferencesTester extends AndroidTestCase {
    public void testSaveName() {
        final String PREF_NAME = "name";
        final String PREF_NAME_DEFAULT = "none";
        final String PREF_NAME_SET = "Shooting Star";
        SharedPreferences preferences = getContext().getSharedPreferences(
            "Test", Context.MODE_PRIVATE);

        // write a value
        Editor editor = preferences.edit();
        editor.putString(PREF_NAME, PREF_NAME_SET);
        editor.commit();

        // read a value
        String pref = preferences.getString(PREF_NAME, PREF_NAME_DEFAULT);
        assertTrue("pref now set", pref.equals(PREF_NAME_SET));
    }
}
```

How does an app persist relational data in a database?

The Android OS contains a `sqlite` database, and hence it is the default method for persisting any relational data. To access the database in Android, apps use a helper class called `SQLiteOpenHelper`. Listing 20-21 shows what an app must extend to use `SQLiteOpenHelper`. The listing shows that `SQLiteOpenHelper` provides convenient methods to handle creating and updating a database. The other methods in the abstract class help an app perform various database functions.

LISTING 20-21: The database helper with unimplemented method

```

public class DatabaseHelperBlank extends SQLiteOpenHelper {
    private static final String DATABASE_NAME = "WILEYDB";

    public DatabaseHelperBlank(Context context) {
        super(context, DATABASE_NAME, null, 1);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    }
}

```

To implement the `SQLiteOpenHelper`, an app first must be able to create a database using code such as the code in Listing 20-22. There is nothing special about this SQL statement except that it contains the `BaseColumns._ID` field. Android developers know to use this field in all their databases, because certain other Android classes, such as the `SQLiteCursor`, require it.

LISTING 20-22: Creates the database

```

@Override
public void onCreate(SQLiteDatabase db) {
    final String createStatement = "CREATE TABLE " + TABLE_NAME + " ("
        + BaseColumns._ID + " INTEGER PRIMARY KEY AUTOINCREMENT,"
        + COLUMN_NAME + " TEXT, " + COLUMN_AGE + " INTEGER)";
    db.execSQL(createStatement);
}

```

Beyond creating a database, Android apps use the functions within `SQLiteOpenHelper` to access the database. Listing 20-23 shows how an app would query the database for ages. The code gets a reference to a `SQLiteDatabase` from the `DatabaseHelper` singleton and then uses the `query()` method to execute a query. It also makes sure to close the cursor using a `finally` statement. `DatabaseHelper` contains a complete implementation of the empty methods in `DatabaseHelperBlank`.

LISTING 20-23: A database query

```

public int queryForAge(String name, Context context) {
    int age = -1;
    final DatabaseHelper helper = DatabaseHelper.getInstance(context);
    final SQLiteDatabase db = helper.getReadableDatabase();
    final String[] columns = new String[] { COLUMN_NAME, COLUMN_AGE };
    final String selection = COLUMN_NAME + "=?";
}

```

```

final String[] args = new String[] { name };

Cursor cursor = db.query(TABLE_NAME, columns, selection, args, null,
    null, null);
try {
    final boolean hasData = cursor.moveToFirst();
    if (hasData) {
        final int columnIndex = cursor.getColumnIndex(COLUMN_AGE);
        age = cursor.getInt(columnIndex);
    }
} finally {
    cursor.close();
}
return age;
}

```

How does an app store data in files?

Android presents apps with a filesystem that they can use with the familiar `java.io` libraries. However, Android separates the filesystem into several internal and external areas that have different uses.

What is the difference between internal and external file storage?

Several differences exist between internal and external file storage that deal with availability, accessibility, and what happens when the user uninstalls an app:

- **Availability**—Internal storage is always available, whereas external storage may not be if the users happen to remove or unmount their SD card.
- **Accessibility**—Files installed into internal storage are accessible only by the app, whereas external files are readable by any app.
- **Uninstalling**—Android removes all internal files and removes external files only if they are stored in a specially named directory.

Apps can access these different areas using APIs within a `Context`. Listing 20-24 shows some lines of code an app might use to access the different filesystems. Notice that the code also includes a method for checking if the external files are available. This is essential any time an app accesses external storage.

LISTING 20-24: Accesses file directories

```

public class FilesTest extends AndroidTestCase {
    public void testFileStorage() {
        if (!isExternalStorageReadable()) {

```

continues

LISTING 20-24 *(continued)*

```

        fail("External storage not readable");
    }

    // path for the SD card
    String root = Environment.getExternalStorageDirectory()
        .getAbsolutePath();
    // path for external files that Android deletes upon uninstall
    String externalFilesDir = getContext().getExternalFilesDir("myfiles")
        .getAbsolutePath();
    assertTrue(
        externalFilesDir,
        externalFilesDir
        .contains("Android/data/com.wiley.acinginterview/files/myfiles"));
    // path to internal storage directory
    String internalFilesDir = getContext().getFilesDir().getAbsolutePath();
    assertTrue(internalFilesDir,
        internalFilesDir
        .contains("/data/data/com.wiley.acinginterview/files"));
}

private boolean isExternalStorageReadable() {
    boolean readable = false;
    String state = Environment.getExternalStorageState();
    if (state.equals(Environment.MEDIA_MOUNTED)
        || (state.equals(Environment.MEDIA_MOUNTED_READ_ONLY))) {
        readable = true;
    }
    return readable;
}
}

```

When might an app use internal versus external file storage?

Apps should limit use of internal storage because on some devices, internal storage is separate from the external storage and may be limited. However, internal storage is useful for storing sensitive data that other apps should not have access to. External files are good for large files. In general, apps should utilize the external files directory for any file unless they want the user to be able to manipulate them. In such cases, apps sometimes create directories in the root of external storage. Having files stored there would also allow users to keep their data in between installations, which may be desirable.

ANDROID HARDWARE

One of the features that makes Android programming unique is access to the device's hardware. Apps might access hardware to do things like access the camera to help the user take a picture, sense the device's motion or location, activate the calling function, or access other hardware.

Many different kinds of hardware are available, and each has its own intricacies and APIs. However, the APIs do have some common patterns. Any kind of hardware access requires three things: understanding some details about how the hardware works so an app can properly use it; accessing it through the appropriate API; and properly handling resources and battery life.

This section discusses just one kind of Android hardware called Location Services that allows apps to know the device's location with varying degrees of accuracy. Location Services is an important feature of Android devices, so Android developers need to know how it works. What's more, accessing location data is similar to other kinds of hardware access so it serves as a good, representative example.

How does an app use the Location Services API within an Activity?

To access a device's location, apps register to get updates from Google Play services. Then when done, apps unregister. The paradigm of releasing resources when an app is done with them is common for hardware access, where the hardware in question consumes significant power or needs to be explicitly stopped.

Using the Location Services API requires several steps to register to receive updates, receive them, and then unregister when the app doesn't need them anymore.

Listing 20-25 shows an example of using Location Services. It is a simplified version of the official Google code available from <http://developer.android.com/training/location/receive-location-updates.html>. It has several important parts:

- **Specifies location requirements**—During `onCreate()`, the app creates a `LocationRequest`. It describes how frequently the app wants to receive updates and how accurate they need to be.
- **Connects**—During `onStart()` the Activity connects to Google Play services. When Location Services is ready, Android calls `onConnected()`.
- **Registers to receive updates**: In `startPeriodicUpdates()`, the app registers a `LocationListener`.
- **Receives location updates**—Location Services passes `Location` objects back to `onLocationChanged()`.
- **Release resources**—When the user leaves the app, it calls `onStop()`, which stops further updates and disconnects. This step is important because without it, the device would continue to collect locations forever.

LISTING 20-25: Senses location

```
public class LocationActivity extends Activity implements
    GooglePlayServicesClient.ConnectionCallbacks,
    GooglePlayServicesClient.OnConnectionFailedListener, LocationListener {
    private static final String TAG = "LocationActivity";

    // A request to connect to Location Services
```

continues

LISTING 20-25 *(continued)*

```
private LocationRequest mLocationRequest;

// Stores the current instantiation of the location client in this object
private LocationClient mLocationClient;

private TextView locationLog;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.locations);

    locationLog = (TextView) findViewById(R.id.tv_location_log);
    createLocationRequest();
}

private void createLocationRequest() {
    mLocationRequest = LocationRequest.create();
    mLocationRequest.setInterval(5000);
    mLocationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
    mLocationRequest.setFastestInterval(1000);

    mLocationClient = new LocationClient(this, this, this);
}

// control connecting and disconnecting during visible time
// of the activity

@Override
public void onStop() {

    if (mLocationClient.isConnected()) {
        stopPeriodicUpdates();
    }
    mLocationClient.disconnect();

    super.onStop();
}

@Override
public void onStart() {
    mLocationClient.connect();
    super.onStart();
}

@Override
public void onConnected(Bundle bundle) {
    final int resultCode = GooglePlayServicesUtil
        .isGooglePlayServicesAvailable(this);
    // If Google Play services is available
    if (ConnectionResult.SUCCESS == resultCode) {
        startPeriodicUpdates();
    }
}
```



```

        } else {
            Log.e(TAG, "cannot connect to location services");
        }
    }

    private void startPeriodicUpdates() {
        mLocationClient.requestLocationUpdates(mLocationRequest, this);
    }

    public void stopPeriodicUpdates() {
        mLocationClient.removeLocationUpdates(this);
    }

    @Override
    public void onDisconnected() {
        Log.e(TAG, "disconnected");
    }

