



Robert C. Martin Series

Agile Java™

Crafting Code
with Test-Driven
Development

```
private Object monitor = new Object()

public void testClock() throws Except
    final int seconds = 2;
    final List<Date> tics = new Arra
ClockListener listener = new Clo
    private int count = 0;
    public void update(Date dat
        tics.add(date);
        if (++count == seconds)
            synchronized(moni
                monitor.notifyAll();
            }
        }
    );
clock = new Clock(listener);
synchronized(monitor) {
    monitor.wait();
}
clock.step();
verify(tics, seconds);
}
```

Foreword by Ron Jeffries

Jeff Langr

Table of Contents

Copyright.....	1
Praise for Agile Java.....	2
Robert C. Martin Series.....	5
About the Author.....	20
Foreword.....	22
Acknowledgments.....	24
Introduction.....	26
An Agile Overview.....	34
Setting Up.....	46
Lesson 1. Getting Started.....	56
Testing.....	56
Design.....	57
A Simple Test.....	58
JUnit.....	60
Adding a Test.....	62
Creating a Student.....	64
Creating the Student Class.....	65
Constructors.....	66
Local Variables.....	67
Returning a Value from a Method.....	69
Assertions.....	71
Instance Variables.....	74
Summarizing the Test.....	77
Refactoring.....	78
this.....	80
private.....	82
Naming Conventions.....	84
Whitespace.....	86
Exercises.....	87
Lesson 2. Java Basics.....	88
CourseSession.....	88
Enrolling Students.....	90
int.....	90
Initialization.....	93
Default Constructors.....	94
Suites.....	95
The SDK and java.util.ArrayList.....	96
Adding Objects.....	98
Incremental Refactoring.....	100
Objects in Memory.....	101
Packages and the import Statement.....	103
The java.lang Package.....	104
The Default Package and the package Statement.....	104
The setUp Method.....	106
More Refactoring.....	108
Class Constants.....	109
Dates.....	111
Overloaded Constructors.....	116
Deprecation Warnings.....	116
Refactoring.....	117
Creating Dates with Calendar.....	120
Comments.....	121
Javadoc Comments.....	122
Exercises.....	125
Lesson 3. Strings and Packages.....	128
Characters and Strings.....	128
Strings.....	131
StringBuilder.....	132
System Properties.....	134
Looping through All Students.....	135
Single-Responsibility Principle.....	137
Refactoring.....	140
System.out.....	143
Using System.out.....	144
Refactoring.....	145

Package Structure.....	146
Access Modifiers.....	147
Using Ant.....	153
Exercises.....	156

Lesson 4. Class Methods and Fields..... **158**

Class Methods.....	158
Class Variables.....	162
Operating on Class Variables with Class Methods.....	163
Static Import.....	166
Incrementing.....	169
Factory Methods.....	170
Simple Design.....	172
Static Dangers.....	172
Using Statics: Various Notes.....	173
Jeff's Rule of Statics.....	174
Booleans.....	175
Tests as Documentation.....	179
More on Initialization.....	182
Exceptions.....	183
Revisiting Primitive-Type Field Initialization.....	184
Exercises.....	185

Lesson 5. Interfaces and Polymorphism..... **188**

Sorting: Preparation.....	188
Sorting: Collections.sort.....	189
CourseReportTest.....	190
Interfaces.....	191
Why Interfaces.....	193
Implementing Comparable.....	194
Sorting on Department and Number.....	196
The if Statement.....	197
Grading Students.....	198
Floating-Point Numbers.....	198
Testing Grades.....	200
Refactoring.....	203
Enums.....	204
Polymorphism.....	206
Using Interface References.....	212
ArrayList and the List Interface.....	214
Exercises.....	214

Lesson 6. Inheritance..... **220**

The switch Statement.....	220
Case Labels Are Just Labels.....	221
Maps.....	224
Inheritance.....	226
Abstract Classes.....	229
Extending Methods.....	231
Refactoring.....	232
Enhancing the Grade Enum.....	234
Summer Course Sessions.....	235
Calling Superclass Constructors.....	236
Refactoring.....	240
More on Constructors.....	243
Inheritance and Polymorphism.....	245
The Principle of Subcontracting.....	245
Exercises.....	253

Lesson 7. Legacy Elements..... **256**

Looping Constructs.....	257
Breaking Up a Student's Name.....	257
Comparing Java Loops.....	265
Refactoring.....	266
Looping Control Statements.....	267
The Ternary Operator.....	270
Legacy Collections.....	271
Iterators.....	272
Iterators and the for-each Loop.....	274
Casting.....	275
Wrapper Classes.....	277
Arrays.....	280
Refactoring.....	289
Exercises.....	291

Lesson 8. Exceptions and Logging..... **294**

Exceptions.....	295
Dealing with Exceptions.....	297
Checked Exceptions.....	298
Exception Hierarchy.....	300
Creating Your Own Exception Type.....	301
Checked Exceptions vs. Unchecked Exceptions.....	303
Messages.....	304
Catching Multiple Exceptions.....	305
Rethrowing Exceptions.....	307
Stack Traces.....	310
The finally Block.....	310
Refactoring.....	312
Logging.....	315
Logging in Java.....	316
Testing Logging.....	319
Logging to Files.....	323
Testing Philosophy for Logging.....	325
More on FileHandler.....	327
Logging Levels.....	327
Logging Hierarchies.....	329
Additional Notes on Logging.....	330
Exercises.....	330
Lesson 9. Maps and Equality.....	336
Logical Operators.....	336
Short-Circuiting.....	338
Hash Tables.....	338
Courses.....	340
Refactoring Session.....	342
Equality.....	348
The Contract for Equality.....	351
Apples and Oranges.....	352
Collections and Equality.....	354
Hash Tables.....	355
Collisions.....	357
An Ideal Hash Algorithm.....	358
A Final Note on hashCode.....	360
More on Using HashMaps.....	362
Additional Hash Table and Set Implementations.....	368
toString.....	368
Strings and Equality.....	370
Exercises.....	371
Lesson 10. Mathematics.....	374
BigDecimal.....	375
More on Primitive Numerics.....	378
Integer Math.....	379
Numeric Casting.....	380
Expression Evaluation Order.....	381
NaN.....	382
Infinity.....	383
Numeric Overflow.....	384
Bit Manipulation.....	385
java.lang.Math.....	393
Numeric Wrapper Classes.....	395
Random Numbers.....	396
Exercises.....	400
Lesson 11. IO.....	404
Organization.....	404
Character Streams.....	405
Writing to a File.....	410
java.io.File.....	413
Byte Streams and Conversion	414
A Student User Interface.....	415
Testing the Application.....	418
Data Streams.....	420
CourseCatalog.....	421
Advanced Streams.....	424
Object Streams.....	424
Random Access Files.....	432
The Student Directory.....	433
sis.db.DataFileTest.....	435
Static Nested Classes and Inner Classes.....	437
sis.db.DataFile.....	438

sis.db.KeyFileTest.....	441
sis.db.KeyFile.....	443
sis.util.IOUtilTest.....	444
sis.util.IOUtil.....	445
sis.util.TestUtil.....	446
Developing the Solution.....	447
Exercises.....	447

Lesson 12. Reflection and Other Advanced Topics..... 450

Mock Objects Revisited.....	450
The Jim Bob ACH Interface.....	452
The Mock Class.....	454
The Account Class Implementation.....	456
Anonymous Inner Classes.....	458
Adapters.....	460
Accessing Variables from the Enclosing Class.....	462
Tradeoffs.....	464
Reflection.....	465
Using JUnit Code.....	466
The Class Class.....	467
Building the Suite.....	469
Class Modifiers.....	471
Dynamic Proxy.....	473
A Secure Account Class.....	474
Building the Secure Account Solution.....	476
The SecureProxy Class.....	481
Problems With Reflection.....	484
Exercises.....	484

Lesson 13. Multithreading..... 487

Multithreading.....	487
Search Server.....	487
The Search Class.....	488
Less Dependent Testing.....	491
The Server.....	495
Waiting in the Test.....	497
Creating and Running Threads.....	498
Cooperative and Preemptive Multitasking.....	502
Synchronization.....	503
Creating Threads with Runnable.....	505
Synchronized.....	506
Synchronized Collections.....	507
BlockingQueue.....	508
Stopping Threads.....	510
Wait/Notify.....	511
Additional Notes on wait and notify.....	516
Locks and Conditions.....	517
Thread Priorities.....	519
Deadlocks.....	520
ThreadLocal.....	520
The Timer Class.....	524
Thread Miscellany.....	526
Summary: Basic Design Principles for Synchronization.....	531
Exercises.....	531

Lesson 14. Generics..... 534

Parameterized Types.....	534
Collection Framework.....	535
Multiple Type Parameters.....	536
Creating Parameterized Types.....	536
Erasure.....	539
Upper Bounds.....	541
Wildcards.....	543
Implications of Using Wildcards.....	545
Generic Methods.....	547
Wildcard Capture.....	547
Super.....	548
Additional Bounds.....	550
Raw Types.....	551
Checked Collections.....	553
Arrays.....	555
Additional Limitations.....	556
Reflection.....	557
Final Notes.....	557
Exercises.....	558

Lesson 15. Assertions and Annotations.....	560
Assertions.....	560
The assert Statement vs. JUnit Assert Methods.....	562
Annotations.....	562
Building a Testing Tool.....	563
TestRunnerTest.....	564
TestRunner.....	565
The @TestMethod Annotation.....	568
Retention.....	570
Annotation Targets.....	571
Skipping Test Methods.....	573
Modifying TestRunner.....	573
Single-Value Annotations.....	575
A TestRunner User Interface Class.....	577
Array Parameters.....	578
Multiple Parameter Annotations.....	580
Default Values.....	582
Additional Return Types and Complex Annotation Types.....	583
Package Annotations.....	584
Compatibility Considerations.....	586
Additional Notes on Annotations.....	586
Summary.....	587
Exercises.....	587
Additional Lesson I. Swing, Part 1.....	590
Swing.....	591
Getting Started.....	592
Swing Application Design.....	596
Panels.....	597
Refactoring.....	601
More Widgets.....	605
Refactoring.....	608
Button Clicks and ActionListeners.....	611
List Models.....	613
The Application.....	616
Layout.....	619
GridBagLayout.....	627
Moving Forward.....	631
Additional Lesson II. Swing, Part 2.....	632
Miscellaneous Aesthetics.....	633
Feel.....	637
Tables.....	660
Feedback.....	665
Responsiveness.....	672
Remaining Tasks.....	675
Final Notes.....	676
Additional Lesson III. Java Miscellany.....	678
JARs.....	678
Regular Expressions.....	681
Cloning and Covariance.....	687
JDBC.....	689
Internationalization.....	698
Call by Reference versus Call by Value.....	706
Java Periphery.....	708
What Else Is There?.....	718
Appendix A: An Agile Java Glossary.....	726
.....	726
Appendix B: Java Operator Precedence Rules.....	726
Appendix C: Getting Started with IDEA.....	727
IDEA.....	731
The Hello Project.....	735
Running Tests.....	735
Taking Advantage of IDEA.....	738
Agile Java References.....	758
bvdindexIndex.....	760



Robert C. Martin Series

Agile Java™

Crafting Code
with Test-Driven
Development

```
private Object monitor = new Object()

public void testClock() throws Exception {
    final int seconds = 2;
    final List<Date> ticks = new ArrayList();
    ClockListener listener = new ClockListener() {
        private int count = 0;
        public void update(Date date) {
            ticks.add(date);
            if (++count == seconds)
                synchronized(monitor) {
                    monitor.notifyAll();
                }
        }
    };
    clock = new Clock(listener);
    synchronized(monitor) {
        monitor.wait();
    }
    clock.stop();
    verify(ticks, seconds);
}
```

Foreword by Ron Jeffries

Jeff Langr

Praise for Agile Java

“The most practical, productive, and enjoyable book on programming I’ve read. Results Driven.”

Steve Bartolin, President and CEO, The Broadmoor

“At long last a book that integrates solid beginning lessons in both Java and Test-Driven Development! I know most of the software development apprentices that go through my facility could have avoided the pain of unlearning years of ad hoc programming habits if Jeff’s book had been available when they first learned Java.”

Steven A. Gordon, Ph.D., Manager,
Arizona State University Software Factory (<http://sf.asu.edu/>)

“Jeff draws on his exceptionally deep expertise in Java and Agile processes to show us a remarkably powerful approach for building Java programs that are clean, well-structured, and thoroughly tested. Highly recommended! ”

Paul Hodgetts, Founder and CEO, Agile Logic, Inc.

“A great way to learn Java. *Agile Java* takes you through not only the basics of the Java language, setup, and tools, but also provides excellent instruction on concepts of test-driven development, refactoring, and Object-Oriented programming. This book is required reading for our engineering team. ”

Andrew Masters, President and CIO, 5two8, Inc.

“*Agile Java* is a must-read for anyone developing Java applications. The text contains innovative and insightful ideas on how to write high-quality, extensible Java applications faster and more efficiently than traditional methods. Beginner and experienced developers will learn a tremendous amount from the concepts taught in this book. ”

Bret McInnis, Vice President eBusiness Technologies, Corporate Express

This page intentionally left blank

Agile Java

Robert C. Martin Series

The mission of this series is to improve the state of the art of software craftsmanship. The books in this series are technical, pragmatic, and substantial. The authors are highly experienced craftsmen and professionals dedicated to writing about what actually works in practice, as opposed to what might work in theory. You will read about what the author has done, not what he thinks you should do. If the book is about programming, there will be lots of code. If the book is about managing, there will be lots of case studies from real projects.

These are the books that all serious practitioners will have on their bookshelves. These are the books that will be remembered for making a difference and for guiding professionals to become true craftsman.

Managing Agile Projects

Sanjiv Augustine

Agile Estimating and Planning

Mike Cohn

Working Effectively with Legacy Code

Michael C. Feathers

Agile Java™: Crafting Code with Test-Driven Development

Jeff Langr

Agile Principles, Patterns, and Practices in C#

Robert C. Martin and Micah Martin

Agile Software Development: Principles, Patterns, and Practices

Robert C. Martin

UML For Java™ Programmers

Robert C. Martin

Fit for Developing Software: Framework for Integrated Tests

Rick Mugridge and Ward Cunningham

Agile Software Development with SCRUM

Ken Schwaber and Mike Beedle

Extreme Software Engineering: A Hands on Approach

Daniel H. Steinberg and Daniel W. Palmer

For more information, visit <http://www.prenhallprofessional.com/martinseries>

Agile Java

Crafting Code with Test-Driven Development

Jeff Langr



Prentice Hall Professional Technical Reference

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

Unix is a registered trademark of The Open Group. Windows is a registered trademark of Microsoft Corporation.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U. S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the U. S., please contact:

International Sales
international@pearsoned.com

Visit us on the Web: www.phptr.com

Library of Congress Cataloging Number: 2004114916

Copyright © 2005 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671 3447

ISBN 10: 0-13-148239-4

ISBN 13: 978-0-13-148239-5

Text printed in the United States on recycled paper at Phoenix Book Tech.
Third printing, October 2007

In memory of Edmund Langr and Heather Mooney

This page intentionally left blank

Contents

About the Authorxix
Forewordxxi
Acknowledgmentsxxiii
Introduction1
Who Is This Book For?3
What This Book Is Not4
How to Use This Book5
Exercises6
Conventions Used in This Book6
An Agile Overview9
What Is “Agile?”9
What Is Java?11
Why OO?13
What Is an Object?13
What Are Classes?14
Why UML?16
What Is Inheritance?17
Why Test-Driven Development?18
Setting Up21
Software You’ll Need21
Does It Work?26
Compiling Hello World27
Executing Hello World27
Still Stuck?28
Lesson 1: Getting Started31
Testing31
Design32

A Simple Test33
JUnit35
Adding a Test37
Creating a Student39
Creating the Student Class40
Constructors41
Local Variables42
Returning a Value from a Method44
Assertions46
Instance Variables49
Summarizing the Test52
Refactoring53
this55
private57
Naming Conventions59
Whitespace61
Exercises62
Lesson 2: Java Basics63
CourseSession63
Enrolling Students65
int65
Initialization68
Default Constructors69
Suites70
The SDK and java.util.ArrayList71
Adding Objects73
Incremental Refactoring75
Objects in Memory76
Packages and the import Statement78
The java.lang Package79
The Default Package and the package Statement79
The setup Method81
More Refactoring83
Class Constants84
Dates86
Overload Constructors86
Deprecation Warnings91

Refactoring	92
Creating Dates with Calendar	95
Comments	96
Javadoc Comments	97
Exercises	100
Lesson 3: Strings and Packages	103
Characters and Strings	103
Strings	106
StringBuilder	107
System Properties	109
Looping Through All Students	110
Single-Responsibility Principle	112
Refactoring	115
System.out	118
Using System.out	119
Refactoring	120
Package Structure	121
Access Modifiers	122
Using Ant	128
Exercises	131
Lesson 4: Class Methods and Fields	133
Class Methods	133
Class Variables	137
Operating on Class Variables with Class Methods	138
Static Import	141
Incrementing	144
Factory Methods	145
Simple Design	147
Static Dangers	147
Using Statics: Various Notes	148
Jeff's Rule of Statics	149
Booleans	150
Tests as Documentation	154
More on Initialization	157
Exceptions	158
Revisiting Primitive-Type Field Initialization	159
Exercises	160

Lesson 5: Interfaces and Polymorphism	163
Sorting: Preparation	163
Sorting: Collections.sort	164
CourseReportTest	165
Interfaces	166
Why Interfaces	168
Implementing Comparable	169
Sorting on Department and Number	171
The if Statement	172
Grading Students	173
Floating-Point Numbers	173
Testing Grades	175
Refactoring	178
Enums	179
Polymorphism	181
Using Interface References	187
ArrayList and the List Interface	189
Exercises	189
Lesson 6: Inheritance	195
The switch Statement	195
Case Labels Are Just Labels	196
Maps	199
Inheritance	201
Abstract Classes	204
Extending Methods	206
Refactoring	207
Enhancing the Grade Enum	209
Summer Course Sessions	210
Calling Superclass Constructors	211
Refactoring	215
More on Constructors	218
Inheritance and Polymorphism	220
The Principle of Subcontracting	220
Exercises	228
Lesson 7: Legacy Elements	231
Looping Constructs	232
Breaking Up a Student's Name	232

The while Loop	234
Comparing Java Loops	240
Refactoring	241
Looping Control Statements	242
The Ternary Operator	245
Legacy Collections	246
Iterators	247
Iterators and the for-each Loop	249
Casting	250
Wrapper Classes	252
Arrays	255
Refactoring	264
Exercises	266
Lesson 8: Exceptions and Logging	269
Exceptions	270
Dealing With Exceptions	272
Checked Exceptions	273
Exception Hierarchy	275
Creating Your Own Exception Type	276
Checked Exceptions vs. Unchecked Exceptions	278
Messages	279
Catching Multiple Exceptions	280
Rethrowing Exceptions	282
Stack Traces	285
The finally Block	285
Refactoring	287
Logging	290
Logging in Java	291
Testing Logging	294
Logging to Files	298
Testing Philosophy for Logging	300
More on FileHandler	302
Logging Levels	302
Logging Hierarchies	304
Additional Notes on Logging	305
Exercises	305
Lesson 9: Maps and Equality	311
Logical Operators	311

Short-Circuiting	313
Hash Tables	313
Courses	315
Refactoring Session	317
Equality	323
The Contract for Equality	326
Apples and Oranges	327
Collections and Equality	329
Hash Tables	330
Collisions	332
An Ideal Hash Algorithm	333
A Final Note on hashCode	335
More on Using HashMaps	337
Additional Hash Tables and Set Implementations	341
toString	343
Strings and Equality	345
Exercises	346
Lesson 10: Mathematics	349
BigDecimal	350
More on Primitive Numerics	353
Integer Math	354
Numeric Casting	355
Expression Evaluation Order	356
NaN	357
Infinity	358
Numeric Overflow	359
Bit Manipulation	360
Java.lang.Math	368
Numeric Wrapper Classes	370
Random Numbers	371
Exercises	375
Lesson 11: IO	379
Organization	379
Character Streams	380
Writing to a File	385
Java.io.File	388
Byte Streams and Conversion	389
A Student User Interface	390

Testing the Application	393
Data Streams	395
CourseCatalog	396
Advanced Streams	399
Object Streams	399
Random Access Files	407
The Student Directory	408
sis.db.DataFileTest	410
Static Nested Classes and Inner Classes	412
sis.db.DataFile	413
sis.db.KeyFileTest	416
sis.db.KeyFile	418
sis.util.IOUtilTest	419
sis.util.IOUtil	420
sis.util.TestUtil	421
Developing the Solution	422
Exercises	422
Lesson 12: Reflection and Other Advanced Topics	425
Mock Objects Revisited	425
The Jim Bob ACH Interface	427
The Mock Class	429
The Account Class Implementation	431
Anonymous Inner Classes	433
Adapters	435
Accessing Variables from the Enclosing Class	437
Tradeoffs	439
Reflection	440
Using JUnit Code	441
The Class Class	442
Building the Suite	444
Class Modifiers	446
Dynamic Proxy	448
A Secure Account Class	449
Building the Secure Account Solution	451
The SecureProxy Class	456
Problems With Reflection	459
Exercises	459

Lesson 13: Multithreading	461
Multithreading	462
Search Server	462
The Search Class	463
Less Dependent Testing	466
The Server	470
Waiting in the Test	472
Creating and Running Threads	473
Cooperative and Preemptive Multitasking	477
Synchronization	478
Creating Threads with Runnable	480
Synchronized	481
Synchronized Collections	482
BlockingQueue	483
Stopping Threads	485
Wait/Notify	486
Additional Notes on wait and notify	491
Locks and Conditions	492
Thread Priorities	494
Deadlocks	495
ThreadLocal	495
The Timer Class	499
Thread Miscellany	501
Summary: Basic Design Principles for Synchronization	506
Exercises	506
Lesson 14: Generics	509
Parameterized Types	509
Collection Framework	510
Multiple Type Parameters	511
Creating Parameterized Types	511
Erasure	514
Upper Bounds	516
Wildcards	518
Implications of Using Wildcards	520
Generic Methods	522
Wildcard Capture	522
Super	523

Additional Bounds	525
Raw Types	526
Checked Collections	528
Arrays	530
Additional Limitations	531
Reflection	532
Final Notes	532
Exercises	533
Lesson 15: Assertions and Annotations	535
Assertions	535
The assert Statement vs. JUnit Assert Methods	537
Annotations	537
Building a Testing Tool	538
TestRunnerTest	539
TestRunner	540
The @TestMethod Annotation	543
Retention	545
Annotation Targets	546
Skipping Test Methods	548
Modifying TestRunner	548
Single-Value Annotations	550
A TestRunner User Interface Class	552
Array Parameters	553
Multiple Parameter Annotations	555
Default Values	557
Additional Return Types and Complex Annotation Types	558
Package Annotations	559
Compatibility Considerations	561
Additional Notes on Annotations	561
Summary	562
Exercises	562
Additional Lesson I: Swing, Part 1	565
Swing	566
Getting Started	567
Swing Application Design	571
Panels	572
Refactoring	576

More Widgets	580
Refactoring	583
Button Clicks and ActionListeners	586
List Models	588
The Application	591
Layout	594
GridLayout	602
Moving Forward	606
Additional Lesson II: Swing, Part 2	607
Miscellaneous Aesthetics	608
Feel	612
Tables	635
Feedback	640
Responsiveness	647
Remaining Tasks	650
Final Notes	651
Additional Lesson III: Java Miscellany	653
JARs	653
Regular Expressions	656
Cloning and Covariance	662
JDBC	664
Internationalization	673
Call by Reference versus Call by Value	681
Java Periphery	683
What Else Is There?	693
Appendix A: An Agile Java Glossary	701
Appendix B: Java Operator Precedence Rules	715
Appendix C: Getting Started with IDEA	717
IDEA	717
The Hello Project	718
Running Tests	724
Taking Advantage of IDEA	729
Agile Java References	733
Index	735

About the Author

Jeff Langr is an independent software consultant with a score and more years of development experience. He provides expertise to customers in software development, design, and agile processes through his company Langr Software Solutions (<http://www.LangrSoft.com>).

Langr worked for Uncle Bob Martin for two years at the esteemed Object Mentor, recognized as the premier XP consulting firm. He consulted at and has been employed by several Fortune 500 companies, as well as the obligatory failed dot-com.

Langr's background includes teaching a university course on Java. He has successfully taught hundreds of other professional students in Java, TDD, XP, and object-oriented development. Langr has spoken about software development numerous times at national conferences and local user group meetings.

Langr wrote the Prentice Hall book *Essential Java Style* (Langr2000), a guide for building quality Java code. Five years later you can still find blog entries of people who swear by their dog-eared copy. Several of Langr's articles on Java and TDD have appeared in *Software Development*, *C/C++ Users Journal*, and various online magazine sites including Developer.com. You can find links to these and other articles at <http://www.langrsoft.com/resources.html>.

He would like to add that he lives for software development, and hopes his love for the craft of building code comes through in Agile Java. Langr resides in Colorado Springs with his wife Kathy and his children Katie, Tim, and Anna.

This page intentionally left blank

Foreword

Jeff Langr has written a very interesting Java book in *Agile Java: Crafting Code with Test-Driven Development*. Its purpose is to teach new programmers the Java language, and to do so in the context of the best development approach that he and I know—namely Test-Driven Development (TDD). It's an undertaking of great potential value, and Jeff has done it nicely. It's my pleasure to provide this foreword and to recommend this book.

Agile Java isn't just for rank beginners. It's also a good book for bringing experienced programmers new to the Java language up to speed. It won't replace a certification manual or one of those huge books of "Everything Java." That's not its point. The point of *Agile Java* is to get you up to speed. Better yet, you come up to speed using TDD, which will serve you well in future learning and in your day-to-day work.

The book starts right off with object-oriented concepts and ideas. It helps if you know a bit about objects when you step in, but if you don't, hang on and you should pick up the basic ideas as you go along. Furthermore, every step of the way you'll be using the Test-Driven Development technique. If you haven't tried TDD, it may seem a bit odd at the beginning, but if you're like most of us who have given it a fair try, it will become a frequently-used tool in your kit.

If you already have Java and JUnit set up on your machine, dig right in. If not, be sure to use the "Setting Up" chapter to get your system correctly configured before moving on to the real examples. Once you can compile and run a simple Java program on your machine, you're ready to go.

Jeff asks you to type in the tests and example code, and I would echo that request. The TDD discipline is one that is learned by doing and practicing, not just by reading. You need to develop your own sense of the rhythm of development. Besides, typing in the examples from any programming book is the best way to learn what it has to offer.

In *Agile Java*, Jeff helps you build pieces of two applications. One of these relates to a student information system, the other focuses on playing chess. By the time you have worked through all the chapters, Jeff has introduced you to the basics of Java. Perhaps more importantly, you have met some of the most important deep capabilities, including interfaces, polymorphism, mock objects, reflection, multi-threading, and generics.

I found that Lesson 10, regarding the mathematical features of Java, particularly brought out the way I like to use tests as part of my learning of a new fea-

ture of a language or library. It's easy to read about something like BigDecimal and think, "I get it." For a while, I might even really get it. But when I encode my learning as tests, two things happen: First, I learn things about the topic that I would have missed if just reading about it. Writing the code pounds ideas into my thick head a bit better than just reading. Second, the tests record what I've learned as well as my thought process as I learned it. Because I've developed the habit of saving all my test cases, I can refer back to them and quickly refresh my memory. Often I'll even put a book and page reference into the tests as a comment, in case I want to go back later and dig out more.

Lesson 11, regarding I/O, includes a nice example of something I'm not very familiar with. Since I don't work in Java much, and most of the languages I commonly use don't have an equivalent, I'm not familiar with nested classes. Jeff gives a good example of when we would be well-advised to use nested classes, and shows how to use and test them.

As I write this foreword, I'm really getting into the book, because Jeff is taking me where I've not gone before. I like that. Lesson 12 is about Mock Objects, and the first example is one that we agile software developers encounter often: How can we deal in our incremental development with an external API that is fairly well defined but isn't available yet? Jeff shows us how to do this by defining the interface—from the documentation if necessary—and then by building a Mock Object that represents our understanding of what the API will do when we finally get it. Writing tests against our Mock Object gives us tests that we can use to ensure that the API does what we expect when we finally get the real code. An excellent addition to your bag of tricks!

Jeff is an educator, and a darn good one. Jeff wants us to think, and to work! He knows that if you and I are ready to learn, we have to practice: we have to do the work. His chapters have exercises. We are well-advised to think about all of them, and to do the ones that cover topics we aren't familiar with. That's how we'll really hammer these ideas into our heads. Read and study his examples, type them in to drill them into your mind, and then follow his lead as you work the interesting examples. You'll be glad you did.

Agile Java: Crafting Code with Test-Driven Development offers you at least three benefits: You'll learn things about Java that you probably didn't know, even if you're not an absolute beginner. You'll learn how to use test-driven development in a wide range of cases, including some in which you'd probably find difficult to invent on your own. And, through building up your skill, you'll add this valuable technique to your professional bag of tricks.

I enjoyed the book and found it valuable. I think you will, too. Enjoy!

Ron Jeffries
www.XProgramming.com
Pinckney, Michigan
November 2, 2004

Acknowledgments

Many thanks to all of the fantastic developers and learning-to-be developers for their insightful comments on *Agile Java*.

Thanks to Bob Martin of Object Mentor for being a great mentor, ex-employer, and book series editor! Bob's review early on helped guide *Agile Java* to a more concise implementation. I'm exceptionally proud to have my book in the Robert Martin Series. I'm also very honored to have Ron Jeffries contribute kind words on the book.

Jeff Bay of ThoughtWorks did an outstanding job and gave me just what I asked for—brutally honest feedback. Jeff is responsible for providing the stellar exercises at the end of each section.

Many thanks to Steve Arneil, Dave Astels, Tim Camper, Dan D'Eramo, Michael Feathers, Paul Holser, Jerry Jackson, Ron Jeffries, Bob Koss, Torri Lopez, Andrew Masters, Chris Mathews, Jim Newkirk, Wendy and David Peterson, Michael Rachow, Jason Rohman, Tito Velez, and Jeroen Wenting. These reviewers provided extensive feedback on the book.

Special thanks to Paul Petralia and his crew at Prentice Hall for making my experience as smooth as possible.

Once again I thank my sister Christine Langr for providing the cover photo and helping me work through various design questions.

Thanks also go to Jim Condit of San Antonio for the Nerf play and use of his abode, and to my wife Kathy Langr for providing examples (and for again putting up with me throughout the arduous writing process).

This page intentionally left blank

Introduction

I am a software craftsman.¹ I have spent much of my software development career trying to quickly build solutions to problems. At the same time I have tried to ensure that my solutions demonstrate carefully crafted code. I strive for perfection in code, yet I know that it is unattainable, particularly with the constant pressure from business to get product out the door. I take modest pride in the code I build daily, yet when I look at the code I wrote the day before, I wonder, “What the heck was I thinking?” This is what keeps the craft challenging to me—the constant desire to figure out how to do things a little better the next time, and with a little less pain than this time.

Agile Java represents a successful approach to learning and mastering Java development. It is based on the way I have learned to best teach programming and to learn new programming languages myself: Using test-driven development (TDD), a technique that introduces a large amount of low-level feedback. This feedback allows you to more quickly see the results of your actions. Using TDD, you will learn how to craft Java code to produce solid object-oriented designs and highly maintainable high-quality systems.

I have used TDD in production systems for over four years and am still amazed at what it does for me. It has improved the quality of my code, it has taught me new things each week, it has made me more productive. I have also created and taught language courses using TDD, both at my own company and at Object Mentor (which continues to teach their language courses with this approach).

Prior to learning TDD, I spent more than fifteen years learning, developing in, and teaching languages the “classic” way—without using tests to drive the development. The student builds and executes example code. The student obtains feedback on what the code is teaching by viewing the output from code execution. While this is a perfectly valid approach, my anecdotal experience is that using it results in less-than-ideal retention of language details.

¹See [McBreen2000].

In contrast, the high volume of rapid feedback in TDD constantly reinforces correct coding and quickly points out incorrect coding. The classic code-run-and-observe approach provides feedback, but at a much slower rate. Unfortunately, it is currently the predominant method of teaching programming.

Others have attempted more innovative approaches to teaching. In the 1990s, Adele Goldberg created a product known as LearningWorks designed for teaching younger students. It allowed a user to directly manipulate visual objects by dynamically executing bits of code. The user saw immediate results from their actions. A recent Java training tool uses a similar approach. It allows the student to execute bits of code to produce visual effects on “live” objects.

The problem with approaches like these is that they are bound to the learning environment. Once you complete the training, you must still learn how to construct your own system from the ground up, without the use of these constrained tools. By using TDD as the driver for learning, you are taught an unbounded technique that you can continue to use in your professional software development career.

Agile Java takes the most object-oriented approach feasible. Part of the difficulty in learning Java is the “bootstrapping” involved. What is the minimum you must learn in order to be able to write classes of some substance?

Most books start by teaching you the prototypical first Java program—the “hello world” application. But it is quite a mouthful: `class Hello { public static void main(String[] args) { System.out.println("hello world"); } }`. This brief program contains at least a dozen concepts that you must ultimately learn. Worse, out of those dozen concepts, at least three are nonobject-oriented concepts that you are better off learning *much* later.

In this book, you will learn the right way to code from the start and come back to fully understand “hello world” later in the book.² Using TDD, you will be able to write good object-oriented code immediately. You’ll still have a big initial hurdle to get over, but this approach keeps you from having to first understand not-very-object-oriented concepts such as static methods and arrays. You will learn all core Java concepts in due time, but your initial emphasis is on objects.

Agile Java presents a cleaner break from the old way of doing things. It allows you to pretend for a while that there was never a language called C, the syntactical basis for Java that has been around for 30 years. While C is a

²Don’t fret, though: You’ll actually start with the “hello world” application in order to ensure you can compile and execute Java code. But you won’t have to understand it all until later.

Who Is This Book For?

great language, its imprint on Java left a quite a few constructs that can distract you from building good object-oriented systems. Using Agile Java, you can learn the right way of doing things before having to understand these legacies of the Java language.

Who Is This Book For?

I designed Agile Java for new programmers who want to learn Java as their first language. The book can also be effective for programmers familiar with TDD but new to Java, or vice versa. Experienced Java developers may find that going through Agile Java presents them with a new, and I hope better, way of approaching things.

This edition of Agile Java covers Java 2 Standard Edition (J2SE) version 5.0.

Sun has made available dozens of class libraries, or APIs (application programming interfaces), that enhance the core Java language. Some examples: JMS (Java Messaging Service) provides a definition for standard messaging-based solutions. EJBs (Enterprise Java Beans) provide a means of building component-based software for large enterprise solutions. JDBC (Java DataBase Connectivity) supplies a standard interface for interacting with relational databases. About a dozen of the advanced APIs are collectively known as J2EE (Java 2 Enterprise Edition). Many of the APIs require entire books for comprehensive coverage. There are dozens of books on J2EE.

This book covers only a few of the additional APIs at an introductory level. Technologies that are used pervasively in the majority of enterprise applications, such as logging, JDBC, and Swing, are presented in Agile Java. Some of the information (for example, logging) will teach you all you need to know for most applications. Other lessons (for example, Swing and JDBC) will teach you a basic understanding of the technology. These lessons will provide you with enough to get started and will tell you where to go for more information.

If you are developing mobile applications, you will be using J2ME (Java 2 Micro Edition). J2ME is a version of Java geared toward environments with limited resources, such as cell phones. J2ME has some significant limitations compared to J2SE. This book does not discuss anything specific with respect to J2ME. However, most of the core techniques and concepts of Java development are applicable to the J2ME environment.

In order to use any of these add-on Java technologies, you must first understand the core language and libraries provided in J2SE. Agile Java will help you build that knowledge.

What This Book Is Not

Agile Java is not an exhaustive treatise on every aspect of the Java language. It instead provides an agile approach to learning the language. Rather than giving you all the fish, I teach you how to fish and sometimes where to find the fish. Agile Java will teach you the majority of the core language concepts. Indeed, upon completing the core fifteen lessons, you will be able to produce quality production Java code. However, there are bound to be a few esoteric language features and nuances that I do not cover in the book.

One way to become familiar with the dusty corners of the language is to peruse the Java Language Specification (JLS). The second edition of the language specification is available at <http://java.sun.com/docs/books/jls>. This edition covers versions of Java up to but not including J2SE 5.0. A third edition of the JLS is in the works at the time I write this. You can find a maintenance review version at http://java.sun.com/docs/books/jls/java_language-3_0-mr-spec.zip.

For additional understanding of what is in the Java API library, the Java API documentation and actual source code is the best place to go.

Agile Java is not a certification study guide. It will not prepare you for a certification exam. A good book on certification will teach you how to take a test. It will also teach you how to decipher (deliberately) poorly written code that shouldn't have been written that way in the first place. Agile Java will teach you instead how to write professional Java code.

Agile Java doesn't attempt to coddle you. Learning to program is a significant challenge. Programming involves thinking and solving problems—it is not an easy undertaking for idiots or dummies. I've tried to avoid insulting your intelligence in this book. That being said, I have tried to make Agile Java an enjoyable, easy read. It's a conversation between you and me, much like the conversations you will continue to have in your professional development career. It's also a conversation between you and your computer.

TDD Disclaimer

Some developers experienced in TDD will note stylistic differences between their approach and my approach(es) in Agile Java. There are many ways to do TDD. None of these techniques are perfect or ordained as the absolute right way to do things. Do whatever works best for you. Do what makes the most sense, as long as it doesn't violate the basic tenets set forth in Agile Java.

Readers will also find areas in which the code could be improved.³ Even as a beginning developer, you no doubt will encounter code in Agile Java that you don't like. Let me know. Send in your suggestions, and I may incorporate them in the next edition. And fix your own implementations. *You can improve almost any code or technique.* Do!

After the first lesson in Agile Java, the tests appear wholesale, as if they were coded in one fell swoop. This is not the case: Each test was built assertion by assertion, in much smaller increments than the book can afford to present. Keep this in mind when writing your own code—one of the most important aspects of TDD is taking small, small steps with constant feedback. And when I say small, I mean small! If you think you're taking small steps, try taking even smaller steps.

How to Use This Book

The core of Agile Java is fifteen lessons of about 30 pages each. You will start with baby steps in Java, TDD, and OO. You will finish with a strong foundation for professional Java development.

The core lessons are sequential. You should start at Lesson 1 and complete each lesson before proceeding to the next. Once you have completed the core lessons, you should have a solid understanding of how to build robust Java code.

If you haven't completed the fifteen core lessons, you should not assume you know how to write good Java code! (Even if you have completed the lessons, you're not an expert . . . yet.) Each lesson builds upon the previous lessons. If you stop short of completing the lessons, your lack of full understanding may lead you to construct poor code.