    @Override
    public void onConnectionFailed(ConnectionResult connectionResult) {
        Log.e(TAG, "failed to connect");
    }

    @Override
    public void onLocationChanged(Location loc) {
        Log.d(TAG, "LOCATION UPDATE");
        final String summary = loc.getLatitude() + ", " + loc.getLongitude()
            + " " + loc.getAccuracy();
        locationLog.setText(summary);
    }
}

```

What factors affect Location performance and battery usage?

The main intricacy with Location Services is conserving battery life while getting the required accuracy and frequency of updates. Apps also may need to take into account the environment in which the app is operating. For example, indoor or urban environments make GPS-based location problematic; however, the network-based locations can continue to provide locations at lower accuracy. Hence, when working with Location Services an app needs to take these considerations into account.

How does an app minimize battery usage?

To help an app make the trade-off between battery life and accuracy and frequency of updates, the `LocationRequest` has several explicit priorities that apps can use:

- `PRIORITY_HIGH_ACCURACY`—For applications such as tracking the user's location on the screen, that require frequent, high accuracy updates

- `PRIORITY_BALANCED_POWER_ACCURACY`—For applications that want to receive locations at a certain interval, but also want to receive locations more frequently if possible
- `PRIORITY_LOW_POWER`—For applications that do not need a high accuracy, where “city” level locations are appropriate
- `PRIORITY_NO_POWER`—For applications that do not have a critical need for up-to-date locations, but still want to receive them. Using this setting produces no additional effect on power usage, because it receives updates only when other apps request locations.

SUMMARY

This chapter reviewed some important areas of Android development. Four main Android components allow apps to show user interfaces, run background services, receive broadcast `Intents`, and store shared data. The components communicate using `Intents` and `ContentResolvers`. Layouts can be challenging, but by using `FrameLayout`, `RelativeLayout`, and `LinearLayout` apps can create a number of arrangements of `Views`. `Fragments` help to isolate pieces of the user interface. Apps can persist name/value pairs, databases, and files. Finally, apps can access the device’s hardware. Location Services is one kind of Android hardware that requires minimizing battery usage while achieving the best accuracy possible.

Overall, this chapter covered some of the important concepts you should be familiar with as an Android developer. This chapter complements the skills described in the rest of this book so that when combined, this book describes a comprehensive set of skills a successful Android developer needs.

This final chapter illustrates that the technology landscape and specifically the Java language is always changing: Java is being used in more innovative ways, in ways you wouldn’t always expect, and that makes it very important to keep your skills and knowledge up to date.

If you only take one thing from reading this book, it is that you *must* keep your skills up to date. With new libraries and frameworks being introduced regularly and even new technologies using Java, your next interview may be for a role developing on a platform you have never used before.

Practice writing code regularly. You should get enough practice if you have a current day job as a Java developer, but do try to get experience outside of your current domain, and use new libraries. You may even be able to introduce some new concepts in your current role.

Good luck and best wishes with finding your next role as a Java developer, and hopefully this book has helped in making your next technical interview a successful one.

APPENDIX

Introducing Scala

Recent years have seen many new languages created with different properties than Java, but that run on the JVM. One such language is Groovy, a dynamic language that is often used in scripting, testing, and other places where you may want to remove a lot of Java's verbose boilerplate code. Jython is a version of Python that runs on the JVM, and Clojure is a dialect of Lisp that compiles into Java bytecode.

Scala is a functional programming language for the JVM. This chapter introduces Scala, and provides a few questions that could possibly come up in an interview if you ever have any exposure to the language.

Scala introduces some new concepts that may not be familiar to experienced Java programmers, such as functions as values, and a concise language that allows the compiler to infer the types of variables and return types of functions.

You can download Scala from the official website at <http://www.scala-lang.org/download/>, or you can include it as part of your build automatically by using a plug-in. If you use Maven, the following plug-in will compile any Scala source in your project's `src/main/scala` directory:

```
<plugin>
  <groupId>org.scala-tools</groupId>
  <artifactId>maven-scala-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>compile</goal>
        <goal>testCompile</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

SCALA BASICS

Scala's creator, Martin Odersky, describes Scala as "a blend of object-oriented and functional programming concepts." If Scala is new to you, a common first step is to write "Scala as Java," using the object-oriented techniques, and then improve on this as you understand more of the functional elements of the language and gain more experience.

Scala source code compiles into Java class files, so any valid Scala application will run on a JVM. It is certainly possible to have a source repository that contains both Scala and Java source.

How are classes defined in Scala?

Listing A-1 shows some of the core language features of Scala.

LISTING A-1: A sample Scala class

```
class ScalaIntro {  
  
    val value: String = "This is a String"  
    var variable: Int = -100  
  
    def stringLength(s: String): Int = s.length  
  
    def writeOutput = {  
        println(s"variable is set to: $variable")  
        variable = 25  
        println(s"variable is now set to: $variable")  
  
        println(s"value is set to: $value")  
  
        println(s"The length of the string $value is: " + stringLength(value))  
    }  
}
```

This listing shows several ways of storing and manipulating data. The keyword `val` is what you use to store *values*: These are immutable, and cannot change. The `var` keyword is for *variables*, which are values that can change after being set. If you try to modify a `val`, the compiler will enforce that this cannot happen.

A function is defined with the `def` keyword. Parameter lists to the function are enclosed in parentheses, as a comma-separated list taking the format `name: Type`. The return type of the function is declared after the function parameter list. If your function has no parameters, such as `writeOutput`, you can omit the parentheses completely. Even the curly braces for a function are optional if the whole function is one line, such as `stringLength`. The `return` keyword is optional for the value returned from a function: The last statement in the function is the value returned.

The compiler is smart enough to work out where the end of a statement is; no semicolons are necessary, unless you have more than one statement on the same line.

Scala also has a more intuitive notion of equality: The `==` operator will automatically run the instance's `equals` method, rather than Java's default check that two objects are actually the same reference. You can still check for *referential equality* with the `eq` method.

Similar to classes, Scala also introduces the `object` keyword, which is used to represent *singleton classes*. These are classes of a single instance; and their methods can be thought of as similar to Java's static methods. To run code on the command line with Java, you need to define a static method, `main`. Because Scala compiles to Java class files, this is the same for Scala. Listing A-2 shows how to define some runnable code.

LISTING A-2: Creating a runnable Scala singleton class

```
object HelloWorld {
  def main(args: Array[String]) = {
    println("Hello world!")
    println("The parameters passed were: " + args.mkString(", "))
  }
}
```

Any code outside of a function or field declaration is run on object construction. Scala also provides a helper class, called `App`, that provides a `main` method. These two features can be combined to produce concise, runnable applications, as shown in Listing A-3.

LISTING A-3: Extending the `App` object

```
object RunWithApp extends App {
  val pi = 3.14
  val radius = 200

  val area = pi * radius * radius

  println(s"The area of the circle is: $area")
}
```

This is run as if all the statements were in a `main` method.

THE COMPILER CAN INFER TYPES

The compiler can infer many of the types of fields and functions. Notice that in Listing A-3, none of the `vals` had a type declared, but `pi` was treated as a floating-point number, and `radius` was treated as an integer.

While developing this code, if you decide to change a field—for instance, from a `Float` to an `Int`—you do not have a type declaration to change, which makes your development and debug cycle much faster.

What is a case class?

A case class is a class definition that works just like a regular Scala class, but the compiler fills in logical `equals` and `toString` methods, as well as immutable access to the fields of the class. You construct case classes without using the `new` keyword. Listing A-4 shows a simple example.

LISTING A-4: Using case classes

```
case class Person(name: String, age: Int, salary: Int)

object CaseClassExample extends App {
  val firstPerson = Person("Alice", 24, 3000)
  val secondPerson = Person("Bob", 25, 2500)
  val thirdPerson = Person("Charlie", 30, 1000)

  println(s"The first person is: $firstPerson")
  println("Second person and third person are equal: " + secondPerson == thirdPerson)
}
```

This `Person` class has no extra methods or fields, and is being used here in a way similar to the way you might use domain objects in Java. Case classes are simply the same as regular classes; they can have methods and fields, and you can override the filled-in methods of `equals`, `toString`, or even the fields passed as constructor parameters.

Case classes have one more powerful feature: Instances can be decomposed into their constructed fields, and can be used in a technique called *pattern matching*.

What is pattern matching?

Pattern matching works in a similar way to Java's `switch` statement, though you are not restricted to a small number of types to match. Listing A-5 shows a simple example of pattern matching on an integer.

LISTING A-5: Pattern matching

```
def matchInteger(i: Int): String = {
  i match {
    case 1 => "The value passed was 1"
    case 2 => "The value passed was 2"
    case x if x < 0 => s"The value passed was negative: $x"
    case _ => "The value passed was greater than 2"
  }
}
```

The `match` keyword is used in pattern matching, and each possible outcome is set in a `case` statement. Each case can be an explicit value, such as the first two cases in Listing A-5. It can also be parameterized, such as the negative case, which is also combined with a conditional statement. A default case is marked with an underscore, which in Scala often represents an ignored value.

Using case classes with pattern matching is a powerful feature. Listing A-6 is an example of how to decompose a case class to access its values.

LISTING A-6: Pattern matching with case classes

```
case class NameAgePair(name: String, age: Int)

def matchNameAgePair(pair: NameAgePair): String = {
  pair match {
    case NameAgePair("Alan", 53) => "Alan is 53"
    case NameAgePair("Alan", _) => "Alan is not 53"
    case NameAgePair(name, 53) => s"Another person, $name, is 53"
    case NameAgePair(name, age) => s"This unexpected person, $name, is $age"
  }
}
```

Similar to Listing A-5, you can match on explicit values *within* the case class, assign variables for access on the right-hand side of the case, or ignore the field completely.