Each lesson starts with a brief overview of the topics to be discussed. The remainder of the lesson is a narrative. I describe each language feature in the text, specified by test code. I demonstrate each feature in a corresponding code implementation. I have interspersed discussions of TDD technique, OO principles, and good development practices in these lessons.

I've supplied three additional lessons to cover a few more Java topics. Two of the lessons present an introduction to Java's tool for user interface development, Swing. These two lessons will provide you with enough information to begin building robust user interface applications in Java. But the bigger in-

³A constant developer pair would have helped.

tent is to give you some ideas for how to build them using TDD. The third additional lesson presents an overview for a number of Java topics, things that most Java developers will want to know.

For the most effective learning experience, you should follow along with the lessons by entering and executing each bit of test and implementation code as I present it. While the code is available for downloading (see the next paragraph), I highly recommend that you type each bit of code yourself. Part of doing TDD correctly is getting a good understanding of the rhythm involved in going back and forth between the tests and the code. If you just download the code and execute it, you're not going to learn nearly as much. The tactile response of the keyboard seems to impart a lot of learning.

However, who am I to make you work in a certain way? You may choose to download the code from <http://www.LangrSoft.com/agileJava/code>. I've organized this code by lesson. I have made available the code that is the result of working each lesson—*in the state it exists by the end of the lesson*. This will make it easier for you to pick up at any point, particularly since many of the examples carry through to subsequent lessons.

Exercises

Each of the fifteen core lessons in Agile Java has you build bits and pieces of a student information system for a university. I chose this single common theme to help demonstrate how you can incrementally build upon and extend existing code. Each lesson also finishes with a series of exercises. Instead of the student information system, the bulk of the exercises, provided by Jeff Bay, have you build bits and pieces of a chess application.

Some of the exercises are involved and quite challenging. But I highly recommend that you do every one. The exercises are where the real learning starts—you're figuring out how to solve problems using Java, without my help. Doing all of the exercises will give you a second opportunity to let each lesson sink in.

Conventions Used in This Book

Code flows with the text, in order to make it part of the conversation. If I refer to code “below,” it’s the next piece of code as you continue to read; code “above” appears in recent prior text.

Conventions
Used in this
Book

Code (below) appears in a non-proportional font:

```
this.isCode();
```

Smaller portions of a large block of code may appear in **bold**. The **bold** indicates either new code that relates to the current example or code that has particular relevance:

```
class B {
    public void thisIsANewMethod() {
    }
}
```

Code appearing directly in text, such as `this.someCode()`, also appears in a non-proportional font. However, the names of classes, such as `CourseSession`, appear in the same font as the rest of the text.

I often use ellipses in the code samples. The ellipses indicate code that already exists in a class but is not relevant to the current discussion or example:

```
class C {
    private String interestingVariable;
    ...
    private void someInterestingMethod() {
    }
    ...
}
```

The dialog in Agile Java alternates between expressing ideas in English and expressing them in code. For example, I may refer to the need to cancel a payroll check or I might choose to say that you should send the `cancel` message to the `PayrollCheck` object. The idea is to help you start making the necessary connections between understanding requirements and expressing them in code.

Class names are by convention singular nouns, such as `Customer`. “You use the `Customer` class to create multiple `Customer` objects.” In the name of readability, I will sometimes refer to these `Customer` objects using the plural of the class name: “The collection contains all of the `Customers` loaded from the file system.”

New terms initially appear in *italics*. Most of these terms appear in the Glossary (Appendix A).

Throughout Agile Java, you will take specifications and translate them into Java code. In the tradition of agile processes, I present these specifications using informal English. They are requirements, also known as *stories*. A story is a promise for more conversation. Often you will need to continue a dialog with the presenter of the story (sometimes known as the customer) in order to obtain further details on a story. In the case of Agile Java, if a story

isn't making sense, try reading a bit further. Read the corresponding tests to see how they interpret the story.

 I have highlighted stories throughout Agile Java with a "story-telling" icon. The icon emphasizes the oral informality of stories.

Your best path to learning the craft of programming is to work with an experienced practitioner. You will discover that there are many "secrets" to programming. Some secrets are basic concepts that you must know in order to master the craft. Other secrets represent pitfalls—things to watch out for. You must bridge these challenges, and remember what you learned, to be a successful Java programmer.



I've marked such critical points with a bridge icon (the bridge is crossing over the pitfalls). Many of these critical points will apply to development in any language, not just Java.

An Agile Overview

What Is “Agile?”

This chapter provides a brief introduction to some of the core concepts in the book, including Java, object-oriented programming, and test-driven development. You will get answers to the following questions:

- What is “agile”?
- What is Java?
- What is object-oriented programming?
- Why OO?
- What is an object?
- What are classes?
- Why UML?
- What is inheritance?
- Why test-driven development?

What Is “Agile?”

This book is called Agile Java. “Agile” is a new catchphrase for describing a bunch of related software development *methodologies*. Loosely defined, a methodology is a process for building software. There are many ways to approach constructing software as part of a team; many of these approaches have been formalized into published methodologies. There are dozens of major processes in use; you may have heard the names of some of them: waterfall, RUP (Rational Unified Process), XP (extreme programming), and Scrum.

Methodologies have evolved as we learn more about how to effectively work. Waterfall is an older approach to building software that promotes copious documentation, rigid up-front (i.e., at the start of the project) definitions of requirements and system design, and division of a project into serialized phases. Its name derives from a diagram that shows progress flowing down from one phase to the next.

What Is “Agile”?

While waterfall may be appropriate for some efforts, it has some limitations that can cause severe problems on other software development projects. Using a waterfall process means that you are less able to adapt to change as a project progresses. For these reasons, waterfall is sometimes referred to as a *heavyweight* process. It is heavy because it requires considerable baggage and restricts development teams from being able to rapidly respond to change.

In contrast, *agile* processes, a fairly recent phenomena, are *lightweight* processes. When following an agile process, there is less emphasis on written documentation and on having everything finalized up front. Agile processes are designed around accommodating change. XP is probably the most well-known example of an agile process.

RUP is another very well-known process. Technically RUP is a *process framework*: It supplies a catalog of things you can mix and match in order to create your own customized process. Such a RUP *process instance* can straddle the fence between being a heavyweight and an agile process.

When following any kind of process, heavyweight or agile, you must always do the following things when building software:

- analysis: determine what you need to build by gathering and refining requirements for what the software should do
- planning: figure out how long it will take to build the software
- design: determine how to put together what you are building
- coding: construct the software using one or more programming languages
- testing: ensure that what you built works
- deployment: deliver the software to the environment in which it will be executed and used
- documentation: describe the software for various audiences, including end users who need to know how to use the software and developers who need to know how to maintain the software
- review: ensure, through peer consensus, that the software remains maintainable and retains high quality

If you are not doing all of these things, you are *hacking*—producing code with abandon and no consideration for business needs. Some processes such as XP appear not to promote everything in the above list. In reality, they do, but in a different, kind-of-sideways fashion.

I cover little with respect to methodology in this book. You can find shelves of books devoted to learning about methodologies. Most develop-

ment teams use a hybrid methodology—they take successful techniques from various published methodologies and adapt them to their own needs. Many development shops do not even acknowledge a methodology at all. But even these shops usually follow some sort of unstated process.

This book is about constructing the software. Agile Java is centered on a technique known as test-driven development, or TDD. TDD is not a methodology itself but a practice that you can use as part of virtually any software development process. TDD derives from XP, where it is one of about a dozen required practices. *You do not need to do XP in order to do TDD or in order to use this book!*

I do not discuss XP or other methodologies often in this book. I've centered Agile Java on a technique that will allow you to build and mold your code in an agile fashion. You will learn how to accommodate change in your system via TDD. Even if you are using the most rigid process imaginable, you can still use the techniques in Agile Java to construct high-quality Java code.

What Is Java?

What Is Java?

When you hear people refer to Java, they often mean the Java language. The Java language allows you to write instructions, or code, that your computer will interpret in order to execute a software *application*, or *program*. Examples of applications are Microsoft Word, Netscape, and the little clock that runs in the corner of your monitor. *Coding*, or *programming*, is the act of writing programs.

Another thing people mean when they refer to Java is the Java *platform*. The term platform usually indicates an underlying operating system, such as Windows or Unix™, on which you can run applications.¹ Java acts as a platform: It not only defines a language but also gives you a complete environment in which to build and execute your programs. The Java platform acts as a layer between your application and the underlying operating system; it is a mini-operating system in itself. This allows you to write code in a single language and execute it on virtually all major operating systems today.

You will download the Java *software development kit* (SDK), which provides you with three main components:

- a *compiler* (`javac`)
- a virtual machine, or VM (`java`)
- a *class library*, or *API* (application programming interface)

¹[WhatIs2004].

The compiler is a program that reads Java *source files*, ensures they contain valid Java code, and produces *class files*. A source file is a text file that contains the code you type. The class files you generate using the compiler contain *byte codes* that represent the code you typed in. Byte codes appear in a format that the VM can rapidly read and interpret.

The VM (or JVM—Java virtual machine) is a program that executes the code within your class files. The term “virtual machine” derives from the fact that it acts as if it were a complete platform, or operating system, from the standpoint of your Java programs. Your code does not make direct calls to the operating system application programming interface (API), as it might if you were programming directly for Windows in a language such as C or C++.

The Numbers Game

The history of Java contains a handful of significant releases. The first public release of Java (in 1996) was known as Java 1.0. Versions 1.1, 1.2, 1.3, and 1.4 followed, separated by roughly one year each. As of version 1.2, Sun renamed the platform from simply Java to Java 2. With the latest release, Sun chose to embark upon a new numbering scheme, presumably to impart the significance of the release. The version you will learn in this book is formally known as Java 2 Standard Edition 5.0. I will use the abbreviated name J2SE 5.0 throughout the book.

Instead, you write code only in the Java language and use only a set of libraries, known as *class libraries*, or APIs, provided as part of the Java SDK. You might see references to the VM as an *interpreter*, since that is its chief responsibility: The VM interprets the needs of your code and dispatches these needs to the underlying operating system as necessary. The VM is also responsible for allocating and controlling any memory that your code needs.

Java is known as an *object-oriented* programming language. The basic premise of an object-oriented programming language is that you can create abstractions of real-world things, or objects, in code. For example, you can create a Java code representation of a calculator that emulates some of the characteristics and behavior of a real-world calculator.

If you have never programmed before, you are in luck: Object-oriented (OO) programming is much easier to learn from scratch than if you have already learned a non-OO programming language like Cobol or C. If you *have* been tainted by a procedural or declarative language, you may want to read this chapter a few times. Expect that learning OO is not easy; expect to spend a few months before the light bulb in your head stays on all the time without blinking.

This chapter explains the basics of object orientation. Conceptually, OO is reasonably straightforward, but it can take some time for the ideas to sink in, particularly without concrete code examples. Hence, this chapter is brief; instead, you will learn the bulk of OO as you learn to code in Java.

What Is an Object?

Why OO?

Object-oriented programming has been around for quite some time—since the 1960s—but only during the 1990s did it begin to truly take off in terms of acceptance. Much of what we know about how to program well with objects has been discovered in the past decade. The most important thing we have learned is that OO, done properly, can improve your ability to manage and maintain software applications as they mature and grow.

What Is an Object?

An object is a code-based *abstraction* of some relevant concept. If you are building a payroll system, an object could be a programming representation of a check, an employee's rate classification, or a check printer. It could also be an abstraction of an activity, such as a routing process to ensure that employees are directed to the appropriate processing queue. Abstraction is best defined as “amplification of the essential, elimination of the irrelevant.”²

A check object might contain many details such as check number, payee, and amount. But the payroll system is not interested in things such as the size and color of the check, so you would not represent these things in payroll code. You might consider them for the check printing system.

There is some value to using objects that have real-world relevancy, but you must be careful not to take the real world too far into your code. As an example, a payroll system might have an employee object, containing details such as an employee ID and his salary. You also want to ensure that you can give raises to an employee. A real-world design might suggest that the code for giving raises belongs elsewhere, but in fact the most appropriate solution in an OO system is for the employee object to execute the code to apply a raise. (“What? An employee giving himself a raise??”) In Agile Java, you will discover how to make these sorts of design decisions.

²[Martin2003].



What Are Classes?

Figure 1 *Sending a Message*

An object-oriented system should be primarily about behavior. The core concept of OO is that objects send messages to each other in order to effect behavior. One object sends a message to another object to tell it to do something. As a real-world example, I send you a message to you by telling you to lock the front door. In an OO system, a security object sends a message to a door controller object, telling it to secure the door it controls (see Figure 1).

The object being sent the message—the door controller in this example—is known as the *receiver*. The receiver decides what to do with the message. You decide to lock the front door by turning the deadbolt. A different person might have chosen to use the latch. In the OO system, the door controller object might have interacted with a hardware device to activate an electromagnetic lock or to cause an electronic deadbolt to eject.

In either case, the sender of the message is interested only in an abstract concept: that the receiver secure the door. The message sender doesn't care how the door is secured; the details of how that happens is something that only the receiver knows. This is a key concept in OO known as *encapsulation*: Objects should hide unnecessary details from other objects in the system.

The fact that different implementations can be dropped into place without the client knowing or caring which one is going to be used represents another very important OO concept known as *polymorphism*. The security system shouldn't know or care whether the door controller is an electromagnetic lock controller or an electronic deadbolt controller. You will learn about polymorphism in greater detail in one of the lessons.

What Are Classes?

There might be five electromagnetic doors in the building where the security system is deployed, all located in different areas. Each door is thus a separate object with a different physical address. However, all doors share the common behavior of needing to be secured, so they all can be classified similarly. A *class* provides a way of defining commonality for a related grouping of objects. It is a template, or blueprint, for constructing new objects.

What Are
Classes?

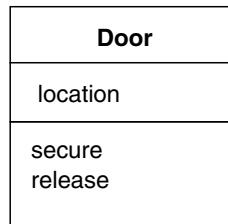


Figure 2 *The Door Class*

The OO security system might define a Door class. The Door class specifies that all Door objects must provide secure and release (unlock) behavior. Furthermore, each Door object should retain its own location. A picture of this Door class is shown in Figure 2.

The class box shows the name of the class in the first (topmost) compartment. The second compartment lists the attributes—the information that

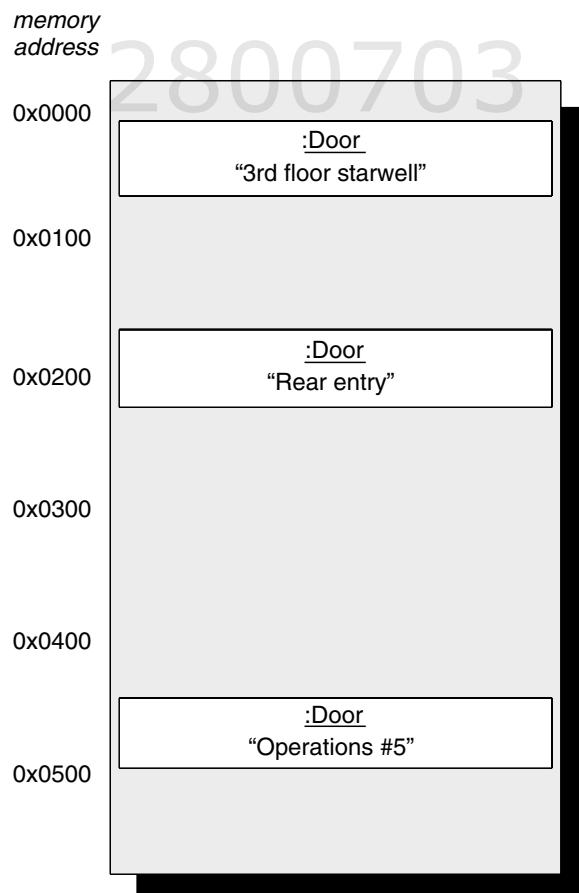


Figure 3 *Door Objects in Memory*

Why UML?

each Door object will store. The third compartment lists the behaviors that each Door object supports—the messages that each door will respond to.

In an object-oriented programming language, you use the Door class as a basis for creating, or *instantiating*, new objects. Each object that you instantiate is allocated a unique area in memory. Any changes you make to an object are exclusive of all other objects in memory.

A *class diagram* is used to represent the structure of an object-oriented system. It shows how classes relate to, or associate with, each other. A class diagram can provide a quick visual understanding of a system. You can also use it as one of the primary indicators of the quality of a system's design.

The basis of all relationships in a class diagram is the association. An association between two classes is formed when one class is *dependent* on the other class or when both classes are dependent on each other. Class A is dependent on class B if class A works only when class B is present.

A main reason class A becomes dependent upon class B is that class A sends one or more messages to class B. Since the Security object sends a message to the DoorController object, the Security class is dependent upon the DoorController class.

You show a one-way dependency between two classes with a navigable association (see Figure 4).

Why UML?

This book uses the UML (Unified Modeling Language) standard to illustrate some of the code. The class boxes shown in earlier sections use UML. The UML is the de facto standard for modeling object-oriented systems. The chief benefit of using UML is that it is universally understood and can be used to quickly express design concepts to other developers.

UML itself is not a methodology; it is a diagramming language. It is a tool that can be used to support documenting any kind of object-oriented system. UML can be used on projects using virtually any methodology.

This book will explain the rudiments of the UML it uses. A good place to obtain a better understanding of UML is the book *UML Distilled*, by Martin

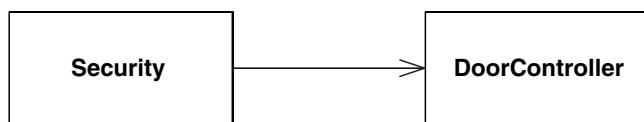


Figure 4 *Class Dependency*

Fowler.³ Also, you can find the latest version of the UML specification online at <http://www.omg.org/technology/documents/formal/uml.htm>.

Most of the UML diagrams in this book show minimal detail. I usually omit attributes, and I only show behaviors when they are particularly relevant. By and large, I intend for the UML to give you a quick pictorial representation of the design of the system. *The UML representation is not the design—it is a model of the design.*⁴ The design is the code itself.

To the extent that UML helps you understand the design of the system, it is valuable and worthwhile. Once it begins duplicating information that is more readily understood by reading code, UML becomes a burdensome, expensive artifact to maintain. Use UML judiciously and it will be an invaluable tool for communication.

What Is
Inheritance?

What Is Inheritance?

A brief discussion of an object-oriented concept known as *inheritance* may help you to understand one of the concepts in the first lesson. Inheritance is a relationship between classes in the system that allows for one class to use another class as a basis for specialization of behavior.

In our real-world example, doors are a class of things that you can shut and open and use as entry/exit points. Beyond that commonality, you can specialize doors: There are automatic doors, elevator doors, bank vault doors, and so on. These are all still doors that open and close, but they each provide a bit of additional specialized behavior.

In your security system, you have the common Door class that specifies secure and release behavior and allows each Door object to store a location. However, you must also support specialty doors that provide additional behaviors and attributes. An AlarmDoor provides the ability to trigger an audible alarm when activated and is also able to operate as a secure door. Since the secure and release behaviors of the AlarmDoor are the same as those in the Door class, the AlarmDoor can designate that it is inheriting from the Door class and need only provide specification for alarm activation.

Inheritance provides the AlarmDoor class with the ability to reuse the secure and release behaviors without having to specify them. For an AlarmDoor object, these behaviors appear as if they were defined in the AlarmDoor class.

³See [Fowler2000].

⁴[Martin2003/Reeves1992].

Why Test-Driven Development?