How does the Scala interpreter enable faster development?

If you run the `scala` command from the command line with no parameters, you enter the Scala interpreter:

```
$ scala
Welcome to Scala version 2.10.2 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_17).
Type in expressions to have them evaluated.
Type :help for more information.
scala>
```

You can type commands into the console as valid Scala statements, and the interpreter will evaluate the statements and return values:

```
scala> val i = 100
i: Int = 100

scala> i < 200
res0: Boolean = true

scala> i + 34
res1: Int = 134
```

If you want to submit a large piece of code for evaluation, you can do so with the `:paste` command, which defers evaluation until you press Ctrl-D:

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

case class Car(model: String, mileage: Int)

val c = Car("Sedan", 10000)
val d = Car("SUV", 120000)

c == d
<CTRL-D>
// Exiting paste mode, now interpreting.

defined class Car
c: Car = Car(Sedan,10000)
d: Car = Car(SUV,120000)
res2: Boolean = false
```

Being able to use the Scala interpreter as a scratchpad can let you work on prototypes and understand specifics about the language, or even a certain domain, by running and examining small snippets of code.

FUNCTIONS AS VALUES

How does Scala represent functions?

A function can be treated as a value. That is, a function can be assigned to a `val` or `var`, or even returned from another function. Listing A-7 shows how to define the syntax for a function definition, and how to use functions as values.

LISTING A-7: Treating functions as values

```
object FunctionValue extends App {

  val square: (Int => Int) = x => x * x
  val threeSquared = square(3)
  val threeSquared = square(3)

  def printNumber(string: String): (Int => String) =
    number => string + " " + number.toString

  val numberPrinter = printNumber("The value passed to the function is:")
  println(numberPrinter(-100))
}
```


The value held in the `square` field has the type `Int => Int`. That is, it takes an `Int`, and returns an `Int`. You can then treat `square` just like a regular function, by passing parameters.

The body of the `square` field is a function, so it has a parameter `x`, and the body of that function is defined after the `=>` identifier.

The `printNumber` function takes a `String`, and returns a function that itself takes an `Int` and returns a `String`. The function returned from `printNumber` uses the passed `String` parameter to create a function that uses the given `String`.

What is a higher-order function?

A *higher-order function* is defined as a function that takes another function as a parameter. Listing A-8 shows an example.

LISTING A-8: Using a higher-order function

```
case class BankAccount(accountId: String, balance: Double)

object HigherOrder extends App {

  def alterAccount(account: BankAccount, alteration: (Double => Double)):BankAccount = {
    val newBalance = alteration(account.balance)
    BankAccount(account.accountId, newBalance)
  }

  def applyMinimalInterest(amount: Double) = amount * 1.01
  def applyBonusInterest(amount: Double) = amount * 1.09

  val accountOne = BankAccount("12345", 100.00)
  println(alterAccount(accountOne, applyMinimalInterest))

  val accountTwo = BankAccount("55555", 2000000.00)
  println(alterAccount(accountTwo, applyBonusInterest))
}
```

The `alterAccount` function provides a generic way to make changes to the balance of a `BankAccount` without having to provide the specifics. Both functions, `applyMinimalInterest` and `applyBonusInterest`, have the type `Double => Double`, so these can be applied to the `alterAccount` function.

If you need to provide different ways to change the balance of a `BankAccount`, you can do so without needing to change the `alterAccount` function at all.

Many of Scala's core libraries use higher-order functions to provide ways to manipulate the data in their data structures. Listing A-9 shows two ways to manipulate Scala's `List` object: changing each value and filtering the list.

LISTING A-9: Higher-order functions on Scala's list type

```
object ListManipulations extends App {

  val list = List(-3, -2, -1, 0, 1, 2, 3, 4, 5)

  def absolute(value: Int): Int = {
    value match {
      case x if x < 0 => -value
      case _ => value
    }
  }

  def isPositive(value: Int): Boolean = {
    value >= 0
  }

  println(list.map(absolute))
  println(list.filter(isPositive))
}
```

Running this application produces the following output:

```
List(3, 2, 1, 0, 1, 2, 3, 4, 5)
List(0, 1, 2, 3, 4, 5)
```

The `map` method on the `List` type applies the function provided to each value in the list, returning each value in a new list. The `filter` method applies the function provided to each value, and if the predicate returns true, then that value is added to the new list to be returned.

METHODS AND FUNCTIONS

Sometimes the words “method” and “function” are used interchangeably, but there is a very distinct difference between the two.

A *function* is a statement, or set of statements, that is run when the function is called by name. It is independent of any external factors or state. It has no association with an instance of a particular object.

A *method* is a statement, or set of statements, that is associated with an instance of a class. Running methods with the same parameters, but on a different object, may give a different result.

Most of the examples in this appendix so far have been functions, but `map` and `filter` in Listing A-9 are methods, because their result depends on the values contained in the `List` instance.

In Java, a static method is a function.

IMMUTABILITY

Why is immutability important in Scala?

Developers are drawn toward Scala because of its ability to build highly scalable systems. One key factor in building scalable systems is the use of immutable data. Immutable objects can be shared freely between different executing threads without a need to provide synchronization or locks, simply because the values cannot change.

Immutability is at the heart of Scala; you have already seen that case classes are immutable. The core data structures provided by Scala's libraries are immutable, too. Listing A-9 showed how to create a list; the `map` and `filter` methods demonstrated in that listing returned new `List` objects. The old list values had not changed, and any other code referencing the old list would never see any changes from the `map`, `filter`, or any other methods.

Lists in Scala are linked lists—each value points to the subsequent value in the list. Prepending a value onto a linked list does not change any of the other values in the list, making it a very efficient operation. Listing A-10 shows the syntax for doing so.

LISTING A-10: Prepending values onto lists

```
object Lists extends App {
    val listOne = List(1, 2, 3, 4, 5)
    val listTwo = 0 :: listOne
    val listThree = -2 :: -1 :: listTwo

    println(listOne)
    println(listTwo)
    println(listThree)
}
```

Pronounced *cons*, the `::` operator prepends the value on the front of a list instance. Figure A-1 shows how the lists are represented in memory.

When iterating over immutable, recursive data structures, such as the `List` object, it can make sense to do so using recursive functions. Listing A-11 shows how you can implement your own `find` method, searching for a given value in a Scala `List`. Be aware that `x :: xs` can be used in pattern matching, which decomposes a list into its head, `x`, and the remaining list, `xs`. `Nil` is an empty list.

LISTING A-11: Finding a value in a list

```
@tailrec
def find(value: Int, list: List[Int]): Boolean = {
    list match {
        case Nil => false
        case x :: _ if x == value => true
        case _ :: xs => find(value, xs)
    }
}
```

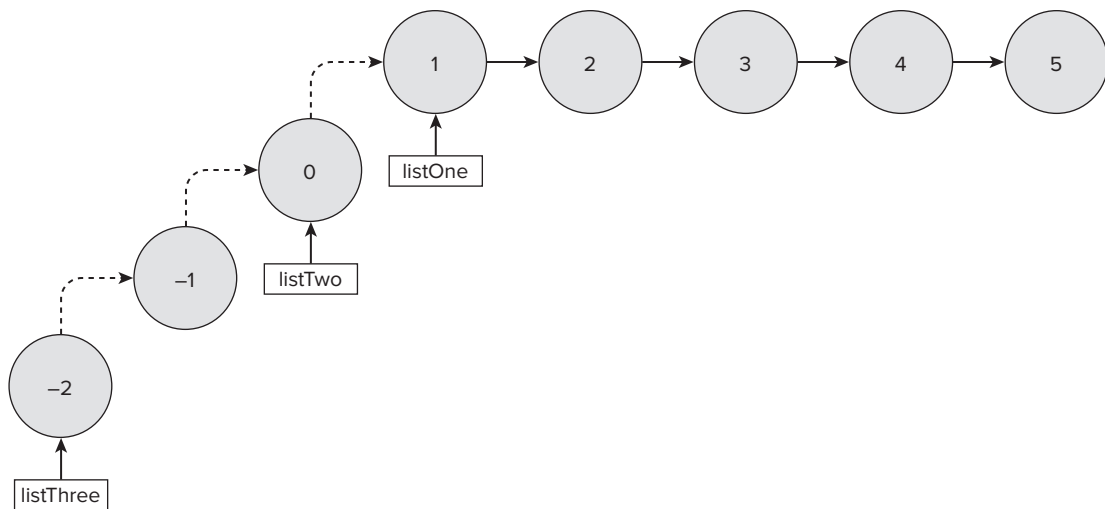


FIGURE A-1

The `case` statements are executed in order. The base case is evaluated first. If a given list is empty, the value is not in the list. Otherwise, if the head of the list matches the value to find, the function returns `true`. Finally, strip the head element from the list, and recursively call `find` again, until the value is found or the list terminates.

What is tail recursion?

This function is annotated with `@tailrec`. This annotation informs the compiler that you believe this function is tail-recursive.

A function is *tail recursive* if the recursive call is the last operation in the function; the method does not rely on the result of the recursive call for any more computation. The beauty of a tail-recursive function is that the compiler can rewrite the method in an iterative fashion, similar to a `while` loop, with mutable state. This allows long, deep recursion without the need for many stack frames, which could result in a `StackOverflowError`.

Any tail-recursive function can be replaced with an iterative version, using mutable variables to control the state. Recursive methods are often much shorter and easier to understand, so allowing the compiler to manage the state of the method is much safer.

If the compiler cannot make the function tail recursive, it will throw an error.

SUMMARY

This has been a brief, whirlwind introductory tour to Scala and some of its features. If this is new to you, hopefully this will serve as enough for you to take a deeper look into the language.

The key concepts introduced here, such as functions as values and immutability are not specific to Scala, but to functional programming in general. If you look at other functional languages, some have no ability to modify variables at all once they have been set.

The Scala ecosystem is growing at a rapid pace, with new features being added and new tools being available almost on a weekly basis.

Some other tools and frameworks include the *Simple Build Tool* (SBT), which is a way to build and deploy Scala applications. It is very different than tools such as Maven or Ant, but those tools will definitely help with your understanding of how SBT works, especially around dependency management. *Play*, the web application framework examined in Chapter 13, can be used with Scala source files, and *Akka*, discussed in Chapter 11, works very well with Scala. Both Play and Akka are developed as Scala applications. *Slick* allows you to query databases and treat the responses as plain Scala collections.

Many companies are looking to expand from Java into Scala, and most are willing to take on experienced Java developers who are keen to learn and use Scala. If you are willing, and take some time to learn and understand how Scala works, this could be a differentiating factor between you and another candidate. You can find plenty of good online resources for learning Scala, and Martin Odersky also runs free online functional programming classes.

If you cannot convince your current employer to allow you to develop your production code in Scala, then a good way to gain experience is to write your tests in Scala. Several good test frameworks are available, and most will integrate with modern build tools and continuous integration environments. *Specs2* and *ScalaTest* are two common test frameworks.

Scala is one of the more popular modern languages built to use the JVM. Many argue that the future of development will be a move away from Java, but still take advantage of the JVM and its tools. Make sure you keep track of what languages employers are looking for, and try to keep your skills up to date to make yourself as employable as possible.

INDEX

Symbols

- `::` (cons operator), Scala, 349
- `;` (semi-colon), technical interviews, 20
- (curly brackets)
 - JDBC, 195
 - scope, 149
- `[]` (double braces), syntactic sugar, 134
- `==` (equals signs), Scala, 343
- `?` (question mark)
 - JDBC, 195
 - wildcards, 106

Numbers

- 201 Created, 223
- 204 No Content, 223, 228
- 400 Bad Request, 224
- 403 Forbidden, 224
- 404 Not Found, 224
- 405 Method Not Allowed, 224
- 500 Internal Server Error, 224
- 503 Service Unavailable, 224

A

- absolute value, `Integer.MIN_VALUE`, 90
- `AbstractHandler`, 209
- `AbstractTransactionalJUnit4`
 - `SpringContextTests`, JUnit, 260–261
- ACID. *See* atomic, consistent, isolated, durable
- acronyms
 - naming conventions, 112
 - resume, 12
- Activities, 317
- Activity
 - Android, 314, 318–321
 - apps, 318–320
 - Fragment, 329–332
 - Intent, 315–316
 - life cycle, 318
 - Service, 324–325
- `ActorCrash`, 172

- `ActorRef`, 171, 174
- actors, concurrency, 169–174
- `ActorSystem.actorOf`, 170
- `addAll`, 295
- `addChild`, 135
- `addOne`, 81
- `AdminConsole`, 204
- `AdminConsoleServlet`, 202
- `@After`, 110, 125–126, 137, 146, 261
- `@AfterClass`, 125, 137
- Akka, 169–171, 351
- algorithms. *See also* interview algorithms; recursive algorithms
 - average-case value, 23
 - best-case values, 23
 - Big O Notation, 23–24
 - bubble sort, 26–27
 - garbage collection, 148–149
 - insert sort, 27–28
 - language-agnostic, 4
 - mark-and-sweep, 149
 - merge sort, 30–32
 - n , 23
 - $O(n^2)$, 23
 - on paper, 20
 - pseudocode, 20
 - quicksort, 29–30
 - recursive, 30
 - sorted lists, 24–32
 - sorting, 73
 - space complexity, 23–24
 - Strategy Pattern, 55
 - time complexity, 23–24
 - worst-case value, 23
 - writing core, 23–34
 - written technical test, 18
- alphabetize, 73
- `alterAccount`, 347
- anagrams, 72–74
- Android, 313–340
 - Activity, 314, 318–321
 - apps
 - Activities, 317

Android (*continued*)

- Activity, 318–320
- Fragment, 332
- Intents, 316–317
- resources, 320–321
- Service, 322–325
- View, 326–328
- battery, 339–340
- BroadcastReceiver, 314, 318–320, 321
- components, 314–315
- external storage, 335–336
- hardware, 336–340
- Intents, 315–317
- internal storage, 335–336
- Location Services, 337–339
- name/value pairs, 333
- persistent data, 323–324, 333–336
- relational databases, 333–335
- Service, 314, 322–325
- user interface, 326–332
- AndroidManifest.xml, 317, 322
- annotations, 109–111
 - JUnit, 109–110
 - Spring, 254–255
- ANR. *See* Application Not Responding
- Ant, 309–311
 - DBDeploy, 197
 - JUnit testing, 123, 124
- Apache
 - Commons, 209, 211
 - boilerplate code, 287–290
 - InputStream, 211–289
 - libraries, 287–290
 - Maven, 301
 - OutputStream, 288–290
 - String, 288
 - HttpClient, 225
- API library, method names, 20
- App, 343
- app, 214, 216
- Application, 215
- Application Not Responding (ANR), 322
- application.conf, 214
- ApplicationContext, 249, 251, 259
- applications. *See also* web applications
 - context, Spring, 245–255, 264
 - databases, 177–199
 - definition, Maven, 304–307
 - JSON, 241–242
- ApplicationServlet, 202
- apps. *See* Android
- ArithmeticException, 172
- arrayCopy, 36
- ArrayIndexOutOfBoundsException, 68
- ArrayList, 35, 37, 117, 133
 - bubble sort algorithm, 28
 - JUnit testing, 134
 - LinkedList, 38–39
- arrays
 - defining, 36
 - Java basics, 98
 - lists, 36–39
 - objects, 98
 - size reallocation, 37
 - toString(), 98
- Arrays, 24, 35
- artifactId, 305
- Artifactory, 301
- artifacts, Maven, 307–308
- asList, 35
- Assert, 124, 127
- assertArrayEquals, 127
- Assert.assertEquals(), 101
- assertEquals, 127, 128
- assertFalse, 127, 128
- assertNotNull, 127
- assertThat, 143
- assertTrue, 127, 128
- at-computer technical tests, 18–19
- atomic, consistent, isolated, durable (ACID), 190
- Atomic, 168–169
- AtomicBoolean, 168
- AtomicInteger, 168–169
- AtomicReference, 168
- authentication, 205
- AuthenticationStrategy, 226
- autoboxing, 107–108