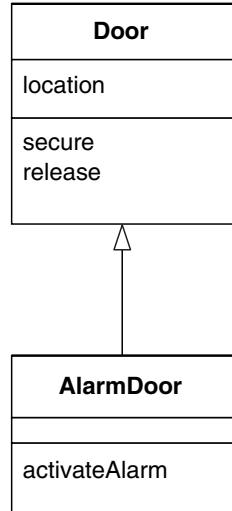


Figure 5 Inheritance

Note that the relationship between AlarmDoor and Door is also a navigable association. AlarmDoor is dependent upon Door—it cannot exist without Door, since AlarmDoor inherits its code behavior and attributes from Door. UML shows the inheritance relationship as a directional arrow with the arrowhead closed instead of opened.

In the first lesson, you'll use inheritance to take advantage of a test framework. In a later lesson, you'll learn about inheritance in greater depth.

Why Test-Driven Development?

You will write virtually all code in this book using TDD. TDD is a technique where you specify the system in terms of tests. You write tests prior to (or as) you write the production code; you do not write production code then supply tests after the fact.⁵

TDD is a simple, short-cycled mechanism that you repeat throughout your coding day. Each cycle lasts from seconds to minutes, and consists of the following steps:

- Write a specification, in code and in the form of a *unit*⁶ *test*.
- Demonstrate test failure (you haven't implemented the specification yet).

⁵[Langr2001].

⁶The test verifies a functional *unit* of your code. Another term for unit test is *programmer test*.

- Write code to meet the specification.
- Demonstrate test success.
- “Refactor,” or rework the code, to ensure that the system still has an optimally clean code base.

That's it. You run all tests against the entire system at all times, ensuring that no new code breaks anything else in the system.

Tests drive several positive aspects of the system:

- *quality*. TDD minimizes the number of defects, since by definition, you test everything in the system. You improve your system's design, since TDD drives you toward more *decoupled* designs—designs where classes are not as heavily dependent upon other classes in the system.
- *documentation of capability*. Each unit test specifies the appropriate use of a production class.
- *malleability*. Having tests in place means that you can continually improve the quality of the code base without fear of breaking something that already works. This can result in lower maintenance costs.
- *consistent pacing*. Since each cycle in TDD is very short, feedback levels are high. You quickly discover if you are going down a rat hole. You learn to maintain a very consistent, sustainable rate of development.

An additional benefit of TDD is that it helps demonstrate the examples in this book. You quickly learn how to translate specifications into test code and how to translate test code into production J2SE 5.0 code.

From a developer's standpoint, TDD is infectious.⁷ Most developers who give it an honest try retain it as a valuable tool in their development toolbox. Many of the developers I've talked to about TDD, including some seasoned Java pioneers, say that they never want to go back to the “old way” of doing things. I was first exposed to the idea of always writing tests for code in 1996, when I saw Kent Beck speak at a Smalltalk Solutions conference. My personal reaction was “I'm a programmer, not a tester!” Today, I'm amazed at the value it brings me and the amount by which it improves my development capabilities. I wouldn't do it otherwise.

Why Test-Driven Development?

⁷[Beck1998].

This page intentionally left blank

Setting Up

This section focuses on what you'll need to do in order to get started.

Software You'll Need

Software You'll Need

IDE or Programmer's Editor

Programming in Java means that you will need to be able to enter source code into your system, compile the source code, and execute the resulting application classes. There are two primary modes of accomplishing this work:

- You can use what is known as an integrated development environment (IDE). An IDE supplies just about everything you'll need to develop in Java in one application. Examples of IDEs include IntelliJ IDEA, Eclipse, Borland JBuilder, and NetBeans.

I'm currently rebuilding the examples for this book using IntelliJ IDEA, available from JetBrains (<http://www.jetbrains.com>). IDEA does a great job with respect to letting you "work the way you want to work." Its complete support for J2SE 5.0 also arrived sooner than the other IDEs mentioned.

- You can use a programmer's editor. A programmer's editor is an application that allows you to type in code and save it to the file system. It will allow you to configure external tools, such as the Java compiler and VM, that you can execute from within the editor. A programmer's editor typically provides additional support to make the task of programming easier. For example, most programmer's editors recognize other languages enough to provide you with meaningful color schemes (a feature known as syntax highlighting).

Some of the more popular programmer's editors include emacs, vi, TextPad, UltraEdit, and SlickEdit. My personal choice is TextPad, a \$27 editor that you can download from <http://www.textpad.com>.

While you could use Windows Notepad (or WordPad) as an editor, you will find that the lack of programming features makes it an ineffective programming tool.

So what is the difference between a programmer's editor and an IDE?

An IDE is often, but not always, limited to a single language. IntelliJ IDEA, JBuilder, and NetBeans specifically target Java, but Microsoft's Visual Studio and Eclipse both provide support for other languages. Modern IDEs know a lot about languages and their class libraries. This knowledge allows the IDEs to provide advanced programming aids such as auto-completion (you type a few letters and the IDE completes what it thought you wanted to type). Modern IDEs also contain sophisticated navigation tools that understand how all your code weaves together. Finally, modern IDEs typically contain a debugger. A debugger is a tool you can use to step through code as it executes. By using a debugger, you can gain a better understanding of what the code does.

In contrast, a programmer's editor is a general-purpose tool that supports editing virtually any kind of program or file. In recent years, programmer's editors have been enhanced to understand more and more about specific languages. However, the level of sophistication remains much lower than an IDE. In order to navigate through code using a programmer's editor, you will typically have to execute text searches. In contrast, IDEs make navigation as simple as a single keystroke. While I know of no programmer's editor that includes a debugger, you can configure most programmer's tools to execute a debugger as an external tool.

Some people find IDEs constricting—you have to work the way the IDE wants you to work. Most IDEs are highly configurable, but you will probably always find something that doesn't work the way you want it to. On the flip side, once you've gotten accustomed to the advanced features that an IDE provides, you'll miss them if you go back to a programmer's editor.

Most of the code in this book was originally constructed under TextPad, since the IDEs generally take a while to support new language versions. You may also find that the IDE you intended to use does not yet support J2SE 5.0. It can take up to a year for vendors to bring their product up to date with the latest Java version.

Agile Java does not assume you use any specific IDE or editor. The few examples in Agile Java that discuss compilation specifics are based upon command-line execution.

If you are using IntelliJ IDEA, Appendix C shows you how to get started with the "Hello World" example shown in this chapter. Appendix C also shows you how to build and run the test-driven examples from Lesson 1. If you're using any other IDE, consult its help documentation for information on configuring it.

Java

You need version 5.0 of the Java 2 SDK (Software Development Kit) in order to run all the examples in Agile Java. If you are running an IDE, it may already include the SDK. I still recommend installing the SDK, as it is wise to learn how to build and execute Java programs exclusive of an IDE.

You can download the SDK from <http://java.sun.com>. You will also want to download the corresponding documentation from the same site. There are installation instructions located at the site to install the SDK that you will want to follow.

Note that you will have the option of downloading either the SDK or the JRE (Java Runtime Environment). You want to download the SDK (which includes the JRE).

The JRE is in essence the JVM. You might download and install the VM if you are interested only in executing Java applications on your system, not in building Java applications. The JVM is supplied separately so that you can ship a minimal set of components with your application when you deploy it to other machines.

After you install the SDK, you will want to extract the documentation, which is a voluminous set of web pages stored in a subdirectory named `doc`. The best place to extract these is into the directory in which you installed the SDK (under Windows, this might be `c:\Program Files\Java\jdk1.5.0`). Once you have extracted the docs, you may want to create a shortcut in your web browser to the various `index.html` files located within. Minimally, you will want to be able to pull up the API documentation, located in the `api` folder (subdirectory), which is contained directly within the `doc` folder.

Finally, if you intend to compile and execute Java programs from the command line, you will need to update your path to include the `bin` directory located directly within the JDK install directory. You will also want to create an environment variable named `JAVA_HOME` that points to the installation directory of the JDK (if one was not already created by installation of the JDK). Refer to an operating system guide for information on how to set the path and environment variables.

Software You'll
Need

Checking Your Java Installation

You can do a quick sanity check of your installation and configuration. From the command line, execute the command:

```
java -version
```

You should see something similar to:

```
java version "1.5.0"  
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-12345)  
Java HotSpot(TM) Client VM (build 1.5.0-12345, mixed mode, sharing)
```

If you instead see something like:

```
'java' is not recognized as an internal or external command, operable program or batch file.
```

Or:

```
java: command not found
```

then you have not set your path correctly. To view the path under Windows, enter the command:

```
path
```

Under Unix, enter:

```
echo $PATH
```

You should see the directory in which you installed Java appear on the path. If you do not, try restarting the shell or command prompt.

If you see a different version of Java, perhaps 1.3 or 1.4, another version of Java appears in the path before the 1.5 version. Try moving your 1.5 install directory to the head of the path.¹

You will also want to test that you properly set the JAVA_HOME environment variable. Try executing (under Windows):

```
"%JAVA_HOME%\bin\java -version
```

You should see the same version as before. Under Unix:

```
"$JAVA_HOME/bin/java" -version
```

The quotes handle the possibility that there are spaces in the path name. If you don't get a good result, view the value of the JAVA_HOME environment variable. Unix:

```
echo $JAVA_HOME
```

Windows:

```
echo %JAVA_HOME%
```

¹Windows puts the latest version of Java in the Windows SYSTEM32 directory. You might consider (carefully) deleting this version.

If you can't get past these steps, go no farther! Seek assistance (see the section Still Stuck? at the end of this chapter).

JUnit

You may need to download and install JUnit in order to get started. JUnit is a simple unit-testing framework that you will need in order to do TDD.

Most major IDEs provide direct or indirect support for JUnit. Many IDEs ship with the JUnit product already built in. If you are building and executing Java applications from the command line or as an external tool from a programmer's editor, you will need to install JUnit.

You can download JUnit for free from <http://www.junit.org>. Installation is a matter of extracting the contents of the downloaded zip file onto your hard drive.

Most important, the JUnit zip file contains a file named `junit.jar`. This file contains the JUnit class libraries to which your code will need to refer in order for you to write tests for your programs.

I built the tests in this book using JUnit version 3.8.1.

Software You'll
Need

Ant

You may also need to download and install Ant. Ant is an XML-based build utility that has become the standard for building and deploying Java projects.

Most major IDEs provide direct or indirect support for Ant. Many IDEs ship with the Ant product already built in. If you are building and executing Java applications from the command line or as an external tool from a programmer's editor, you will want to install Ant.

Ant is available for download at <http://ant.apache.org>. As with JUnit, installation is a matter of extracting the contents of the downloaded zip file onto your hard drive.

You can choose to defer the installation of Ant until you reach Lesson 3, the first use of Ant. This lesson includes a brief introduction to Ant.

If you are not using an IDE with built-in Ant support: You will want to include a reference to the `bin` directory of your Ant installation in your `path` statement. Refer to an operating system guide for information on how to set the `path`. You will also need to create an environment variable named `ANT_HOME` that points to the installation directory of Ant. Refer to an operating system guide for information on how to set the `path` and environment variables.

I built the examples in this book using Ant version 1.6.

Does It Work?

To get started, you will code the “Hello World” program. The hello world program is the prototypical first Java program. When executed, it prints the text “hello world” onto your console. Many programmers venturing into new languages create a hello world program to demonstrate the basic ability to compile and execute code in that new language environment.

Does It Work?

The javac Compiler

The basic format of the javac compiler, included in Sun’s Java Software Development kit, is

```
javac <options> <source files>
```

javac provides numerous command line options. You can obtain a list of the options by entering the command javac and pressing enter. All options are of course optional. Options precede the name of the source file (or files) being compiled. An example javac command:

```
javac -g:none Hello.java
```

In this command, the text -g:none is an option that tells the compiler to generate no debugging information (information used to help you decipher execution problems).

You can always issue the command:

```
java -version
```

with no source file, to see which version of the Java compiler you are executing.

I include “hello world” here solely as a quick means of feedback. Many of the concepts in it are more advanced topics that you will learn later in the book. This exercise exists to ensure you are able to compile and execute Java programs. For now, type the code below, exactly as it appears, into your IDE or editor.

```
class Hello {  
    public static void main(String[] args) {  
        System.out.println("hello world");  
    }  
}
```

Save this using the filename `Hello.java`. Case is important in Java. If you save it as `hello.java`, you may not be able to compile or execute the code.²

²Windows is a little less picky about case. You may find that Windows doesn’t always complain about case when it should.

Compiling Hello World

Either use your IDE to compile Hello (if your IDE requires explicit compilation³) or compile it from the command line:

```
javac Hello.java
```

See the sidebar for more information about the javac command.

You should be in the directory where you saved Hello.java. If all went well, you will see no messages from the compiler. If you mistyped something, you will likely receive compilation errors. For example, you might see:

```
Hello.java:4: ';' expected
  }
  ^
1 error
```

This specific error means that you forgot to type the semicolon (;) at the end of the third line of text.

You might see multiple errors, depending on what you mistyped. Correct any errors and try to compile the file again. If you still have problems, make sure your code looks *exactly* as it appears above. Finally, see the section Still Stuck? at the end of this chapter.

A successful compilation of Hello.java will produce the file Hello.class. Execute a `dir` or `ls` command to produce a directory listing and ensure that Hello.class exists. Hello.class is the binary compiled version of the source code—the *class file*. Hello.class is what the Java VM will read and interpret in order to execute the code you typed in Hello.java.

Executing Hello World

Executing Hello World

To execute the Hello application from the command line, enter:

```
java Hello
```

Make sure that Hello is capitalized as shown.

If you entered the command correctly, you should see the output:

```
hello world
```

³Some IDEs require you to explicitly trigger a build. Other IDEs, such as Eclipse, automatically compile the necessary source files each time you save an edit change.

Still Stuck?

Problems Executing Hello

- If you see:

'java' is not recognized as an internal or external command, ...

or

java: Command not found.

then you either have not installed Java or the Java bin directory is not part of your path statement. Refer to the section Checking Your Java Installation for some assistance.

- If you see:

Exception in thread "main" java.lang.NoClassDefFoundError: hello (wrong name: Hello) ...

then you entered the wrong case, for example:

java hello

Correct the command and try again.

- If you see:

Exception in thread "main" java.lang.NoClassDefFoundError: Hello/class

then you entered the .class portion of the name. Correct the command and try again.

Each time you compile the source file, the Java compiler will replace any associated class file that already exists.

Only where pertinent in this book (for example, when I discuss classpath) do I demonstrate use of the javac and java commands. I highly recommend that you familiarize yourself with command-line compilation and execution. The Java compiler and VM work virtually the same across all platforms. If you have difficulty compiling and executing code within your IDE, you will have to refer to the documentation for your IDE.

Still Stuck?

Getting your first Java program compiled and executed can be frustrating, as there are many things that can go wrong. Unfortunately, I can't cover all possibilities in this book. If you are still having trouble, there are wonderful places to go on the web that will provide you with all the help you need. Sun's official Java web site (<http://java.sun.com>) provides you with a wealth of information right from the source.

Another great place to find assistance is the JavaRanch web site at <http://www.javaranch.com>. You'll need to register with a real name in order to use this service. Most of the moderators of the site are particularly helpful and friendly to Java novices. Look for the forum named "Java in General (beginner)" and post your messages there.

As with any public forum, first read the JavaRanch guidelines on posting. The link <http://www.faqs.org/faqs/usenet/primer/part1/> contains a document on etiquette for posting to Usenet. Most of this etiquette applies to virtually any public forum, including JavaRanch. Asking for help in the right way elicits a much better response.

If you're still stuck, I apologize. Please send an email to me at agileJava@LangrSoft.com.

Still Stuck?

This page intentionally left blank

Lesson 1

Getting Started

Most of the lessons in the first half of Agile Java involve the development of various pieces of a student information system. You will not build a complete system, but you will work on various subsystems that might be part of a complete system.

The student information system involves many different aspects of running a school or university: registration, grades, course scheduling, billing, records, and so on.

In Lesson 1, you will:

- create a simple Java class
- create a test class that exercises the Java class
- use the JUnit framework
- learn about constructors
- refactor the code that you write

This lesson is very detail oriented. I will explicitly describe the steps you should take in doing test-driven development. Future lessons will assume that you are following the cycle of test-driven development in order to produce appropriate tests and code.

Testing

Testing

Test-driven development means that you will write tests for virtually every bit of code. It also means that you will write the tests first. The tests are a means of specifying what the code needs to do. After writing the corresponding code, the tests are run to ensure that the code does what the tests specify.



Figure 1.1 Test and Production Class

Each class you code will have a corresponding test class. In Figure 1.1, `StudentTest` is the test class for the production class `Student`.

`StudentTest` will have to create objects of class `Student`, send messages to these objects, and prove that once all the messages have been sent, things are as expected. `StudentTest` is thus dependent on `Student`, as shown by the navigable association in the diagram. Conversely, `Student` is not dependent on `StudentTest`: The production classes you build should know nothing about the tests written for them.

Design

Design

You design and construct a system based on customer needs, or requirements. Part of the design process is translating the customer requirements into rough ideas or sketches of how the system will be used. For a web-based system, this means designing the web pages that will provide the application's functionality. If you are building "inbetween" software known as middleware, your system will be used by other client software and will interact with other server software. In this case, you will want start by defining the communication points—the *interfaces*—between the other systems and the middleware.

You start by building only high-level designs, not highly detailed specifications. You will continually refine the design as you understand more about the customer needs. You will also update the design as you discover what works well and what doesn't work well in the Java code that you build. The power of object-oriented development can allow you this flexibility, the ability to quickly adapt your design to changing conditions.

Designing the system from the outside in as described above would be a daunting task without complete understanding of the language in which you're going to build it—Java. To get started, you will construct some of the internal building blocks of the system. This approach will get you past the language basics.



The student information system is primarily about students, so as your first task you will abstract that real-world concept into its object-oriented representation. A candidate class might be `Student`.

Student objects should contain basic information about a student such as name, identification number, and grades. You will concentrate on an even smaller concern first: Create a unique student object that stores the name of the student.

The book icon that appears to the left of the preceding paragraph will appear throughout Agile Java. I will use this icon to designate requirements, or *stories*, that you will build into the student information system. These stories are simple descriptions of what you need to build into the system in order to make the customer happy. You will translate stories into detailed specifications that you realize in the form of tests.

A Simple Test

A Simple Test

To represent the initial need to capture student information, start by creating a class that will act as your test case. First, create a new directory or folder on your machine.¹ Then create a file named StudentTest.java in this directory. For the time being, you will save, compile, and execute code out of this single directory. Type the following code into your editor:

```
public class StudentTest extends junit.framework.TestCase {  
}
```

Save the file using the file name StudentTest.java.

The two lines of code in StudentTest.java define a class named StudentTest. Everything that comes between the opening and closing braces ({ and }) is part of the definition of StudentTest.

You must designate the class as `public` in order for the testing framework JUnit to recognize it. I will cover `public` in more depth later. For now, understand that the `public` keyword allows other code, including the code in the JUnit framework, to work with the code you write.

The code phrase `extends junit.framework.TestCase` declares StudentTest to be a subclass of another class named `junit.framework.TestCase`. This means that StudentTest will acquire, or *inherit*, all the capabilities (behavior) and data (attributes) from a class named `junit.framework.TestCase`. StudentTest will also be able to add its own behavior and/or attributes. The `extends` clause also

¹The instructions in this lesson are geared toward command-line use of Java. If you are using an IDE, you will create a class named StudentTest in something called the “default package.” If you are prompted for any package name, do not enter anything.