- AutoCloseable, 115
- AUTO_INCREMENT, 276
- @Autowired, 254–255
- autowiring, 253–255
- average-case value, 23
- AVL Tree, 42–43
- await, 165

B

- balanced tree, 43
- base case, 149
- battery, 339–340
- BDD. *See* Behavior-Driven Development
- <bean>, 250
- @Before, 110, 125–126, 137, 146, 261
- @BeforeClass, 125, 137
- BEGIN...END, 195
- Behavior-Driven Development (BDD), 143–146
- best-case values, 23
- Big O Notation, 23–24, 65, 71
- BigInteger, 72, 90
- BiMap, 292

- Binary Help, 43
- binary representation, positive numbers, 89–90
- binary search, lists, 32–33
- binary search tree, 39–41
- binary tree, 39
- boilerplate code
 - Apache Commons, 287–290
 - Spring JDBC, 257–259
- Boolean, 47, 108
- boolean, 36, 88, 108
- BOOT_COMPLETED, 321
- boxing, 107–108
- BroadcastReceiver, 314, 318–320, 321
- bubble sort algorithms, 26–27
- BufferedOutputStream, 61
- build, 53, 305, 306
- build managers, 3
- build tools, 301–311
 - Ant, 309–311
 - Maven, 301–309
- Builder, 53
- Builder Pattern, 51–55
- build.xml, 309
- ByteArrayInputStream, 288
- ByteArrayOutputStream, 288
- bytecode, 147, 152–153, 156

C

- cache
 - Fibonacci sequence, 70
 - Flyweight Pattern, 62
- cachedThreadPool, 163
- Calendar, 296
- call, 160
- Callable, 160, 162
- CallableStatement, 195
- camel case, 111–112
- case class, Scala, 344
- case sensitivity, SQL statements, 178
- Cassandra, 191
- Catalina, 206
- catch, 130, 138
- chain
 - Builder Pattern, 53
 - exceptions, 113–115
 - filter, 204–205
- changelog, 197
- char, 88, 89, 99
- checkString, 130
- children, trees, 39, 44
- class, case class, Scala, 344
- Class, 135
- .class, 152–153
- ClassCastException, 25, 102
- classes
 - bytecode, 156
 - dependencies, 139
 - immutability, 115–116
 - Java Collections API, 117–118
 - JVM, 153
 - naming conventions, 111
 - recursive class definition, LinkedList, 75
 - Scala, 342–343
 - singleton, 343
 - Spring, 248
 - web applications, 207–208
- classloading, 153, 191
- ClassNotFoundException, 233
- ClasspathXmlApplicationContext, 249
- clear(), 117
- close(), 115
- code smell, 127
- code writing
 - at-computer technical tests, 19
 - face-to-face interviews, 7, 19–21
 - JUnit, 87–88
 - on paper, 18
- Collection, 48, 117, 292, 295
- Collections, 24
- Collections.sort, 25
- column families, Cassandra, 191
- comma-separated values (CSV), 268
- COMMIT, 190
- company information
 - technical tests, 6–7
 - telephone screening, 4–5
- Comparable, 24–25, 40
- Comparator, 24, 25–26, 40
- compare, 26
- compile, 302, 303, 305, 310
- compiler
 - Collections.sort, 25
 - Integer, 107
 - JIT, 153
 - JVM, 152
 - pop, 104
 - primitive types, 89
 - Scala, 343
- concurrency, 159–174
 - actors, 169–174
 - testing, 164–165
 - threads, 159–165
- ConcurrentHashMap, 48, 49
- ConcurrentMultiset, 290
- conf, 214
- @Configuration, 250, 251
- Connection, 193, 256
- connection pools, JDBC, 196

- CONNECTIVITY_CHANGE, 321
- ConsoleLogging, 56
- constant pool, String, 101
- constraints, tables, 186–187
- constructor-arg, 248
- constructor-ref, 249
- ContentProvider, 314–315
- Content-Type, 222
- context:component-scan, 264
- @ContextConfiguration, 260
- Context.startService(), 322
- continue, 28
- @Controller, 264
- controllers, 214, 216
- cookies, 226
- coordinates, Maven, 305
- copy, 288, 289
- copyOf, 294
- COUNT, 182, 188
- CountDownLatch, 132, 164, 165
- Counter, 166
- CounterSetter, 166–167
- countRowsInTable, 262
- cover letter
 - body, 14
 - resume, 11–15
 - writing, 14–15
- CREATE TABLE, 186
- createSuiteDescription, 135
- CSV. *See* comma-separated values
- Cucumber, 144–146
- CURDATE, 188
- currentIterator, 79–80
- curriculum vitae (CV). *See* resume
- customSorting, 26

D

- Data Definition Language (DDL), 186–187
- Data Manipulation Language (DML), 186–187
- data structures, 35–49
 - lists, 35–39
 - maps, 45–48
 - sets, 48–49
 - trees, 39–45
- DatabaseHelper, 334
- DatabaseHelperBlank, 334
- databases
 - applications, 177–199
 - developers, 3
 - Hibernate, 274–275
 - inserting data, 275–277
 - integration tests, 260–261
 - JDBC, 191–197
 - releases, 197

- JDK, 255–257
 - relational
 - Android, 333–335
 - JDBC, 191–192
 - SQL, 177–178
 - SQL, 177–191
- DATA_SUB, 188
- DATE, 188, 280
- Date, 297
- DateTime
 - duration, 297–298
 - Joda Time, 296–297
 - period, 298–299
- DBDeploy, 197, 199
- DDL. *See* Data Definition Language
- decompiler, 106–107
- Decorator Pattern, 60–61, 231–233
- decrement operator, 169
- def, 342
- DefaultHttpClient, 226
- DELETE, 186, 223, 230
- deleteFromTables, 262
- dependencies
 - classes, 139
 - JUnit testing, 138–143
- dependencies, Maven, 305
- dependency injection, Spring, 245–250
- deploy, 304, 311
- deployment descriptors, 201–202
- Deque, 39
- Description, 135
- design patterns, 51–64
 - Builder Pattern, 51–55
 - commonly used, 60–64
 - Decorator Pattern, 60–61, 231–233
 - examples, 51–60
 - Flyweight Pattern, 61–62, 101, 107
 - Null Object Pattern, 41, 62
 - Singleton Pattern, 63–64
 - Strategy Pattern, 55–58
 - Template Pattern, 58–60
- destroy, 204
- difference, 296
- @DirtiesContext, 260
- DispatcherServlet, 263
- dist, 215
- DividingActor, 172
- DML. *See* Data Manipulation Language
- doFilter, 204, 205
- Domain-Specific Language (DSL), 143
- doPut, 209
- double, 88
- downcasting, 89
- DSL. *See* Domain-Specific Language
- duration, Joda Time, 297–299
- dynamic state, 318–320

E

E, 104
 eager instantiation, 251, 282
 Echo, 216
 /echo, 216
 echoReply, 216
 echo-server, 213–214
 EchoServlet, 210
 Eclipse, 19, 97
 Eden Space, 148
 EJBs. *See* Enterprise Java Beans
 Element, 37
 else, 70
 Enterprise Java Beans (EJB), 245
 entities
 HQL, 281
 REST, 226
 Entry, 46
 entrySet(), 117
 Enum, 63
 eq, 343
 equals, 93, 96, 97, 127
 equals(Object other), 96
 Erlang, 169
 Error, 112
 event, 280
 Exception, 112
 exceptions. *See also specific exceptions*
 call, 160
 chain, 113–115
 handling, 112–115
 JUnit, 129–130
 runtime, 113
 try, 115
 exec, 310
 Executor, 161–163, 165
 ExecutorService, 161, 162
 expected, 130
 external storage, Android, 335–336

F

Facebook, 13
 face-to-face interviews, 7–8, 19–21
 factorials, 71–72
 fail, 127
 Fibonacci sequence, 67–71
 FileDictionary, 248
 FileLogging, 56
 FileOutputStream, 60, 61
 Filter, 204
 filter, 59, 348, 349
 filter chain, 204–205
 FilterChain, 205
 FilterConfig, 204

filter-mapping, 204
 final, 36, 92
 finalize, 154, 155
 finally, 256–257
 finalName, 309
 first child, 44
 @FixMethodOrder, 262
 FizzBuzz, 65–67
 Float, 108
 float, 88, 108
 Flyweight Pattern, 61–62, 101, 107
 forEach, 120
 foreign keys, 181, 186–187
 Fragment, 328, 329–332
 FrameLayout, 326
 free, 148
 Freemarker, 269
 FROM, 180
 from, 32
 front-end developers, 3
 FULL OUTER JOIN, 182
 functions
 high-order, 347–348
 methods, 348
 Scala, 346–348
 values, 83, 346–348
 Future, 163
 Future.get, 165

G

G1. *See* Garbage First
 garbage collection, 147–149
 Garbage First (G1), 149
 gc, 153
 generate-test-sources, 304
 generations, heaps, 148
 generators, Hibernate, 276
 generics, 101–107
 E, 104
 interview algorithms, 80–83
 Java Collections API, 101–103
 List, 101–102
 nesting, 103
 performOperation, 82
 Stack, 103–105
 String, 101
 type variance, 105–106
 GenericStack, 105
 GET, 203, 208, 222, 223, 226
 REST, 230
 time(), 215
 get, 101, 292
 ArrayList, 37
 Future, 163

- getBean, 249, 251
- getClass(), 303
- getDescription, 135
- getInstance(), 63
- getIntent(), 315
- getName(), 110, 111
- getname(), 111
- getParameterMap, 207
- getResource(), 303
- getXXX, 194
- Github, 13
- Given, 144–146
- goals, Maven, 302, 308–309
- Google Docs, 4
- Google Play, 337
- Gradle, 123
- Groovy, 119, 341
- GROUP BY, 183–184
- groupId, 305
- Guava
 - BiMap, 292
 - copyOf, 294
 - immutability, 292–294
 - Iterator, 294–295
 - libraries, 290–296
 - Maven, 301
 - Multimap, 291–292
 - Multiset, 290–291
 - Set, 295–296
 - unmodifiable collection, 292–293
- GZIPOutputStream, 61

H

- Hamcrest matchers, 143
- hash table, 46
- hashCode
 - equals, 96, 97
 - HashMap, 46, 47
 - IDEs, 93
 - overriding, 97–98
- hashes. *See* maps
- HashMap, 46, 47, 118
- HashMultiset, 290
- Hashtable, 118
- hasNext, 78–80
- HEAD, 223
- heaps
 - generations, 148
 - JVM, 150–151
 - new, 147
 - stacks, 149–150
 - String, 100
 - trees, 43–45

- Hibernate, 218, 271–285
 - database
 - inserting data, 275–277
 - queries, 274–275
 - table joins, 277–280
 - dates, 280
 - eager instantiation, 282
 - generators, 276
 - lazy initialization, 282–284
 - objects, 271–277
 - performance impact, 282–284
- Hibernate query language (HQL), 281–282
- high-order functions, 347–348
- hiring manager, 7, 8
- \$HOME/.m2/repository, 304
- HQL. *See* Hibernate query language
- HTML, 203, 217–218, 287
- HTTP, 221–230
 - Apache Commons, 211
 - clients, 224–226
 - methods, 223
 - Play, 214–215
 - requests
 - HttpClient, 225
 - Java code, 216
 - Jetty, 208
 - Servlet API, 207
 - Telnet, 221–223
 - XML, 239
 - response codes, 223–224
 - REST, 226–230
 - Spring, 252
 - status codes, 226
- HttpClient, 225
- HttpServlet, 203, 209
- HttpServletRequest, 207, 267
- HttpServletResponse, 205
- URLConnection, 225, 226

I

- idempotency, 230
- IDEs. *See* integrated development environments
- if, 70
- @Ignore, 127, 137
- IllegalStateException, 114
- ImageView, 326
- immutability
 - classes, 115–116
 - Guava, 292–294
 - Scala, 349–350
- IN, 188
- increment operator, 169
- index, SQL, 187–188
- inheritance, 94

- init, 204
- in-memory databases, 198–199
- inner joins, 181–182
- InputStream, 209, 231, 267
 - Apache Commons, 211, 288
 - Decorator Pattern, 60, 61
 - OutputStream, 288–289
- INSERT, 184, 186
- insert sort algorithms, 27–28
- install, 302
- INSTANCE, 63
- instant, 297
- int, 24, 96, 108, 233
 - arrays, 36
 - compare, 26
 - Integer.MIN_VALUE, 90
 - primitive types, 88
- Integer, 80–83, 108
 - compiler, 107
 - Flyweight Pattern, 61–62
 - MySQL, 193
- Integer.MIN_VALUE, 89–90
- IntegerOperation, 81
- integers, 36
- integrated development environments (IDEs), 97
 - at-computer technical tests, 19
 - equals, 93
 - hashCode, 93
 - JUnit
 - JAR, 6, 19
 - testing, 123, 138
 - naming conventions, 111
 - technical test, 5, 6
 - XML, 236
- integration tests
 - databases, 260–261
 - Maven, 306–307
 - Spring, 259–262
 - unit tests, 138–143
- integration-test, 306
- IntegrationTest.java, 306
- IntelliJ, 19, 97
- Intent, 315–316, 324
- IntentFilter, 317
- intent-filter, 317
- Intents, 315–317
- IntentService, 322
- interfaces
 - Android, 326–332
 - Java 8, 119–120
 - marker, 233
- intern(), 100–101
- internal storage, Android, 335–336
- interning, 100–101
- interpreter, Scala, 345–346
- InterruptedException, 165
- intersection, 296
- interview algorithms
 - anagrams, 72–74
 - factorials, 71–72
 - Fibonacci sequence, 67–71
 - FizzBuzz, 65–67
 - generics, 80–83
 - implementing, 65–84
 - Integer, 80–83
 - Iterator, 78–80
 - library functionality, 72–80
 - linked lists, 75–77
 - palindrome, 77–78
 - StackOverflowException, 83
 - String, 74–75
- Interview Zen, 4, 65
- interviews
 - decision making, 8
 - technical tests, 5–7
 - telephone screening, 4–5
 - types, 3–9
- IOException, 129
- IOUtils, 288, 289
- Iterable, 120
- Iterator
 - Guava, 294–295
 - interview algorithms, 78–80
 - Map, 291

J

- JAD, 106
- JAR
 - Ant, 309–310
 - finalName, 309
 - JUnit
 - at-computer technical tests, 19
 - IDE, 6, 19
 - technical tests, 6
 - Maven, 301, 303, 306, 308, 309
 - WAR, 206
- jar, 305
- Jasper, 206
- Java 8, 119–120
 - JdbcTemplate, 259
- Java API, 18–19, 35
- Java basics, 87–121
 - annotations, 109–111
 - arrays, 98
 - autoboxing, 107–108
 - exception handling, 112–115
 - generics, 101–107
 - Java 8, 119–120
 - Java library, 115–118
 - naming conventions, 111–112
 - objects, 91–98

Java Basics (*continued*)

- primitive types, 88–90
- `String`, 98–101
- unboxing, 107–108
- Java Collections API, 48, 101–103, 117–118
- Java Database Connectivity (JDBC), 113, 191–197
 - Play, 218
 - relational databases, 191–192
 - Spring, 255–259
 - SQL queries, 192–193
 - stored procedures, 195
- Java Development Kit (JDK), 255–257
- Java library, 115–118
- Java Message Service (JMS), 255
- Java Specification Requests (JSRs), 255
- Java virtual machine (JVM), 147–157
 - arrays, 36
 - classes, 153
 - compiler, 152
 - `finalize`, 154
 - garbage collection, 147–149
 - heaps, 150–151
 - in-memory databases, 198
 - interoperability with Java, 152–157
 - memory tuning, 149–152
 - native method, 156
 - objects, 231–233
 - `OutOfMemoryError`, 49, 147
 - `Pair`, 233
 - PermGen, 134
 - Scala, 341–351
 - shutdown hooks, 156–157
 - `String`, 99
 - `Thread`, 160
 - threads, 166
 - `WeakReference`, 154–156
 - XML, 249
- `javac`, 310
- `java.lang.Object`, 96
- `java.lang.reflect.InvocationTargetException`, 138
- Javascript Object Notation (JSON), 191, 216, 240–243
- JavaServer Pages (JSPs), 206, 255
- `java.util.Date`, 280
- `java.util.HashMap`, 96
- `java.util.TreeSet`, 96
- JAXB, 234–237
- JDBC. *See* Java Database Connectivity
- `JdbcTemplate`, 218
- `JdbcTemplate`, 245, 257–259
- `JdbcUtils`, 257
- JDK. *See* Java Development Kit
- Jetty
 - HTTP requests, 208
 - JUnit, 210–211
 - Maven, 211–213

- Servlet API, 209–210
- web applications, 207–213
- `jetty:run`, 307
- JIT. *See* Just In Time
- JMS. *See* Java Message Service
- job specification page, 6
- Joda Time, 280
 - `Date`, 297
 - `DateTime`, 296–297
 - duration, 297–299
 - libraries, 296–300
 - period, 297–299
 - `SimpleTimeFormatter`, 299–300
- joins, tables
 - Hibernate databases, 277–280
 - SQL, 180–182
- JSON. *See* Javascript Object Notation
- JSON Schema, 241
- JSPs. *See* JavaServer Pages
- JSRs. *See* Java Specification Requests
- JUnit
 - `AbstractTransactionalJUnit4`
 - `SpringContextTests`, 260–261
 - annotations, 109–110
 - `Assert`, 124
 - `Assert.assertEquals()`, 101
 - code writing, 87–88
 - exceptions, 129–130
 - `@FixMethodOrder`, 262
- JAR
 - at-computer technical tests, 19
 - IDE, 6, 19
 - technical tests, 6
- Jetty, 210–211
- Maven, 301
- testing, 123–146
 - BDD, 143–146
 - best practices, 127–138
 - dependencies, 138–143
 - IDEs, 123, 138
 - life cycle, 125–127
 - mocks, 138–143
 - `@RunWith`, 132–133
 - TDD, 124
 - timeouts, 130–131
 - verification, 127–129
- junit, 305
- JUnit 4, 5–6
- JUnitCore, 124
- JUnitLifecycle, 127
- Just In Time (JIT), 153
- JVM. *See* Java virtual machine

K

key-column, 278

keySet(), 117
key-value pairs, 46, 214

L

lambdas, 119–120
language-agnostic algorithms, 4
latch.await, 132
layout_gravity, 326
layout_height, 326
layout_weight, 328
layout_width, 326
lazy initialization, 63, 282–284
Leaf, 42
left children, 39
LEFT OUTER JOIN, 181
libraries, 287–300
 Apache Commons, 287–290
 API, 20
 Guava, 290–296
 interview algorithms, 72–80
 Java, 115–118
 Joda Time, 296–300
life cycle
 Activity, 318
 Maven, 303–304
linear complexity, 65
LinearLayout, 328–329
linked lists, 37–38, 75–77
LinkedHashMap, 48, 117–118
LinkedList, 35, 37, 75
 ArrayList, 38–39
 bubble sort algorithm, 28
 Template Pattern, 58
List, 35, 37, 117
 bubble sort algorithm, 28
 generics, 101–102
 interfaces, 120
 Map, 45–46
 merge sort algorithm, 32
 polymorphism, 101
 Scala, 347–348, 350
<#list>, 268
List<int>, 108
ListIterator, 78–79
lists
 arrays, 36–39
 binary search, 32–33
 data structures, 35–39
 Integer, 80–83
 linked, 37–38
 sorted, 6, 24–32
 values, 350
ListView, 318
Location, 337
Location Services, Android, 337–339

LocationRequest, 339
locks, 167
logger.debug(), 87
Logging, 56
logging, Strategy Pattern, 55–58
logs, 215
long, 88, 130

M

main, 343
main thread, 159
malloc, 148
many-to-many relationship, 277, 280
Map, 117
 ConcurrentHashMap, 48
 HashMap, 46
 Iterator, 291
 LinkedHashMap, 48
 List, 45–46
 Set, 49
 TreeMap, 47
map, Scala, 349
mapRow, 259
maps, 45–48
mark-and-sweep algorithm, 149
marker interface, 233