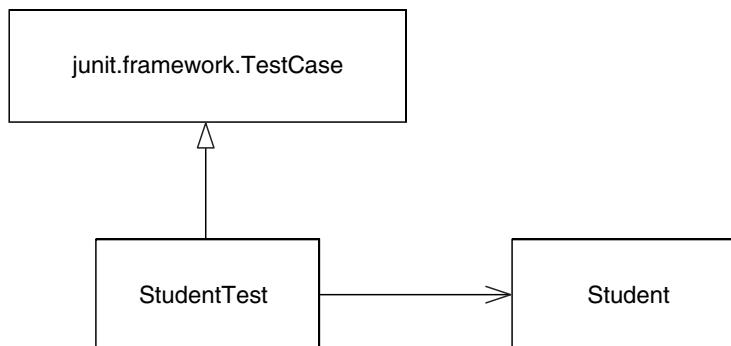


Figure 1.2 *StudentTest Inherits from junit.framework.TestCase*

A Simple Test

allows the JUnit user interface to recognize the `StudentTest` class as something that contains testing methods.

The UML class diagram in Figure 1.2 shows the inheritance relationship between `StudentTest` and `junit.framework.TestCase`. `StudentTest` is now dependent upon both `junit.framework.TestCase` and `Student`. Remember that the arrowhead distinguishes between the type of dependency—a closed arrowhead indicates an inheritance relationship.

Your next step is to compile the `StudentTest` class. In order to do so, you must tell Java where to find any other classes that `StudentTest` references. Currently this includes only the class `junit.framework.TestCase`. This class can be found in a packaged format known as a JAR (Java ARchive) file. The JAR file in which `junit.framework.TestCase` can be found also contains many other classes that comprise the JUnit framework.

I discuss JAR files in more depth in Additional Lesson III. For now, you only need to understand how to tell Java where to find the JUnit JAR file. You do so using the *classpath*.

More on the Classpath

Understanding the classpath can be one of the more confusing aspects of working with Java for beginning developers. For now, you only need a minimal understanding of it.

The classpath is a list of locations separated by semicolons under Windows or colons under Unix. You supply the classpath to both the compiler and the Java VM. A location can be either a JAR file (which contains compiled class files by definition) or a directory that contains compiled class files.

Java depends on the ability to dynamically load other classes when executing or compiling. By supplying a classpath, you provide Java with a list of places to search when it needs to load a specific class.

If you have spaces in your classpath, you may need to surround it with the appropriate quotes for your operating system.

From the command line, you can specify the classpath at the same time that you compile the source file.

```
javac -classpath c:\junit3.8.1\junit.jar StudentTest.java
```

You must specify either the absolute or relative location of the file JUnit.jar;² you will find it in the directory where you installed JUnit. This example specifies the absolute location of JUnit.jar.

You should also ensure that you are in the same directory as StudentTest.java.

If you omit the classpath, the Java compiler will respond with an error message:

```
StudentTest.java:1: package junit.framework does not exist
public class StudentTest extends junit.framework.TestCase {
           ^
1 error
```

JUnit

Your IDE will allow you to specify the classpath somewhere in the current project's property settings. In Eclipse, for example, you specify the classpath in the Properties dialog for the project, under the section Java Build Path, and in the Libraries tab.

JUnit

Once you have successfully compiled StudentTest, you can execute it in JUnit. JUnit provides two GUI-based interfaces and a text interface. Refer to the JUnit documentation for further information. The following command will execute the AWT interface³ against StudentTest.class, using JUnit's class named junit.awtui.TestRunner.

```
java -cp .;c:\junit3.8.1\junit.jar junit.awtui.TestRunner StudentTest
```

²An absolute location represents the explicit, complete path to a file, starting from the drive letter or root of your file system. A relative location contains a path to a file that is relative to your current location. Example: If StudentTest is in /usr/src/student and JUnit.jar is in /usr/src/JUnit3.8.1, you can specify the relative location as ./JUnit3.8.1/JUnit.jar.

³The AWT is Java's more bare-boned user interface toolkit, in contrast with Swing, which provides more controls and features. The AWT version of the JUnit interface is simpler and easier to understand. To use the Swing version, use the class name junit.swingui.TestRunner instead of junit.awtui.TestRunner. To use the text version, use the class name junit.textui.TestRunner.

You once again specify the classpath, this time using the abbreviated keyword `-cp`. Not only does the Java compiler need to know where the JUnit classes are, but the Java VM also needs to be able to find these classes at runtime so it can load them up as needed. In addition, the classpath now contains a `..`, representing the current directory. This is so Java⁴ can locate `StudentTest.class`: If a directory is specified instead of a JAR filename, Java scans the directory for class files as necessary.

The command also contains a single *argument*, `StudentTest`, which you pass to the `junit.awtui.TestRunner` class as the name of the class to be tested.

When you execute the `TestRunner`, you should see a window similar to Figure 1.3.

JUnit

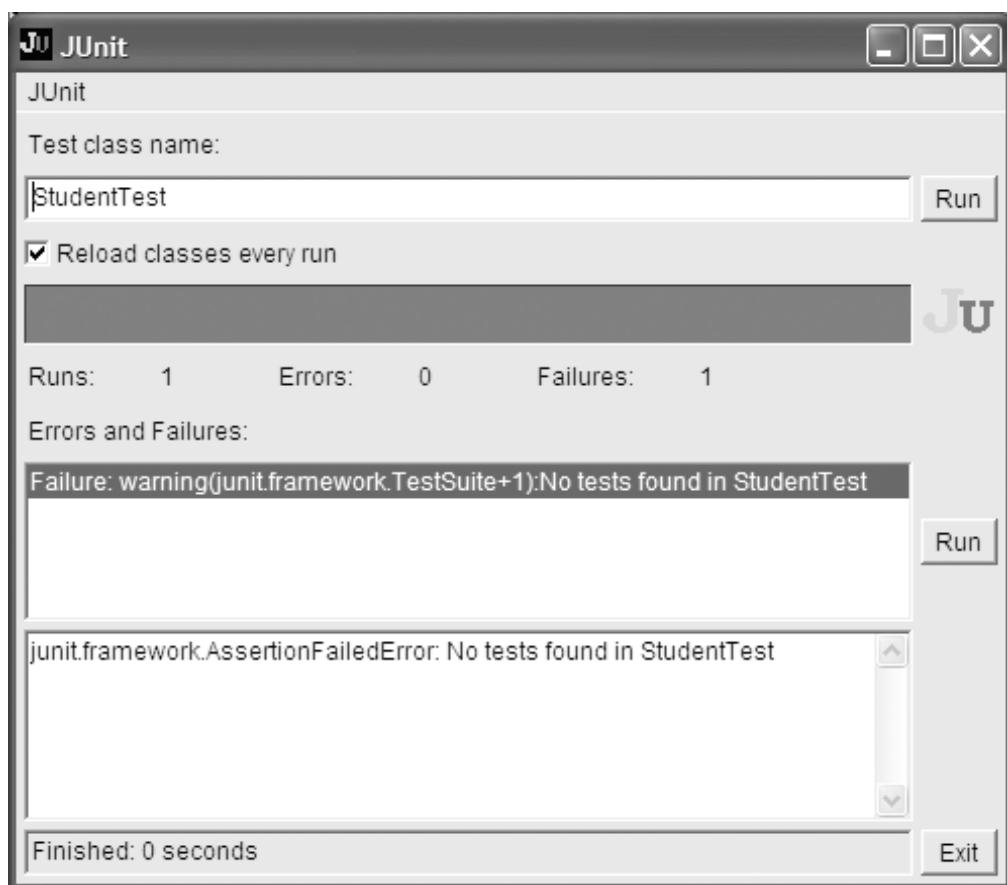


Figure 1.3 JUnit (showing a red bar)

⁴You'll note I use phrases such as “Java does this” often. This is a colloquial (i.e., lazy) way of saying “The Java virtual machine does this,” or “The Java compiler does that.” You should be able to determine whether I’m talking about the VM or the compiler from the context.

There really isn't very much to the JUnit interface. I will discuss only part of it for now, introducing the remainder bit by bit as appropriate. The name of the class being tested, StudentTest, appears near the top in an entry field. The Run button to its right can be clicked to rerun the tests. The interface shows that you have already executed the tests once. If you click the Run button (go ahead!), you will see a very quick flash of the red bar⁵ spanning the width of the window.

The fact that JUnit shows a red bar indicates that something went wrong. The summary below the red bar shows that there is one (1) failure. The Errors and Failures list explains all the things that went wrong; in this case, JUnit complained because there were "No tests found in StudentTest." Your job as a test-driven programmer will be to first view errors and failures in JUnit and then quickly correct them.

Adding a Test

Adding a Test

Edit the source for your StudentTest class to look like the following code:

```
public class StudentTest extends junit.framework.TestCase {
    public void testCreate() {
    }
}
```

The new second and third lines define a *method* within the StudentTest class:

```
public void testCreate() {
```

A method is a block of code that will contain any number of code statements. Like the class declaration, Java uses braces to delineate where the method starts and where it ends. All code that appears between method braces belongs to the method.

The method in this example is named `testCreate`. It is designated as being `public`, another requirement of the JUnit testing framework.

Methods have two general purposes. First, when Java executes a method, it steps through the code contained within the braces. Among other things, method code can call other methods; it can also modify object attributes. Second, a method can return information to the code that executed it.

The `testCreate` method returns nothing to the code that invoked it—JUnit has no need for such information. A method that returns no information provides

⁵If you are color-blind, the statistics below the bar will provide you with the information you need.

what is known as a *void return type*. Later (see Returning a Value from a Method in this lesson) you will learn how to return information from a method.

The empty pair of parentheses () indicates that `testCreate` takes no arguments (also known as *parameters*)—it needs no information passed to it in order to do its work.

The name of the method, `testCreate`, suggests that this is a method to be used for testing. However, to Java, it is just another method name. But for JUnit to recognize a method as a test method, it must meet the following criteria:

- the method must be declared `public`,
- the method must return `void` (nothing),
- the name of the method must start with the word `test`, in lowercase letters, and
- the method cannot take any arguments () .

Compile this code and rerun JUnit's TestRunner. Figure 1.4 shows that things are looking better.

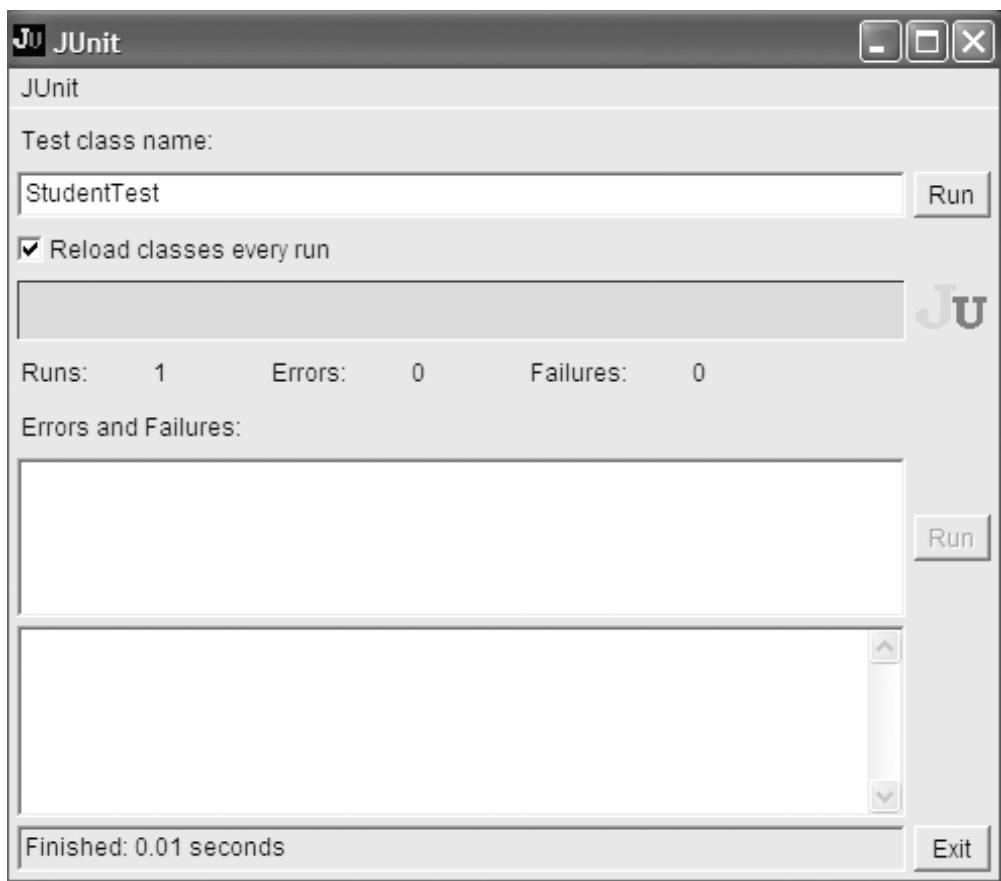


Figure 1.4 JUnit success (showing a green bar)

The green bar is what you will always be striving for. For the test class StudentTest, JUnit shows a successful execution of one test method (Runs: 1) and no errors or failures.

Remember that there is no code in `testCreate`. The successful execution of JUnit demonstrates that an empty test method will always pass.

Creating a Student

Add a single line, or *statement*, to the `testCreate` method:

```
public class StudentTest extends junit.framework.TestCase {
    public void testCreate() {
        new Student("Jane Doe");
    }
}
```

Creating a Student

You terminate each statement with a semicolon (`;`).

When the testing framework calls the `testCreate` method, Java executes this single statement. Once Java executes the statement, it returns control to code in the testing framework that called `testCreate`.

The statement in `testCreate` tells Java to create an object of the class type `Student`:

```
new Student("Jane Doe");
```

You place the `new` keyword before the name of the class to instantiate. You then follow the class name with an *argument list*. The argument list contains information the `Student` class requires in order to be able to instantiate a `Student` object. Different classes will need different pieces of information; some classes will not need any information at all. It is up to the designer of the class (you, in this case) to specify what information must be supplied.

The sole argument in this example represents the name of the student, Jane Doe. The value "Jane Doe" is a *string literal*. String literals represent object instances of the predefined Java class `java.lang.String`. Simply put, string literals are Java's representation of pieces of text.

Soon you will build code for the `Student` class. At that time, you will specify what to do with this `String` argument. There are a few things you can do with arguments: You can use them as input data to another operation, you can store them for later use and/or retrieval, you can ignore them, and you can pass them to other objects.

When the Java VM executes the `new operator` in this statement, it allocates an area in memory to store a representation of the `Student` object. The VM

uses information in the Student class definition to determine just how much memory to allocate.

Creating the Student Class

Compile the test. You should expect to see problems,⁶ since you have only built the test class StudentTest. StudentTest refers to a class named Student, which is nowhere to be found—you haven't built it yet!

**Creating the
Student Class**

```
StudentTest.java:3: cannot find symbol
symbol  : class Student
location: class StudentTest
    new Student("Jane Doe");
           ^
1 error
```

Note the position of the caret (^) beneath the statement in error. It indicates that the Java compiler doesn't know what the source text Student represents.

The compile error is expected. Compilation errors are a good thing—they provide you with feedback throughout the development process. One way you can look at a compilation error is as your very first feedback after writing a test. The feedback answers the question: Have you constructed your code using proper Java syntax so that the test can execute?

Simplifying Compilation and Execution

In order to alleviate the tedium of re-executing each of these statements, you may want to build a batch file or script. An example batch file for Windows:

```
@echo off
javac -cp c:\junit3.8.1\junit.jar *.java
if not errorlevel 1 java -cp .;c:\junit3.8.1\junit.jar junit.awtui.TestRunner StudentTest
```

The batch file does not execute the JUnit test if the compiler returns any compilation errors. A similar script for use under Unix:

```
#!/bin/sh
javac -classpath "/junit3.8.1/junit.jar" *.java
if [ $? -eq 0 ]; then
    java -cp "./junit3.8.1/junit.jar" junit.awtui.TestRunner StudentTest
fi
```

⁶You may see different errors, depending on your Java compiler and/or IDE.

Another option under Unix is to use `make`, a classic build tool available on most systems. However, an even better solution is a tool called Ant. In Lesson 3, you will learn to use Ant as a cross-platform solution for building and running your tests.

To eliminate the current error, create a new class named `Student.java`. Edit it to contain the following code:

```
class Student {  
}
```

Run `javac` again, this time using a wildcard to specify that all source files should be compiled:

```
javac -cp c:\junit3.8.1\junit.jar *.java
```

Constructors

You will receive a new but similar error. Again the compiler cannot find a symbol, but this time it points to the word `new` in the source text. Also, the compiler indicates that the symbol it's looking for is a constructor that takes a `String` argument. The compiler found the class `Student`, but now it needs to know what to do with respect to the "Jane Doe" `String`.

```
StudentTest.java:3: cannot find symbol  
symbol  : constructor Student(java.lang.String)  
location: class Student  
    new Student("Jane Doe");  
          ^  
1 error
```

Constructors

The compiler is complaining that it cannot find an appropriate `Student` *constructor*. A constructor looks a lot like a method. It can contain any number of statements and can take any number of arguments like a method. However, you must always name a constructor the same as the class in which it is defined. Also, you never provide a return value for a constructor, not even `void`. You use a constructor to allow an object to be initialized, often by using other objects that are passed along as arguments to the constructor.

In the example, you want to be able to pass along a student name when you instantiate, or construct, a `Student` object. The code

```
new Student("Jane Doe");
```

implies that there must be a constructor defined in the `Student` class that takes a single parameter of the `String` type. You can define such a constructor by editing `Student.java` so that it looks like this:

```
class Student {
    Student(String name) {
    }
}
```

Compile again and rerun your JUnit tests:

```
javac -classpath c:\junit3.8.1\junit.jar *.java
java -cp .;c:\junit3.8.1\junit.jar junit.awtui.TestRunner StudentTest
```

Local Variables

You should see a green bar.

The constructor currently doesn't do anything with the `name` `String` passed to it. It is lost in the ether! Soon, you'll modify the `Student` class definition to hold on to the name.

Local Variables

So far, when JUnit executes `testCreate`, Java executes the single statement that constructs a new `Student` object. Once this statement executes, control returns to the calling code in the JUnit framework. At this point, the object created in `testCreate` disappears: It had a lifetime only as long as `testCreate` was executing. Another way to say this is that the `Student` object is scoped to the `testCreate` method.

You will want to be able to refer to this `Student` object later in your test. You must somehow hold on to the memory address of where the Java VM decided to store the `Student` object. The `new` operator returns a *reference* to the object's location in memory. You can store this reference by using the *assignment operator*, represented in Java as the equals sign (`=`). Modify `StudentTest`:

```
public class StudentTest extends junit.framework.TestCase {
    public void testCreate() {
        Student student = new Student("Jane Doe");
    }
}
```

The statement is now an *assignment* statement: an object or value on the right-hand side of an assignment operator is stored in a reference on the left-hand side of the operator.

When the Java VM executes this statement, it executes the code to the right-hand side of the assignment operator (=) first, creating a Student object in memory. The VM takes note of the actual memory address where it places the new Student object. Subsequently, the VM assigns this address to a reference on the left-hand side, or first half, of the statement.

The first half of the statement creates a *local variable* reference named student of the type Student. The reference will contain the memory address of the Student object. It is local because it will exist only for the duration of the test method. Local variables are also known as *temp variables* or *temporary variables*.

You could have named the variable someStudent, or janeDoe, but in this case, the generic name student will work just fine. See the section Naming Conventions at the end of this lesson for guidelines on how to name your variables.

A conceptual pictorial representation of the student reference and Student object might look something like Figure 1.5.⁷ This picture exists only to give you an understanding of what's happening behind the scenes. You need not learn how to create your own such pictures.