match, 345
Matcher, 143
Maven
 Ant, 310–311
 application definition, 304–307
 artifacts, 307–308
 build tools, 301–309
 DBDeploy, 197
 goals, 302, 308–309
 integration tests, 306–307
 Jetty, 211–213
 JUnit testing, 123, 124–125
 life cycle, 303–304
 phases, 303, 308–309
 plug-ins, 302
 Scala, 341
 source roots, 303
 structuring project, 302–303
 unit tests, 306–307
 WAR, 206
maven-antrun-plugin, 310
maven-assembly-plugin, 308, 309
maven-compiler-plugin, 306
maven-jetty-plugin, 307
maven-release-plugin, 302
maven-surefire-plugin, 307
Memcached, 191
memoization, 71
memory leaks, 151–152

- memory tuning, JVM, 149–152
- merge, 31, 32
- merge sort algorithms, 30–32
- MessageActor, 170
- messageBody.asText(), 216
- META-INF, 263
- Metaspace, 148
- methods
 - definitions, at-computer technical tests, 19
 - functions, 348
 - HTTP, 223
 - names, 111
 - API library, 20
 - JUnit 4, 5
 - native, 156, 160
 - recursive, 149
 - Scala, 348
 - stacks, 149
 - static, 94
- Mockito, 56, 141, 143
- mockito-core, 305
- mocks, 138–143
- Model, 266
- Model-View-Controller (MVC)
 - HTML, 203
 - Spring, 203, 262–269
 - application context, 264
 - JdbcTemplate, 245
 - REST, 226–227, 266
 - server-side logic, 267–268
 - String, 266–267
 - technical test, 5
 - web controllers, 264–267
- MongoDB, 191
- MultiMap, 49
- MultiMap, 291–292
- MultiSet, 49
- Multiset, 290–291
- MVC. *See* Model-View-Controller
- mvn test, 124–125
- mvnf compiler:compile, 302
- MySQL, 177, 188, 189, 193, 198–199
- mysql>, 177

N

- n
 - algorithms, 23
 - Fibonacci sequence, 68
 - linear complexity, 65
- name: Type, 342
- name/value pairs, 333
- naming conventions, 111–112
- n-ary trees, 44
- native method, 156, 160

- natural ordering, 24, 25, 43
- negative zero, 90
- nesting, 103
- new, 62, 147, 170
- next, 76
- next sibling, 44
- Nexus, 301
- NoClassDefFoundError, 112
- Node, 42
- <none>, 93
- NoSQL, 191
- NoSuchBeanDefinitionException, 254
- NULL, 181, 184, 187
- null, 63, 76, 156
 - checkString, 130
 - JUnit testing, 128
 - LinkedList, 75
 - NullPointerException, 108
 - objects, 91
 - primitive types, 108
 - WeakReference, 154, 155
- Null Object Pattern, 41, 62
- null objects, trees, 41–42
- NullPointerException, 91, 107–108

O

- O(n), 32, 71
- O(n log n), 30, 32
- O(n²), 23, 27
 - bubble sort algorithm, 28
 - quadratic, 65
 - quicksort algorithm, 30
- Object, 46, 154
 - equals, 96, 127
 - get, 101
 - Model, 266
- object, 343
- ObjectInputStream, 231
- ObjectOutputStream, 61, 231–233
- Object/Relational Mapping (ORM), 271
- objects
 - arrays, 98
 - Eden Space, 148
 - final, 92
 - hashCode, 96
 - Hibernate, 271–277
 - HQL, 282
 - Java basics, 91–98
 - JAXB, 236–237
 - JSON, 242–243
 - JVM, 231–233
 - locks, 167
 - null, 41–42
 - null, 91

- polymorphism, 94–95
 - reading and writing, 231–234
 - reference types, 87, 91
 - serialization, 231–234
 - variables, 91
 - visibility modifiers, 92–94
 - XML, 238–239
 - Objects, 24–25, 104
 - Odersky, Martin, 342, 351
 - onAttach(), 332
 - onCreate(), 318, 337
 - onDestroy(), 318
 - onLocationChange(), 337
 - onPause(), 318, 320
 - onReceive, 170, 172, 321
 - onResume(), 318, 320
 - onSaveInstanceState(), 318, 321
 - onStart(), 318, 337
 - onStartCommand(), 323
 - onStop(), 318, 337
 - originalList, 28
 - ORM. *See* Object/Relational Mapping
 - OUT, 188
 - outer joins, 181–182
 - OutOfMemoryError, 49, 112, 147
 - OutputStream, 209, 231
 - Apache Commons, 288–290
 - copy, 288
 - Decorator Pattern, 60, 61
 - InputStream, 288–289
 - Spring MVC, 267
 - @Override, 110–111
- ## P
- package, 303, 306, 308
 - package-jar, 310
 - packaging, 308
 - packing, 206
 - Pair, 233
 - pair programming, 7
 - palindrome, 77–78
 - Parameterized, 132–133, 135
 - parameterized types, 101
 - @Parameterized.Parameters, 133
 - parameters
 - queries, 193–195
 - Scala, 342
 - Servlet API, 206–207
 - parameters(), 133
 - paramPrintout.scala.html, 218
 - @PathVariable, 266
 - pattern matching, 344–345
 - peek, 104, 155
 - peekedValue, 156
 - performOperation, 82
 - period, Joda Time, 297–299
 - Permanent Generation (PermGen), 100, 134, 148
 - persistent data
 - Android, 323–324, 333–336
 - Service, 323–324
 - SQL, 184–185
 - Person, 96–97
 - phases, Maven, 303, 308–309
 - PiCalculator, 162
 - pivot, 29
 - Plain Old Java Objects (POJOs), 245, 264, 271
 - Play, 351
 - HTML, 217–218
 - JDBC, 218
 - web applications, 213–218
 - play, 213–214
 - play.mvc.Controller, 216
 - plugin, 239
 - POJOs. *See* Plain Old Java Objects
 - polymorphism
 - arrays, 98
 - List, 101
 - objects, 94–95
 - Rectangle, 95
 - Square, 95
 - POM. *See* Project Object Model
 - pom.xml, 303, 309
 - pop, 104, 152
 - positive numbers, 89–90
 - POST, 203, 208, 209, 223
 - Play, 216
 - REST, 230
 - post-integration-test, 304, 309
 - pre-integration-test, 304
 - prepareCall, 195
 - PreparedStatement, 194, 256
 - primitive types
 - Atomic, 168
 - compiler, 89
 - downcasting, 89
 - Java basics, 88–90
 - null, 108
 - NullPointerException, 107–108
 - variables, 89
 - PRIORITY_BALANCED_POWER_ACCURACY, 340
 - PRIORITY_HIGH_ACCURACY, 339
 - PRIORITY_LOW_POWER, 340
 - PRIORITY_NO_POWER, 340
 - private, 92–94
 - probationary period, 8
 - product manager, 7
 - project, 215
 - <project>, 307
 - project manager, 7

Project Object Model (POM), Maven, 211, 212, 239, 303, 305
 property, 248
 PropertyPlaceholderConfigurer, 252–253
 protected, 93
 prototype scope, 251
 provided, 306
 pseudocode, 20
 anagrams, 72
 merge sort algorithm, 31
 public, 93, 215
 public voice run(), 160
 push, 104, 152
 PUT, 203, 208, 209, 223, 226
 Play, 216
 REST, 230

Q

QA testing, 307
 quadratic, 65
 @Qualifier, 254
 queries
 Hibernate, 274–275
 parameters, 193–195
 SQL, 187–188
 JDBC, 192–193
 queryForInt, 258
 questions
 at-computer technical tests, 19
 face-to-face interviews, 7, 21
 Stack Overflow, 13
 technical tests, 5, 18
 telephone screening, 4, 5
 Queue, 39, 117
 quicksort algorithm, 29–30

R

race condition, 228
 random access, 36
 RandomizedRunner, 137
 readData, 233
 Reader, 289
 readObject, 233
 RecipeBackupService, 323
 recipeId, 315–316
 RecipeListFragment, 331–332
 recruitment within, 3
 Rectangle, 95
 recursive algorithms, 30
 factorials, 71
 Fibonacci sequence, 69
 linked lists, 76
 recursive class definition, LinkedList, 75

recursive data type, trees, 40
 recursive methods, 149
 Redis, 191
 reference types, 88, 91, 108, 168
 referential equality, 343
 referential integrity, 181
 referral bonuses, 14
 regression, 123
 relational databases
 Android, 333–335
 JDBC, 191–192
 SQL, 177–178
 RelativeLayout, 327–328
 releases, 308
 remote live-coding exercise, 4
 remove, 152
 removeAll, 295
 replace, 100
 repository, 307
 Representational State Transfer (REST), 226–230, 266
 request, 205
 @RequestMapping, 266
 requiredParameter, 205
 resources
 Android apps, 320–321
 JVM, 249
 REST, 226
 response, 205
 REST. *See* Representational State Transfer
 Result, 216
 ResultsController, 266
 ResultSet, 193, 256, 258
 ResultSet.getMetaData, 259
 resume, 4
 acronyms, 12
 developer profile, 12
 education, 12
 experience, 12–13
 key skills, 12
 length, 11
 personal interests, 13–14
 photography, 11–12
 qualifications, 12
 references, 14
 tailoring to particular job, 11
 technical tests, 17
 writing, 11–15
 retainAll, 295
 return, 70, 342
 reverse, 74–75, 78
 reverse data types, 37
 ReverseNumericalOrder, 26
 right children, 39
 RIGHT OUTER JOIN, 181
 ROLLBACK, 191
 routers, 173

- routes, 214–215
- RowMapper, 259
- run, 137, 160, 214
- Runnable, 160, 161, 163, 165
- Runner, 132, 134–137
- RunNotifier, 137, 138
- runtime, exceptions, 113
- RuntimeException, 112, 113, 129
- @RunWith, 132–133, 145

S

- salary, 7, 8
- SBT. *See* Simple Build Tool
- Scala, 83, 341–351
 - case class, 344
 - classes, 342–343
 - compiler, 343
 - functions, 346–348
 - immutability, 349–350
 - interpreter, 345–346
 - lambdas, 119
 - methods, 348
 - parameters, 342
 - pattern matching, 344–345
 - Play, 213, 218
 - tail recursion, 350
 - values, 350
 - variables, 342
- scope, 149
 - Maven, 306
 - Spring, 251–252
- SELECT, 178, 179, 186, 188
- sendError, 205
- Serializable, 233–234
- serialization, 231–243
 - JSON, 240–243
 - objects, 231–234
 - XML, 234–240
- server.hostname, 253
- ServerSetup, 253
- server-side logic, Spring MVC, 267–268
- Service
 - Activity, 324–325
 - Android, 314, 322–325
 - Intent, 324
 - persistent data, 323–324
- Servlet API
 - HTTP requests, 207
 - Jetty, 209–210
 - Maven, 306
 - parameters, 206–207
 - technical test, 5
 - Tomcat, 201–207
 - web applications, 201–207
- servlet container, 201
- ServletResponse, 205, 267
- servlets, Spring MVC, 263
- Set, 48, 49, 96–97, 295–296
- sets, 48–49
- setUp, 109
- SetView, 296
- setXXX, 194
- shared state, threads, 165–167
- SharedPreferences, 333
- ShareIntentActivityReceiver, 317
- Short, 47
- short, 88
- shutdown hooks, 156–157
- Simple Build Tool (SBT), 123, 351
- SimpleDateFormat, 299
- SimpleJdbcTemplate, 262
- SimpleTimeFormatter, 299–300
- SimpleTrees, 40
- singleThreadedExecutor, 163
- Singleton, 63
- singleton classes, 343
- Singleton Pattern, 63–64
- singleton scope, 251
- size, 48, 292
- size(), 117
- Slick, 351
- snapshot build, 307
- snapshotRepository, 307
- SocketOutputStream, 60
- sort, 24, 25–26
- sorted lists, 6, 24–32
- sorting algorithm, 73
- source code, XSD, 239–240
- source roots, Maven, 303
- space complexity, 23–24
- split, 100
- Spring, 245–269
 - application context, 245–255
 - autowiring, 253–255
 - classes, 248
 - dependency injection, 245–250
 - finally, 256–257
 - integration tests, 259–262
 - JDBC, 255–259
 - boilerplate code, 257–259
 - JdbcTemplate, 218
 - MVC, 203, 262–269
 - application context, 264
 - JdbcTemplate, 245
 - REST, 226–227, 266
 - server-side logic, 267–268
 - String, 266–267
 - technical test, 5
 - web controllers, 264–267

Spring (*continued*)

- scope, 251–252
 - template engines, 269
 - XML, 58, 247–248
- SQL, 177–191
- index, 187–188
 - in-memory databases, 198–199
 - persistent data, 184–185
 - queries
 - JDBC, 192–193
 - speeding inefficient, 187–188
 - relational databases, 177–178
 - statements, case sensitivity, 178
 - stored procedures, 188–189
 - table joins, 180–182
 - transactions, 190–191
 - views, 185–186
- SQLException, 256
- SQLiteCursor, 334
- SQLiteDatabase, 334
- SQLiteOpenHelper, 333–334
- Square, 95
- square, 347
- src/main/webapp/WEB-INF, 302
- Stack, 58, 60, 103–105
- Stack Overflow, 13
- stack space, 149–150
- stack trace, 149
- StackOverflowError, 149, 350
- StackOverflowException, 34, 83
- StackPredicate, 58–60
- stacks, heaps, 149–150
- stackValues, 152
- standalone-application-with-dependencies, 309
- start, 215
- START TRANSACTION, 190
- start0, 160
- startPeriodicUpdates(), 337
- START_REDELIVER_INTENT, 323
- Statement, 193
- static, 62, 94
- steps, Cucumber, 144–146
- stop-the-world, 149, 153
- stored procedures
 - JDBC, 195
 - MySQL, 189
 - SQL, 188–189
- Strategy Pattern, 55–58
- String
 - Apache Commons, 288
 - arrays, 36
 - assertThat, 143
 - char, 99
 - checkString, 130
 - Comparable, 25
 - constant pool, 101
 - FileDictionary, 248
 - generics, 101
 - InputStream, 289
 - Intent, 315
 - intern(), 100–101
 - interning, 100–101
 - interview algorithms, 74–75
 - Java basics, 98–101
 - JUnit, 127–128
 - JVM, 99
 - Model, 266
 - palindromes, 78
 - performOperation, 82
 - PermGen, 188
 - reverse, 78
 - Spring MVC, 266–267
 - toWord, 67
 - values, 99–100
- StringBuilder, 74–75
- StringEscapeUtils, 287
- stringLength, 342
- subList, 32
- substring, 100
- SUM, 183
- SupervisorStrategy, 172
- Survivor Space, 148
- switch, 344
- syntactic sugar, 134
- syntax, technical interviews, 20
- System, 36, 153
- system, 306
- system tests, 144
- System.gc, 153
- System.out.println(), 87

T

tables

- constraints, 186–187
 - DDL, 186
 - foreign keys, 186–187
 - hash, 46
 - joins
 - Hibernate databases, 277–280
 - SQL, 180–182
- tail recursion, 350
- @tailrec, 350
- target, 303
- targets, Ant, 309
- tasks, Ant, 309
- TDD. *See* Test-Driven Development
- tearDown, 109
- technical interviews, 7
- technical test
 - at-computer, 18–19

- company information, 6–7
 - face-to-face interviews, 19–21
 - IDE, 5, 6
 - interviews, 5–7
 - JUnit 4, 5–6
 - questions, 5, 18
 - Servlet API, 5
 - sorted lists, 6
 - Spring MVC, 5
 - Tomcat, 5
 - written, 17–18
 - tee, 289
 - TeeOutputStream, 289–290
 - telephone screening
 - company information, 4–5
 - hands-free kit, 4
 - interview, 4–5
 - language-agnostic algorithms, 4
 - notes, 4
 - questions, 4, 5
 - remote live-coding exercise, 4
 - Telnet, 221–223
 - template engines, Spring, 269
 - Template Pattern, 58–60
 - Tenured Generation, 148
 - test, 109
 - Maven, 302, 303, 305
 - @Test
 - Cucumber, 145
 - expected, 130
 - IOException, 129
 - JUnit testing, 125–126
 - JUnitLifecycle, 127
 - long, 130
 - Runner, 136–137
 - Spring, 261
 - testClass, 136
 - testClass, 136
 - Test-Driven Development (TDD), 124
 - testing. *See also* integration tests; JUnit; technical test;
 - unit tests
 - in-memory databases, 198–199
 - QA, 307
 - system, 144
 - TestNG, 301
 - TestSuite, 125
 - Then, 144–146
 - Thread, 160, 161–163
 - thread pool, 161
 - ThreadPrinter, 160
 - threads
 - concurrency, 159–165
 - Counter, 166
 - JVM, 166
 - Runnable, 161
 - shared state, 165–167
 - try/catch, 160
 - Thread.sleep, 160
 - Throwable, 129
 - throwable, 114
 - Throwable.Throwable, 112
 - time(), 215
 - time complexity, algorithms, 23–24
 - timeouts, 130–131
 - toDate, 297
 - to:from, 32
 - Tomcat
 - Servlet API, 201–207
 - technical test, 5
 - web applications, 201–207
 - toString(), 98
 - toWord, 67
 - transactions, SQL, 190–191
 - transient, 233–234
 - TreeMap, 47
 - TreeMultiset, 290
 - trees
 - balanced, 43
 - data structures, 39–45
 - heaps, 43–45
 - n-ary, 44
 - null objects, 41–42
 - recursive data type, 40
 - unbalanced, 42
 - triggers, 189
 - try, 115
 - try/catch, 160
 - try/catch/finally, 115
 - try-with-resources, 115
 - Twitter, 13
 - Two's Complement, 89–90
- ## U
- unbalanced tree, 42
 - unboxing, 107–108
 - Unicode, 89
 - union, 296
 - unit tests
 - integration tests, 138–143
 - Maven, 306–307
 - ResultsController, 266
 - UPDATE, 184, 185, 186
 - updateList, 81–82
 - User Acceptance Testing, 307
- ## V
- val, 342
 - validate, 303

- ValueContainer, 155
- valueOf, 62
- values
 - absolute, 90
 - average-case, 23
 - best-case, 23
 - CSV, 268
 - functions, 83, 346–348
 - key-value pairs, 46, 214
 - lists, 350
 - name/value pairs, 333
 - Scala, 350
 - String, 99–100
 - worst-case, 23, 28, 30
- values, 117
- var, 342
- VARCHAR, 193
- variables
 - naming conventions, 111
 - objects, 91
 - primitive types, 89
 - private, 92–94
 - Scala, 342
 - static, 94
- Velocity, 269
- verbs, 223
- version, 305
- View, 326–328
- ViewResolver, 264, 269
- views, 214
- views, SQL, 185–186
- visibility modifiers, 92–94
- void, 125

W

- WAR. *See* Web Archive
- war, 206
- WeakReference, 154–156
- web applications, 201–219
 - classes, 207–208
 - Jetty, 207–213
 - Play, 213–218

- Servlet API, 201–207
 - Tomcat, 201–207
- Web Archive (WAR), 205–206
 - Maven, 301, 303
 - Play, 215
 - Tomcat, 206
- web controllers, Spring MVC, 264–267
- webapps, 206
- web.xml, 202, 206, 263
- weightSum, 328
- When, 144–146
- WHERE, 178, 179, 180, 185
- while, 31
- wildcards, 106
- willingness to learn, 6
- worst-case value
 - algorithms, 23
 - bubble sort algorithm, 28
 - quicksort algorithm, 30
- write, 60
- writeOutput, 342
- Writer, 289
- written technical tests, 17–18

X

- x++, 169
- x--, 169
- XML
 - autowiring, 253–254
 - IDEs, 236
 - JVM, 249
 - objects, 238–239
 - serialization, 234–240
 - Spring, 58, 247–248, 268
- XML Schema Definition (XSD), 234–240
 - source code, 239–240
- Xmx, 151
- XSD. *See* XML Schema Definition
- XX:MaxPermSize, 151
- XX:Permsize, 151