Behind the scenes, Java maintains a list of all the variables you define and the memory location to which each variable refers. One of the beauties of Java is that you do not have to explicitly code the mundane details of creating and releasing this memory space. Developers using an older language such as C or C++ expend a considerable amount of effort managing memory.

In Java, you don't have to worry as much about managing memory. But it is still possible to create an application with memory "leaks," where the

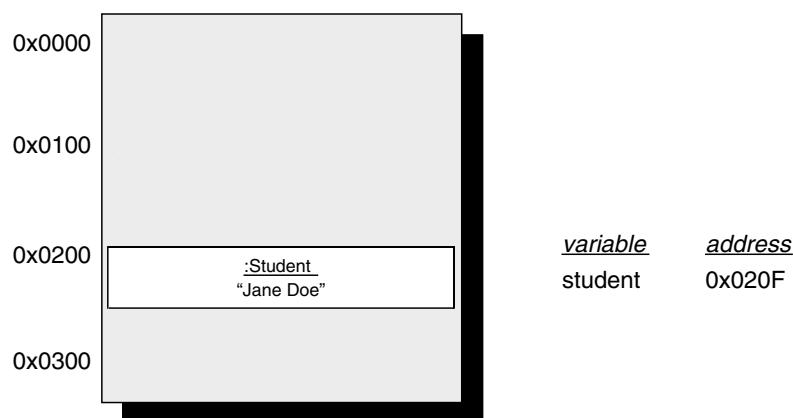


Figure 1.5 Reference to an Object

⁷The memory addresses are shown in hexadecimal, which is indicated by the "0x" that precedes each number. Lesson 10 includes a discussion of hexadecimal numbers.

Local Variables

application continues to use more and more memory until there is none left. It is also possible to create applications that behave incorrectly because of a poor understanding of how Java manages memory for you. You must still understand what's going on behind the scenes in order to master the language.

I will use the conceptual memory diagrams only in this chapter to help you understand what Java does when you manipulate objects. They are not a standard diagramming tool. Once you have a fundamental understanding of memory allocation, you can for the most part put the nuts and bolts of it out of mind when coding in Java.

Recompile all source files and rerun your test. So far, you haven't written any code that explicitly tests anything. Your tests should continue to pass.

Returning a Value from a Method

Returning a Value from a Method

As the next step, you want to ask the Student object created in the test for the student's name.

```
public class StudentTest extends junit.framework.TestCase {
    public void testCreate() {
        Student student = new Student("Jane Doe");
        String studentName = student.getName();
    }
}
```

You now have two statements in `testCreate`. Each statement is terminated by a semicolon. When the Java VM executes `testCreate`, it will execute each statement in turn, top to bottom, before returning control to the calling class.

The second statement is another assignment statement, similar to the first statement. In this statement, however, you are not instantiating a new object. Instead, you are sending a message to the `Student` object, using the `student` reference assigned to in the previous statement.

The right-hand side of this statement is a message send that asks the `Student` object for its name. You, as the programmer and class designer of the `Student` class, are the one who makes the decision about the message name and its arguments. You have decided that the message to which students should respond is called `getName` and that the message will not need to carry any additional information (arguments).

```
student.getName();
```

You also must specify the message *receiver*—the object to which you want the message sent. To do so, you first specify the object reference, `student`, fol-

lowed by a period (.), followed by the message `getName()`. The parentheses indicate that no arguments are passed along with the message. For this second statement to work, you will need to define a corresponding method in the `Student` class called `getName()`.

The left hand of the second statement assigns the memory address of this returned `String` object to a `String` local variable called `studentName`. For this assignment to work, you'll need to define the `getName()` method in `Student` so that it returns a `String` object.

You'll see how to build `getName()` in just a minute.

Compile all source files. The remaining compilation error indicates that the compiler doesn't know about a `getName` method in the `Student` class.

```
StudentTest.java:4: cannot find symbol
symbol : method getName()
location: class Student
    String studentName = student.getName();
                           ^
1 error
```

Returning a
Value from a
Method

You can eliminate this error by adding a `getName` method to the `Student` class definition:

```
class Student {
    Student(String name) {
    }

    String getName() {
    }
}
```

You saw earlier how you can indicate that a method returns nothing by using the `void` keyword. This `getName` method specifies instead a return type of `String`. If you now try to compile `Student.java`, you receive an error:

```
Student.java:5: missing return statement
}
^
1 error
```

Since the method specifies a return type of `String`, it needs a `return` statement that provides a `String` object to be returned to the code that originally sent the `getName` message:

```
class Student {
    Student(String name) {
    }

    String getName() {
        return "";
    }
}
```

The `return` statement here returns an empty `String` object—a `String` with nothing in it. Compile the code again; you should receive no compilation errors. You are ready to run the test in JUnit again.

Assertions

Assertions

You now have a complete context for `testCreate`: The first test statement creates a student with a given name, and the second statement asks for the name from the student object. All that you need do now is add a statement that will demonstrate that the student name is as expected—that it is the same as the name passed to the constructor of the student.

```
public class StudentTest extends junit.framework.TestCase {  
    public void testCreate() {  
        Student student = new Student("Jane Doe");  
        String studentName = student.getName();  
        assertEquals("Jane Doe", studentName);  
    }  
}
```

The third line of code (statement) is used to prove, or assert, that everything went well during the execution of the first two statements. This is the “test” portion of the test method. To paraphrase the intent of the third statement: You want to ensure that the student’s name is the string literal “Jane Doe”. In more general terms, the third statement is an assertion that requires the first argument to be the same as the second.

This third line is another message send, much like the right-hand side of the second statement. However, there is no receiver—to what object is the `assertEquals` message sent? If you specify no receiver, Java assumes that the current object—the object in which the current method is executing—is the receiver.

The class `junit.framework.TestCase` includes the definition for the method `assertEquals`. Remember that in your class definition for `StudentTest`, you declared:

```
public class StudentTest extends junit.framework.TestCase {
```

This declaration means that the `StudentTest` class inherits from `junit.framework.TestCase`. When you send the message `assertEquals` to the current `StudentTest` object, the Java VM will attempt to find a definition for `assertEquals` in `StudentTest`. It will not find one and subsequently will use the

definition it finds in `junit.framework.TestCase`. Once the VM finds the method, it executes it just like any other method.

The important thing to remember is that even though the method is defined in the superclass of `StudentTest`, it operates on the current `StudentTest` object. You will revisit this concept of inheritance in Lesson 6.

The third statement also demonstrates how you can pass more than one argument along with a message by using commas to separate each argument. There are two parameters to `assertEquals`: the String literal "Jane Doe" and the `studentName` reference you created in the second statement. Both represent objects that will be compared within the `assertEquals` method. JUnit will use the result of this comparison to determine if the `testCreate` method should pass or fail. If the string in the memory address referred to by `studentName` is also "Jane Doe", then the test passes.

Recompile and run your test. JUnit should look something like Figure 1.6.

Assertions

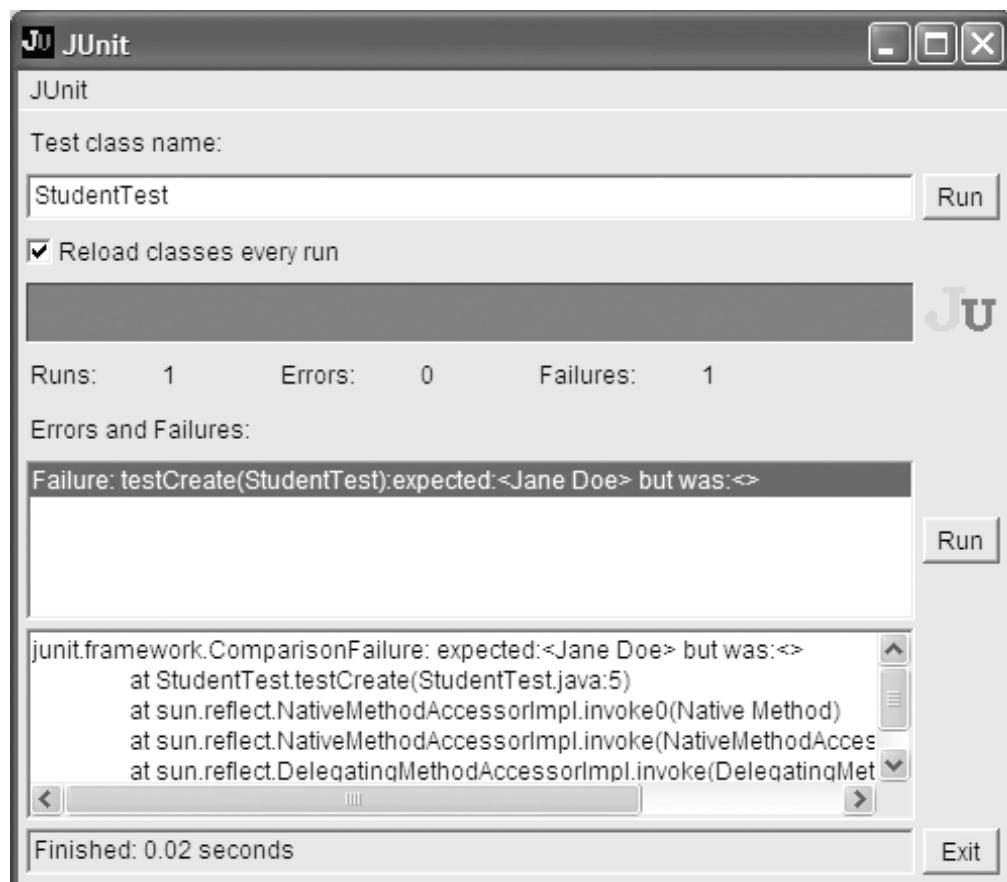


Figure 1.6 Not Quite Done

JUnit shows a red bar and indicates one failure. It also tells you what went wrong in the first listbox:

```
Failure: testCreate(StudentTest): expected:<Jane Doe> but was:<>
```

The method failing is `testCreate`, located in the `StudentTest` class. The problem is that something expected the String literal "Jane Doe" but received instead an empty String. Who is expecting this and where? JUnit divulges this information in the second listbox, showing a *walkback* (also known as a *stack trace*) of the code that led up to the failure. The first line of this stack trace indicates that there was a `ComparisonFailure`, and the second line tells you that the failure occurred at line 5 in `StudentTest.java`.

Assertions

Use your editor to navigate to line 5 of the `StudentTest` source code. This will show that the `assertEquals` method you coded is the source of the comparison failure:

```
assertEquals("Jane Doe", studentName);
```

As mentioned before, the `assertEquals` method compares two objects⁸ and fails the test if they are not equal. JUnit refers to the first parameter, "Jane Doe", as the *expected* value. The second parameter, the `studentName` variable, references the *actual* value—what you are expecting to also have the value "Jane Doe". The `studentName` reference instead points to an empty String value, since that's what you coded to return from the `getName` method.

Fixing the code is a simple matter—change the `getName` method to return "Jane Doe" instead:

```
class Student {
    Student(String name) {
    }

    String getName() {
        return "Jane Doe";
    }
}
```

Recompile and rerun JUnit. Success! (See Figure 1.7.) There's something satisfying about seeing that green bar. Press the Run button again. The bar should stay green. Things are still good—but not that good. Right now, all students will be named Jane Doe. That's not good, unless you're starting an all-woman school that only accepts people with the name Jane Doe. Read on.

⁸Specifically, this example compares a String object to a String variable, or reference. Java *dereferences* the variable to obtain the String object at which it points and uses this for the comparison.

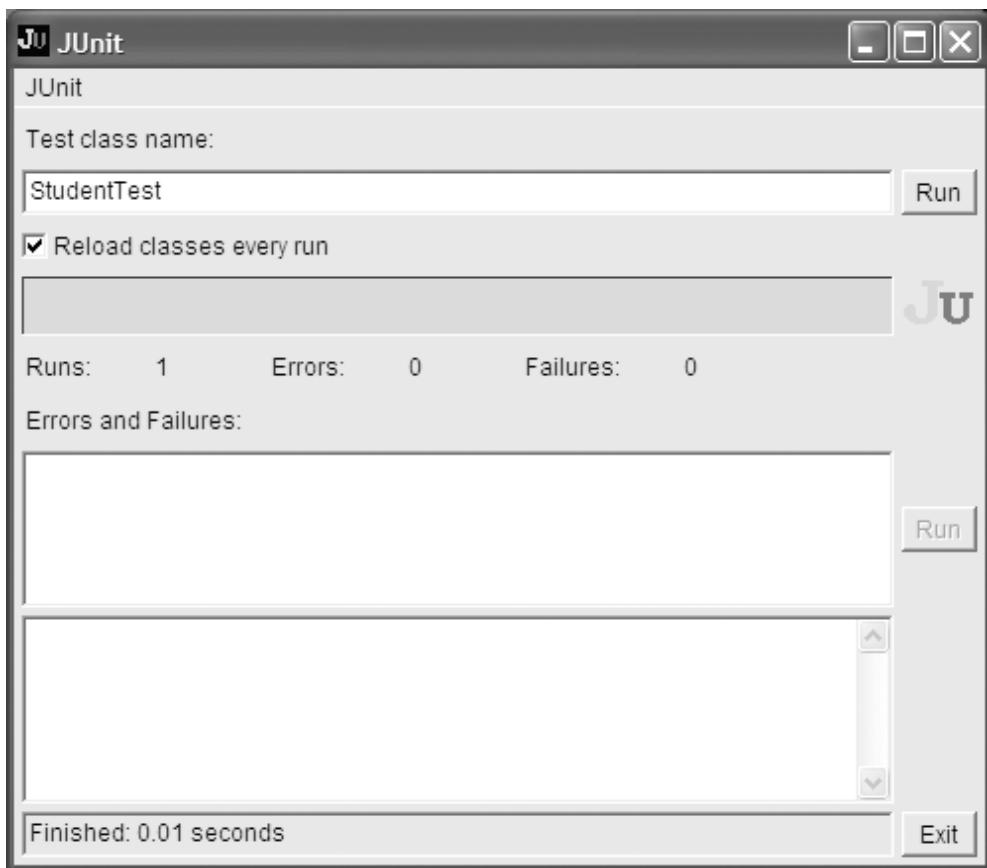


Figure 1.7 Success!

Instance Variables

You have built your first test and corresponding class. You used `StudentTest` to help you build the `Student` class in an incremental manner. The test can also be used to ensure that any future changes you make don't break something you already did.

Unfortunately, the code isn't quite right. If you were to build more students, they would all respond to the `getName` message by saying they were Jane Doe. You'll mature both `StudentTest` and `Student` in this section to deal with the problem.

You can prove the theory that every `Student` object will answer the name Jane Doe by elaborating on the `testCreate` method. Add code to create a second student object:

```
public void testCreate() {
    Student student = new Student("Jane Doe");
    String studentName = student.getName();
    assertEquals("Jane Doe", studentName);

    Student secondStudent = new Student("Joe Blow");
    String secondStudentName = secondStudent.getName();
    assertEquals("Joe Blow", secondStudentName);
}
```

The conceptual picture of the second student in memory is shown in Figure 1.8. Java finds space for the new Student object and stuffs it there. You won't know and shouldn't care where each object ends up. The important thing to get out of the memory diagram is that you now have references to two discrete objects in memory.

Start JUnit again. If you are running from the command line, start it as a background process (under Unix)⁹ or use the start command (under Windows).¹⁰ This will allow JUnit to run as a separate window, returning control to the command line. Whether you are using an IDE or running from the command line, you can keep the JUnit window open. Since JUnit reloads class files that are changed by compilation, you will not need to restart the JUnit GUI each time you make a change.¹¹

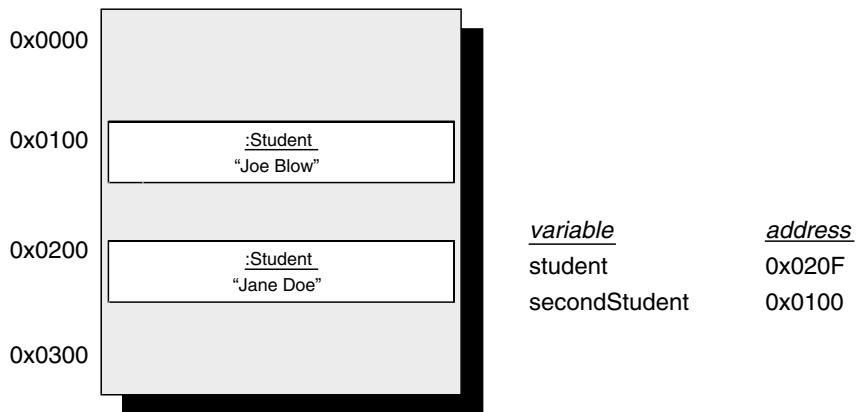


Figure 1.8 *Objects in Memory*

⁹Append an & to the command, under most systems

¹⁰Prepend the command with the word **start**. For example:

```
start java -cp .;c:\junit3.8.1\junit.jar junit.awtui.TestRunner StudentTest
```

¹¹For this to work, ensure that the checkbox labeled “Reload classes every run” is checked in the JUnit GUI.

The test fails:

```
junit.framework.ComparisonFailure: expected:<...oe Blow> but was:<...ane Doe>
```

Sure enough, since `getName` always returns "Jane Doe", the second `assertEquals` statement fails.

The problem is that you passed the student name to the constructor of the `Student` class, but the constructor does nothing with that name. Your code in the `Student` class is responsible for storing the name if anything is going to be able to refer to the name later.

You want the student's name to be an *attribute* of the student—a piece of information that is kept by the student as long as the student object is around. The most direct way to represent an attribute in Java is by defining it as a *field*, also known as an *instance variable*. You declare a field within the braces that delineate the start and end of a class. A field declaration can appear anywhere within the class as long as it appears outside of the methods defined in that class. But according to convention, you should place field declarations near either the start or the end of the class.

Like a local variable, a field has a type. Define the field `myName` to be of the type `String` within the `Student` class as shown here:

```
class Student {  
    String myName;  
  
    Student(String name) {  
    }  
  
    String getName() {  
        return "Jane Doe";  
    }  
}
```

Instance
Variables

In the constructor body, assign the constructor's parameter `name` to `myName`:

```
Student(String name) {  
    myName = name;  
}
```

Finally, return the field from the `getName` method (instead of the `String` literal "Jane Doe").

```
String getName() {  
    return myName;  
}
```

JUnit should still be open, showing the comparison failure and red bar. Rerun the tests by pressing the Run button. The bar should turn green.

Take a look again at the test method:

```
public void testCreate() {
    Student student = new Student("Jane Doe");
    String studentName = student.getName();
    assertEquals("Jane Doe", studentName);

    Student secondStudent = new Student("Joe Blow");
    String secondStudentName = secondStudent.getName();
    assertEquals("Joe Blow", secondStudentName);
}
```

Summarizing the Test

The test demonstrates, and thus documents, how `Student` instances can be created with discrete names. To further bolster this assertion, add a line to the end of the test to ensure that you can still retrieve the proper student name from the first student object created:

```
public void testCreate() {
    Student student = new Student("Jane Doe");
    String studentName = student.getName();
    assertEquals("Jane Doe", studentName);

    Student secondStudent = new Student("Joe Blow");
    String secondStudentName = secondStudent.getName();
    assertEquals("Joe Blow", secondStudentName);

    assertEquals("Jane Doe", student.getName());
}
```

Note that instead of assigning the result of the message send—the result of calling `student.getName()`—to a local variable, you instead substituted it directly as the second parameter of `assertEquals`.

Rerun the test (after compiling). The success of the test shows that `student` and `secondStudent` refer to two distinct objects.

Summarizing the Test

Let's summarize what the test is supposed to be doing, line by line:

<code>Student student = new Student ("Jane Doe");</code>	Create a student whose name is Jane Doe. Store it locally.
<code>String studentName = student .getName();</code>	Ask for the name of the student and store it in a local variable.
<code>assertEquals("Jane Doe", studentName);</code>	Make sure that the name of the student returned was Jane Doe.

That's a lot of things to know in order to understand a few short lines of code. However, you have already seen the foundation for a large part of Java

programming. In well-designed object-oriented Java coding, most statements involve either creating new objects, sending messages to other objects, or assigning object addresses (whether the objects are created via `new` or are returned from a message send) to an object reference.

Refactoring

One of the main problems in software development is the high cost of maintaining code. Part of the reason is that code quickly becomes “crufty” or messy, as the result of rushed efforts or just plain carelessness. Your primary job in building software is to get it to work, a challenge that you will resolve by writing tests before you write the code. Your secondary job is to ensure that the code stays clean. You do this through two mechanisms:



1. ensure that there is no duplicate code in the system, and
2. ensure that the code is clean and expressive, clearly stating the intent of the code

Throughout Agile Java, you will pause frequently to reflect on the code just written. Anything that does not demonstrate adherence to these two simple rules must be reworked, or *refactored*, immediately. Even within a “perfect” design—an unrealistic goal—the poor implementation of code can cause costly headaches for those trying to modify it.

The more you can continually craft the code as you go, the less likely you are to hit a brick wall of code so difficult that you cannot fix it cheaply. Your rule of thumb should be to never leave the code in a worse state than when you started working on it.

Even within your small example so far, there is some less-than-ideal code. Start on the path to cleaning up the code by taking a look at the test:

```
public void testCreate() {
    Student student = new Student("Jane Doe");
    String studentName = student.getName();
    assertEquals("Jane Doe", studentName);

    Student secondStudent = new Student("Joe Blow");
    String secondStudentName = secondStudent.getName();
    assertEquals("Joe Blow", secondStudentName);

    assertEquals("Jane Doe", student.getName());
}
```

The first step is to eliminate the unnecessary local variables `studentName` and `secondStudentName`. They don’t add anything to the understanding of the method

Refactoring

and can be replaced with simple queries to the student object, as in the very last `assertEquals`.

When you are done making this change, recompile and rerun your test in JUnit to ensure you haven't broken anything. Your code should look like this:

```
public void testCreate() {
    Student student = new Student("Jane Doe");
    assertEquals("Jane Doe", student.getName());

    Student secondStudent = new Student("Joe Blow");
    assertEquals("Joe Blow", secondStudent.getName());

    assertEquals("Jane Doe", student.getName());
}
```

Refactoring

The second step: It is considered poor programming practice to embed String literals throughout your code. One reason is that the code can be difficult to follow if it is not clear what each String literal represents.

In this example, you also are breaking the rule against code duplication. Each of the two String literals appears twice within the test method. If you change one String, you have to change the other. This is more work, and it also means there is a possibility that you will introduce a defect in your code by changing one and not the other.

One way to eliminate this redundancy (and add a bit of expressiveness to the code) is to replace the String literals with String constants.

```
final String firstStudentName = "Jane Doe";
```

This statement creates a reference named `firstStudentName` of the type `String` and assigns it an initial value of the String literal "Jane Doe".

The keyword `final` at the beginning of the statement indicates that the `String` reference cannot be changed—no other object can be assigned to the reference. You are never required to specify `final`, but it is considered good form and helps document the intent that `firstStudentName` is acting as a constant. You will learn other uses for `final` later.

Now that you have declared the constant, you can replace the String literals with it.

```
final String firstStudentName = "Jane Doe";
Student student = new Student(firstStudentName);
assertEquals(firstStudentName, student.getName());
...
assertEquals(firstStudentName, student.getName());
```

Compile and rerun your test to ensure that you haven't inadvertently broken anything.

Apply a similar *refactoring* to the other String literal. In addition, change the local variable name student to firstStudent in order to be consistent with the variable name secondStudent. Between each small change, compile and use JUnit to ensure that you haven't broken anything. When finished, your code should look like this:

```
public void testCreate() {
    final String firstStudentName = "Jane Doe";
    Student firstStudent = new Student(firstStudentName);
    assertEquals(firstStudentName, firstStudent.getName());

    final String secondStudentName = "Joe Blow";
    Student secondStudent = new Student(secondStudentName);
    assertEquals(secondStudentName, secondStudent.getName());
    assertEquals(firstStudentName, firstStudent.getName());
}
```

this

The final assertEquals proves your understanding of the way Java works rather than the functionality that you are trying to build. You would not likely keep this assertion around in a production system. You may choose to keep it or delete it. If you delete it, make sure you recompile and rerun your test!

Your development cycle is now:

- Write a small test to assert some piece of functionality.
- Demonstrate that the test fails.
- Write a small bit of code to make this test pass.
- Refactor both the test and code, eliminating duplicate concepts and ensuring that the code is expressive.

This cycle will quickly become an ingrained, natural flow of development.

this

Look at the Student class code to see if it can be improved.

```
class Student {
    String myName;

    Student(String name) {
        myName = name;
    }

    String getName() {
        return myName;
    }
}
```

The code appears clean, but naming the field `myName` is oh so academic. Show your professionalism by using a better name. A first thought might be to call the field `studentName`. However, that introduces duplication in the form of name redundancy—it's clear that the field represents a student's name, since it is defined in the class `Student`.

Also, you generally name your `get` methods after the fields, so the redundancy would become very apparent when you coded a statement like:

```
student.getStudentName();
```

How about simply `name`?

The problem with using `name` as the field name is that it clashes with the *formal parameter* name of the `Student` constructor—both would be called `name`. Try it, though, and see what happens.

```
class Student {
    String name;

    Student(String name) {
        name = name;
    }

    String getName() {
        return name;
    }
}
```

This should pass compilation (you might see a warning). Run your test, however, and it will fail:

```
junit.framework.ComparisonFailure: expected:<Jane Doe> but was:<null>
```

Why? Well, part of the problem is that the Java compiler allows you to name fields the same as formal parameters, or even the same as local variables. When the code is compiled, Java tries to figure out which `name` you meant. The resolution the compiler makes is to use the most locally defined `name`, which happens to be the name of the formal parameter. The statement

```
name = name;
```

thus just results in the object stored in the parameter being assigned to itself. This means that nothing is getting assigned to the instance variable (field) called `name`. Field references that are not assigned an object have a special value of `null`, hence the JUnit message:

```
expected:<Jane Doe> but was:<null>
```

There are two ways to ensure that the value of the formal parameter is assigned to the field: Either ensure both variables have different names or use

this

the Java keyword `this` to differentiate them. Using `this` is the most common approach.

The first approach means you must rename either the parameter or the field. Java programmers use many differing conventions to solve this naming issue. One is to rename the formal parameter so that it uses a single letter or is prefixed with an article. For example, `name` could be renamed to `n` or `aName`. Another common choice is to rename the field by prefixing it with an underscore: `_name`. This makes use of the field stand out like a sore thumb, something that can be very valuable when trying to understand code. There are other schemes, such as prefixing the argument name with the article `a` or an `(aName)`.

The second approach for disambiguating the two is to use the same name for both, but where necessary refer to the field by prefixing it with the Java keyword `this`.

```
class Student {
    String name;

    Student(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }
}
```

private

The `this` keyword represents a reference to the current object—the object for which the code is currently executing. The statement in the above example assigns the value of the formal parameter `name` to the field `name`.

Make sure your test now passes after making this change. From now on, remember to recompile after you change the code and before running your tests—there will be few reminders in the rest of the book.

private

Java allows you to access fields of an object, just like you can call a method:

```
public void testCreate() {
    final String firstStudentName = "Jane Doe";
    Student firstStudent = new Student(firstStudentName);
    assertEquals(firstStudentName, firstStudent.getName());
```

```

final String secondStudentName = "Joe Blow";
Student secondStudent = new Student(secondStudentName);
assertEquals(secondStudentName, secondStudent.getName());

assertEquals(firstStudentName, firstStudent.name);
}

```

If you run this test, it will pass. However, it demonstrates particularly poor object-oriented coding style.



Do not directly expose fields to other objects.

Private

Suppose you want to design the `Student` class so that a student's name is *immutable*—it cannot be changed once you've created a `Student` object. The following test code demonstrates how allowing other objects to access your attributes can be a bad idea:

```

final String firstStudentName = "Jane Doe";
Student firstStudent = new Student(firstStudentName);
firstStudent.name = "June Crow";
assertEquals(firstStudentName, firstStudent.getName());

```

The test shows that `Student` client code—code that is interacting with the `Student` object—can directly modify the `String` stored in the `name` instance variable. While this doesn't seem like a terrible affront, it eliminates any control you have over clients changing your object's data. If you want to allow client code to change the `Student`'s name, you can create a method for the client to use. For example, you could create a method called `setName` that takes a new name `String` as parameter. Within the `setName` method, you could include code for any additional control you needed.

Make the above change to `StudentTest` and demonstrate for yourself that the test fails.

To protect your fields—to hide them—you should designate them as `private`. Change the `Student` class to hide the `name` field.

```

class Student {
    private String name;
    ...
}

```

After doing so, other code that attempts to access the field will not even compile. Since the code in `Student` test refers to the `name` field directly:

```
assertEquals(firstStudentName, firstStudent.name);
```

you will receive a compilation error:

```

name has private access in Student
    assertEquals(firstStudentName, firstStudent.name);
               ^
1 error

```

Remove the offending line, recompile, and retest.

The other interest in making fields private is to reinforce the notion of object orientation and encapsulation: An object-oriented system is about behavior, not data. You want to accomplish things by sending messages. You also want to encapsulate implementation details. Perhaps later you want to change the `Student` class to store both first and last names and change the `getName` method to return the combined name parts. In that case, code that directly accesses the `name` field would no longer be valid.

As with any rule, there are exceptions. There are at least a couple legitimate reasons to not designate a field as `private`. (You'll learn about these much later.)

Naming
Conventions

Naming Conventions

You should have noticed a naming pattern in the Java code you have written so far. Most of the elements in Java that you have already learned about—fields, formal parameters, methods, and local variables—are named in a similar way. This convention is sometimes referred to as *camel case*.¹² When following the camel case naming pattern, you comprise a name, or *identifier*, of words concatenated directly together. You begin each word in the identifier, except for the first word, with a capital letter.

You should name fields using nouns. The name should describe what the field is used for or what it represents, not how it is implemented. Prefixes and suffixes that belie the type of the field are unnecessary and should be avoided. Examples of field names to avoid are `firstNameString`, `trim`, and `sDescription`.

Examples of good field names include `firstName`, `trimmer`, `description`, `name`, `mediaController`, and `lastOrderPlaced`.

Methods are generally either actions or queries: Either you are sending a message to tell an object to do something or you are asking an object for some information. You should use verbs to name action methods. You should also use verbs for query method names. The usual Java convention is to prefix the name of the attribute being retrieved by the word `get`, as in `getNumberOfStudents` and `getStudent`. I will discuss some exceptions to this rule later.

¹²Imagine that the letters represent the view of a camel from the side. The upper letters are the humps. For an interesting discussion of the term, see <http://c2.com/cgi/wiki?CamelCase>. You may hear other names for camel case, such as “mixed case.”

Examples of good method names include `sell`, `cancelOrder`, and `isDoorClosed`.

Name classes using *upper camel case*—camel case where the first letter of the identifier is uppercased.¹³ You should almost always use nouns for class names—objects are abstractions of things. Do not use plural words for class names. A class is used to create a single object at a time; for example, a `Student` class can create a `Student` object. When creating collections of objects, as you will learn to do later, continue to use a nonplural name: `StudentDirectory` instead of `Students`. From a `StudentDirectory` class, you can create a `StudentDirectory` object. From a `Students` class, you could create a `Students` object, but it sounds awkward and leads to similarly awkward code phrases.

Examples of good class names include `Rectangle`, `CompactDisc`, `LaundryList`, and `HourlyPayStrategy`.

In terms of good object-oriented design, you will find that design impacts the ability to name things. A well-designed class does one thing of importance and only that one thing.¹⁴ Classes generally should not do a multitude of things. For example, if you have a class that cuts paychecks, prints a report of all the checks, and calculates chargebacks to various departments, it would be hard to come up with a name that succinctly describes the class. Instead, break the class up into three separate classes: `CheckWriter`, `PayrollSummaryReport`, and `ChargebackCalculator`.

You can use numbers in an identifier, but you cannot use a number as its first character. Avoid special characters—the Java compiler disallows many of them. Avoid underscores (`_`), particularly as separators between words. There are exceptions: As mentioned earlier, some developers prefer to prefix their fields with underscores. Also, class constants (to be introduced later) usually include underscores.

Avoid abbreviations. Clarity in software is highly valued. Take the time to type the few extra characters. Doing so is far cheaper than increasing the time it takes for a future reader to comprehend your code. Replace names such as `cust` and `num` with the more expressive `custard` and `numerology`.¹⁵ Modern IDEs provide support for quickly renaming things in your code and for auto-completing things you are typing, so you have no excuse to use cryptic names. Abbreviations are acceptable if you use the abbreviation in common speech, such as `id` or `tvAdapter`.

¹³Camel case is thus sometimes called lower camel case to distinguish it from upper camel case.

¹⁴A guideline known as the Single-Responsibility Principle [Martin2003].

¹⁵I'll bet you thought they stood for `customer` and `number`. See how easy it is to be led astray?

Remember that Java is case sensitive. This means that `stuDent` represents a different name than `student`. It is bad form to introduce this sort of confusion in naming, however.

These naming conventions indicate that there are commonly accepted coding standards for things that aren't necessarily controlled by the compiler. The compiler will allow you to use hideous names that will irritate your fellow developers. Most shops wisely adopt a common standard to be followed by all developers. The Java community has also come to a higher level of agreement on such standards than, say, the C++ community. Sun's coding conventions for Java, located at <http://java.sun.com/docs/codeconv>, is a good, albeit incomplete and outdated, starting point for creating your own coding standards. Other style/standards books exist, including *Essential Java Style*¹⁶ and *Elements of Java Style*.¹⁷

Whitespace

Whitespace

The layout of code is another area where you and your team should adhere to standards. Whitespace includes spaces, tab characters, form feeds, and new lines (produced by pressing the enter key). Whitespace is required between certain elements and is optional between others. For example, there must be at least one space between the keyword `class` and the name of the class in a class declaration:

```
class Student
```

And the following spacing is allowable (but avoided):

```
String studentName = student . getName();
```

The Java compiler ignores extra whitespace. You should use spaces, tabs, and blank lines to judiciously organize your code. This goes a long way toward allowing easy future understanding of the code.

The examples in this book provide a consistent, solid and generally accepted way of formatting Java code. If your code looks like the examples, it will compile. It will also meet the standards of most Java development shops. You will want to decide on things such as whether to use tabs or spaces for indenting and how many characters to indent by (usually three or four).

¹⁶[Langr2000].

¹⁷[Vermeulen2000].

Exercises

Exercises appear at the end of lesson. Many of the exercises have you build pieces of an application that plays the game of chess. If you are unfamiliar with the rules of chess, you can read them at <http://www.chessvariants.com/d.chess/chess.html>.

Exercises

1. As with Student, you will start simply by creating a class to represent a pawn. First, create an empty test class named PawnTest. Run JUnit against PawnTest and observe that it fails, since you haven't written any test methods yet.
2. Create a test method named `testCreate`. Ensure that you follow the correct syntax for declaring test methods.
3. Add code to `testCreate` that instantiates a Pawn object. Ensure that you receive a compile failure due to the nonexistent class. Create the Pawn class and demonstrate a clean compile.
4. Assign the instantiated Pawn object to a local variable. Ask the pawn for its color. Code a JUnit assertion to show that the default color of a pawn is represented by the string "white". Demonstrate test failure, then add code to Pawn to make the test pass.
5. In `testCreate`, create a second pawn, passing the color "black" to its constructor. Assert that the color of this second pawn is "black". Show the test failure, then make the test pass. Note: Eliminate the default constructor—require that clients creating Pawn objects pass in the color. The change will impact the code you wrote for Exercise #4.
6. In `testCreate`, create constants for the Strings "white" and "black". Make sure you rerun your tests.

Lesson 2

Java Basics

In this lesson you will:

- use the numeric type `int` to count the number of students
- use the Java collection class `java.util.ArrayList` to store many students
- understand default constructors
- learn how to use the J2SE API documentation to understand how to use `java.util.ArrayList`
- restrict the `java.util.ArrayList` collection to contain only Student objects
- create a TestSuite to test more than one class
- learn about packages and the `import` statement
- understand how to define and use class constants
- use date and calendar classes from the system library
- learn the various types of comments that Java allows
- generate API documentation for your code with javadoc

CourseSession

CourseSession



Schools have courses, such as Math 101 and Engl 200, that are taught every semester. Basic course information, such as the department, the number, the number of credits, and the description of the course, generally remains the same from semester to semester.



A course session represents a specific occurrence of a course. A course session stores the dates it will be held and the person teaching

it, among other things. It also must retain an enrollment, or list of students, for the course.

You will define a CourseSession class that captures both the basic course information and the enrollment in the session. As long as you only need to work with the CourseSession objects for a single semester, no two Course Sessions should need to refer to the same course. Once two CourseSession objects must exist for the same course, having the basic course information stored in both CourseSession objects is redundant. For now, multiple sessions is not a consideration; later you will clean up the design to support multiple sessions for a single course.

Create CourseSessionTest.java. Within it, write a test named `testCreate`. Like the `testCreate` method in `StudentTest`, this test method will demonstrate how you create CourseSession objects. A creation test is always a good place to get a handle on what an object looks like just after it's been created.

CourseSession

```
public class CourseSessionTest extends junit.framework.TestCase {
    public void testCreate() {
        CourseSession session = new CourseSession("ENGL", "101");
        assertEquals("ENGL", session.getDepartment());
        assertEquals("101", session.getNumber());
    }
}
```

The test shows that a CourseSession can be created with a course department and number. The test also ensures that the department and number are stored correctly in the CourseSession object.

To get the test to pass, code CourseSession like this:

```
class CourseSession {
    private String department;
    private String number;

    CourseSession(String department, String number) {
        this.department = department;
        this.number = number;
    }

    String getDepartment() {
        return department;
    }

    String getNumber() {
        return number;
    }
}
```

So far you've created a `Student` class that stores student data and a `CourseSession` class that stores course data. Both classes provide "getter" methods to allow other objects to retrieve the data.

However, data classes such as `Student` and `CourseSession` aren't terribly interesting. If all there was to object-oriented development was storing data and retrieving it, systems wouldn't be very useful. They also wouldn't be object-oriented. Remember that object-oriented systems are about modeling behavior. That behavior is effected by sending messages to objects to get them to do something—not to ask them for data.

But, you've got to start somewhere! Plus, you wouldn't be able to write assertions in your test if you weren't able to ask objects what they look like.

Enrolling Students

 Courses don't earn any revenue for the school unless students enroll in them. Much of the student information system will require you to be able to work with more than one student at a time. You will want to store groups, or collections, of students and later execute operations against the students in these collections.

`CourseSession` will need to store a new attribute—a collection of `Student` objects. You will want to bolster your `CourseSession` creation test so that it says something about this new attribute. If you have just created a new course session, you haven't yet enrolled any students in it. What can you assert against an empty course session?

Modify `testCreate` so that it contains the **bolded** assertion:

```
public void testCreate() {  
    CourseSession session = new CourseSession("ENGL", "101");  
    assertEquals("ENGL", session.getDepartment());  
    assertEquals("101", session.getNumber());  
    assertEquals(0, session.getNumberOfStudents());  
}
```

int

The new assertion verifies that the number of students enrolled in a brand-new session should be `0` (zero). The symbol `0` is a *numeric literal* that represents the integer zero. Specifically, it is an integer literal, known in Java as an `int`.

int

Add the method `getNumberOfStudents` to `CourseSession` as follows:

```
class CourseSession {
    ...
    int getNumberOfStudents() {
        return 0;
    }
}
```

(The ellipses represent the instance variables, constructor code, and getter methods that you have already coded.) The return type of `getNumberOfStudents` is specified as `int`. The value returned from a method must match the return type, and in this method it does—`getNumberOfStudents` returns an `int`. The `int` type allows variables to be created that store integer values from `-2,147,483,648` to `2,147,483,647`.

Numbers in Java are not objects like `String` literals are. You cannot send messages to numbers, although numbers can be passed as parameters along with messages just like `Strings` can. Basic arithmetic support in Java is provided syntactically; for many other operations, support is provided by system libraries. You will learn about similar non-object types later in Agile Java. As a whole, these non-object types are known as *primitive types*.

You have proved that a new `CourseSession` object is initialized properly, but you haven't shown that the class can enroll students properly. Create a second test method `testEnrollStudents` that enrolls two students. For each, create a new `Student` object, enroll the student, and ensure that the `CourseSession` object reports the correct number of students.

```
public class CourseSessionTest extends junit.framework.TestCase {
    public void testCreate() {
        ...
    }

    public void testEnrollStudents() {
        CourseSession session = new CourseSession("ENGL", "101");

        Student student1 = new Student("Cain DiVoe");
        session.enroll(student1);
        assertEquals(1, session.getNumberOfStudents());

        Student student2 = new Student("Coralee DeVaughn");
        session.enroll(student2);
        assertEquals(2, session.getNumberOfStudents());
    }
}
```

How do you know that you need a method named `enroll`, and that it should take a `Student` object as a parameter? Part of what you are doing in a test method is designing the *public interface* into a class—how developers

int

will interact with the class. Your goal is to design the class such that the needs of developers who want use it are met in as simple a fashion as possible.

The simplest way of getting the second assertion (that the number of students is two) to pass would be to return 2 from the `getNumberOfStudents` method. However, that would break the first assertion. So you must somehow track the number of students inside of `CourseSession`. To do this, you will again introduce a field. Any time you know you need information to be stored, you will likely use fields to represent object *state*. Change the `CourseSession` class to look like this:

```
class CourseSession {  
    ...  
    private int numberOfStudents = 0;  
    ...  
    int getNumberOfStudents() {  
        return numberOfStudents;  
    }  
  
    void enroll(Student student) {  
        numberOfStudents = numberOfStudents + 1;  
    }  
}
```

int

The field to track the student count is named `numberOfStudents`, it is `private` per good practice, and it is of the type `int`. It is also assigned an *initial value* of 0. When a `CourseSession` object is instantiated, *field initializers* such as this initialization of `numberOfStudents` are executed. Field initializers are executed prior to the invocation of code in the constructor.

The method `getNumberOfStudents` now returns the field `numberOfStudents`, instead of the `int` literal 0.

Each time the `enroll` method is called, you should increment the number of students by one. The single line of code in the `enroll` method accomplishes this:

```
numberOfStudents = numberOfStudents + 1;
```

The + sign and many other mathematical operators are available for working with variables of the `int` type (as well as other numeric types to be discussed later). The expression on the right hand side of the = sign takes whatever value is currently stored in `numberOfStudents` and adds 1 to it. Since the `numberOfStudents` field appears to the left of the = sign, the resultant value of the right-hand side expression is assigned back into it. Bumping up a variable's value by one, as in this example, is a common operation known as

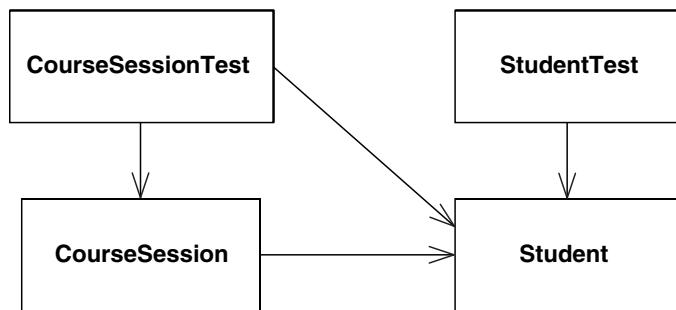


Figure 2.1 CourseSession and Student Class Diagram

incrementing the variable. There are other ways to increment a variable that will be discussed later.

It may seem odd that `numberOfStudents` appears on both sides of the assignment operator. Remember that the Java VM always executes the right hand side of an assignment statement first. It calculates a result using the expression to the right, and assigns this result to the variable on the left.

Note that the `enroll` method has a return type `void`, meaning that it returns nothing to the message sender.

The class diagram in Figure 2.1 shows the existing structure of your system so far.

Conceptually, a course session should be able to hold several students. In reality—in the code—the course session holds references to no student objects. It only holds a count of students. Later, when you modify the Course Session class to actually store Student references, the UML diagram will be modified to show that there is a one-to-many relationship between Course Session and Student.

`CourseSession` depends on `Student`, since the `enroll` method can take a `Student` object as a parameter. In other words, you would not be able to compile the `CourseSession` class if the `Student` class did not exist.

Figure 2.1 will be the last class diagram where I show every test class. Since you are doing test-driven development, future diagrams will imply the existence of a test class for each production class, unless otherwise noted.

Initialization

Initialization

In the last section, you introduced the `numberOfStudents` field and initialized it to `0`. This initialization is technically not required—`int` fields are initialized to `0` by default. Explicitly initializing a field in this manner, while unnecessary, is useful to help explain the intent of the code.

For now, you have two ways to initialize fields: You can initialize at the field level or you can initialize in a constructor. You could have initialized `numberOfStudents` in the `CourseSession` constructor:

```
class CourseSession {
    private String department;
    private String number;
    private int numberOfStudents;

    CourseSession(String department, String number) {
        this.department = department;
        this.number = number;
        numberOfStudents = 0;
    }
    ...
}
```

There is no hard-and-fast rule about where to initialize. I prefer initializing at the field level when possible—having the initialization and declaration in one place makes it easier to follow the code. Also, as you will learn later in this lesson, you can have more than one constructor; initializing at the field level saves you from duplicate initialization code in each constructor.

You will encounter situations where you cannot initialize code at the field level. Initializing in the constructor may be your only alternative.

Default Constructors

Default Constructors

You may have noted that neither of the test classes, `StudentTest` and `CourseSessionTest`, contains a constructor. Often you will not need to explicitly initialize anything, so the Java compiler does not require you to define a constructor. If you do not define any constructors in a class,¹ Java provides a default, no-argument constructor. For `StudentTest`, as an example, it is as if you had coded an empty constructor:

```
class StudentTest extends junit.framework.TestCase {
    StudentTest() {
    }
    ...
}
```

The use of default constructors also implies that Java views constructors as essential elements to a class. A constructor is required in order for Java to initialize a class, even if the constructor contains no additional initialization

¹The Java compiler does allow you to define multiple constructors in a single class; this will be discussed later.

code. If you don't supply a constructor, the Java compiler puts it there for you.

Suites

In the last section, you introduced a second test class, `CourseSessionTest`. Going forward, you could decide to run JUnit against either `CourseSessionTest` or `StudentTest`, depending on which corresponding production class you changed. Unfortunately, it is very possible for you to make a change in the `Student` class that breaks a test in `CourseSessionTest`, yet all the tests in `StudentTest` run successfully.

You could run the tests in `CourseSessionTest` and then run the tests in `StudentTest`, either by restarting JUnit each time or by retyping the test class name in JUnit or even by keeping multiple JUnit windows open. None of these solutions is scalable: As you add more classes, things will quickly become unmanageable.

Instead, JUnit allows you to build *suites*, or collections of tests. Suites can also contain other suites. You can run a suite in the JUnit test runner just like any test class.

Create a new class called `AllTests` with the following code:

```
public class AllTests {  
    public static junit.framework.TestSuite suite() {  
        junit.framework.TestSuite suite =  
            new junit.framework.TestSuite();  
        suite.addTestSuite(StudentTest.class);  
        suite.addTestSuite(CourseSessionTest.class);  
        return suite;  
    }  
}
```

If you start JUnit with the class name `AllTests`, it will run all of the tests in both `CourseSessionTest` and `StudentTest` combined. From now on, run `AllTests` instead of any of the individual class tests.

The job of the `suite` method in `AllTests` is to build the suite of classes to be tested and return it. An object of the type `junit.framework.TestSuite` manages this suite. You add tests to the suite by sending it the message `addTestSuite`. The parameter you send with the message is a *class literal*. A class literal is comprised of the name of the class followed by `.class`. It uniquely identifies the class, and lets the class definition itself be treated much like any other object.

Each time you add a new test class, you will need to remember to add it to the suite built by `AllTests`. This is an error-prone technique, as it's easy to for-

get to update the suite. In Lesson 12, you'll build a better solution: a tool that generates and executes the suite for you.

The code in AllTests also introduces the concept of *static methods*, something you will learn about in Lesson 4. For now, understand that you must declare the suite method as static in order for JUnit to recognize it.

The SDK and java.util.ArrayList

Your CourseSession class tracks the count of students just fine. However, you and I both know that it's just maintaining a counter and not holding onto the students that are being enrolled. For the testEnrollStudents method to be complete, it will need to prove that the CourseSession object is retaining the actual student objects.

One possible solution is to ask the CourseSession for a list of all the enrolled students, then check the list to ensure that it contains the expected students. Modify your test to include the lines shown below in bold.

The SDK and
java.util.ArrayList

```
public void testEnrollStudents() {
    CourseSession session = new CourseSession("ENGL", "101");

    Student student1 = new Student("Cain DiVoe");
    session.enroll(student1);
    assertEquals(1, session.getNumberOfStudents());
    java.util.ArrayList<Student> allStudents = session.getAllStudents();
    assertEquals(1, allStudents.size());
    assertEquals(student1, allStudents.get(0));

    Student student2 = new Student("Coralee DeVaughn");
    session.enroll(student2);
    assertEquals(2, session.getNumberOfStudents());
    assertEquals(2, allStudents.size());
    assertEquals(student1, allStudents.get(0));
    assertEquals(student2, allStudents.get(1));
}
```

The first line of newly added test code:

```
java.util.ArrayList<Student> allStudents = session.getAllStudents();
```

says that there will be a method on CourseSession named `getAllStudents`. The method will have to return an object of the type `java.util.ArrayList<Student>`, since that's the type of the variable to which the result of `getAllStudents` is to be assigned. A type appearing in this form—a class name followed by a parameter type within angle brackets (< and >) is known as a *parameterized type*. In the example, the parameter `Student` of

`java.util.ArrayList` indicates that the `java.util.ArrayList` is *bound* such that it can only contain `Student` objects.

The class `java.util.ArrayList` is one of thousands available as part of the Java SDK class library. You should have downloaded and installed the SDK documentation so that it is available on your machine. You can also browse the documentation online at Sun's Java site. My preference is to have the documentation available locally for speed and accessibility reasons.

Pull up the documentation and navigate to the Java 2 Platform API Specification. You should see something like Figure 2.2.

The Java API documentation will be your best friend, unless you're doing *pair programming*,² in which case it will be your second-best friend. It is divided into three frames. The upper left frame lists all the *packages* that are available in the library. A package is a group of related classes.

The lower left frame defaults to showing all the classes in the library. Once a package is selected, the lower left frame shows only the classes contained within that package. The frame to the right that represents the remainder of the page shows detailed information on the currently selected package or class.

Scroll the package frame until you see the package named `java.util` and select it. The lower left pane should now show a list of *interfaces*, classes, and

The SDK and
`java.util.ArrayList`

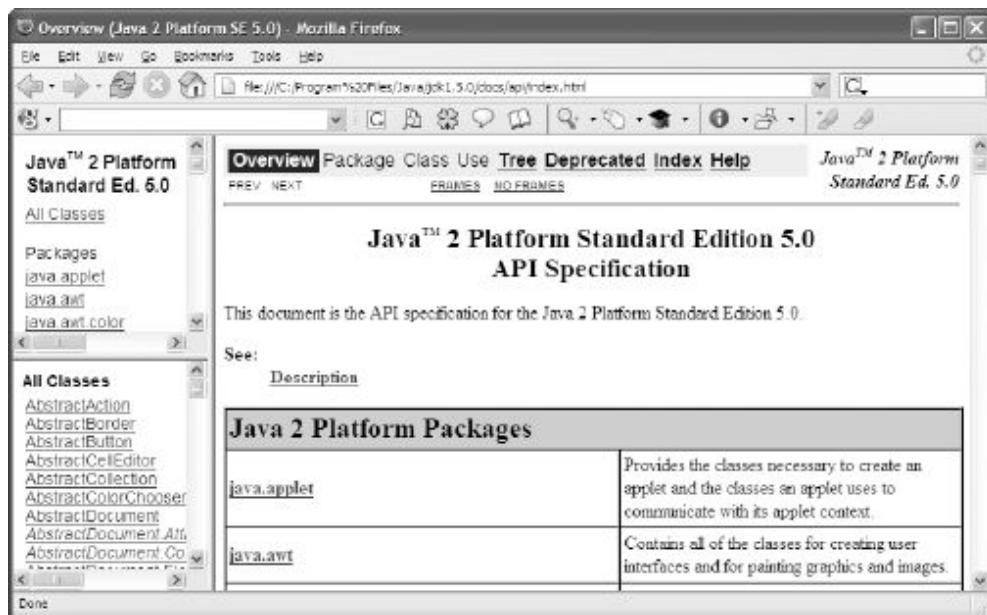


Figure 2.2 Java 2 Platform API Specification

²A technique whereby members of a development team work in dynamic pairs of developers who jointly construct code.

exceptions. I will explain interfaces and exceptions later. For now, scroll down until you see the class named ArrayList and select it.

The package java.util contains several utility classes that you will use often in the course of Java development. The bulk of the classes in the package support something called the Collections Framework. The Collections Framework is a consistent sublibrary of code used to support standard data structures such as lists, linked lists, sets, and hash tables. You will use collections over and over again in Java to work with groupings of related objects.

The main pane (to the right) shows all of the detailed information on the class java.util.ArrayList. Scroll down until you see the Method Summary. The Method Summary shows the methods implemented in the java.util.ArrayList class. Take a few minutes to read through the methods available. Each of the method names can be clicked on for detailed information regarding the method.

You will use three of the java.util.ArrayList methods as part of the current exercise: add, get, and size. One of them, size, is already referenced by the test. The following line of code asserts that there is one object in the java.util.ArrayList object.

```
assertEquals(1, allStudents.size());
```

Adding Objects

The next line of new code asserts that the first element in allStudents is equal to the student that was enrolled.

```
assertEquals(student, allStudents.get(0));
```

According to the API documentation, the get method returns the element at an arbitrary position in the list. This position is passed to the get method as an index. Indexes are zero-based, so get(0) returns the first element in the list.

Adding Objects

The add method is documented in the Java SDK API as taking an Object as parameter. If you click on the parameter Object in the API docs, you will see that it is actually java.lang.Object.

You might have heard that Java is a “pure” object-oriented language, where “everything is an object.”³ The class java.lang.Object is the mother of all classes defined in the Java system class library, as well as of any class that

³A significant part of the language is not object-oriented. This will be discussed shortly.

you define, including Student and StudentTest. Every class inherits from java.lang.Object, either directly or indirectly. StudentTest inherits from junit.framework.TestCase, and junit.framework.TestCase inherits from java.lang.Object.

This inheritance from java.lang.Object is important: You'll learn about several core language features that depend on java.lang.Object. For now, you need to understand that String inherits from java.lang.Object, as does Student, as does any other class you define. This inheritance means that String objects and Student objects are also java.lang.Object objects. The benefit is that String and Student objects can be passed as parameters to any method that takes a java.lang.Object as a parameter. As mentioned, the add method takes an instance of java.lang.Object as a parameter.

Even though you can pass any type of object to a java.util.ArrayList via its add method, you don't usually want to do that. In the CourseSession object, you know that you only want to be able to enroll Students. Hence the parameterized type declaration. One of the benefits to restricting the java.util.ArrayList via a parameterized type is that it protects other types of objects from inadvertently being added to the list.

Suppose someone is allowed to add, say, a String object to the list of students. Subsequently, your code asks for the list of students using getAllStudents and retrieves the String object from the list using the get method. When you attempt to assign this object to a Student reference, the Java VM will choke, generating an error condition. Java is a *strongly typed* language, and it will not allow you to assign a String to a Student reference.

The following changes to CourseSession (appearing in bold) will make the test pass:

```
class CourseSession {  
    ...  
    private java.util.ArrayList<Student> students =  
        new java.util.ArrayList<Student>();  
    ...  
    void enroll(Student student) {  
        numberofStudents = numberofStudents + 1;  
        students.add(student);  
    }  
  
    java.util.ArrayList<Student> getAllStudents() {  
        return students;  
    }  
}
```

A new field, students, is used to store the list of all students. It is initialized to an empty java.util.ArrayList object that is bound to contain only Student

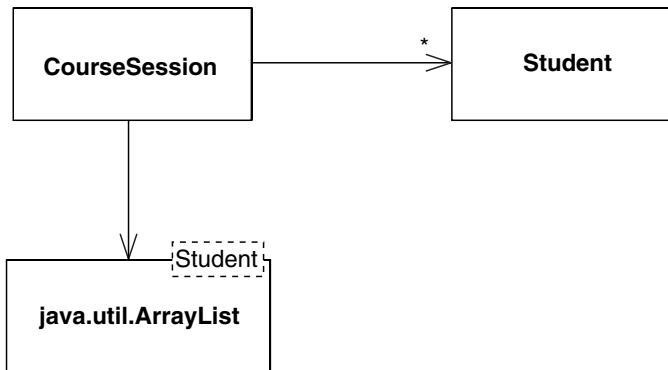


Figure 2.3 Class Diagram with Parameterized Type

objects.⁴ The `enroll` method adds the student to this list, and the `getAllStudents` method simply returns the list.

Figure 2.3 shows how `CourseSession` is now dependent upon the parameterized type `java.util.ArrayList<Student>`. Also, the fact that `CourseSession` is dependent upon zero to many students is shown by the `*` at the end of the association between `CourseSession` and `Student`.

The class diagram in Figure 2.3 is not normal—it really shows the same information twice, in a different fashion. The parameterized type declaration suggests that a single `CourseSession` has a relationship to a number of `Student` objects stored in an `ArrayList`. The association from `CourseSession` to `Student` similarly shows a one-to-many relationship.

Incremental Refactoring

Incremental Refactoring

Since `java.util.ArrayList` provides a `size` method, you can just ask the students `ArrayList` object for its size instead of tracking `numberOfStudents` separately.

```

int getNumberOfStudents() {
    return students.size();
}
  
```

Make this small refactoring, then recompile and rerun the test. Having the test gives you the confidence to change the code with impunity—if the change doesn’t work, you simply undo it and try something different.

⁴Due to width restrictions, I’ve broken the `students` field declaration into two lines. You can type this as a single line if you choose.

Since you no longer return `numberOfStudents` from `getNumberOfStudents`, you can stop incrementing it in the `enroll` method. This also means you can delete the `numberOfStudents` field entirely.

While you could scan the `CourseSession` class for uses of `numberOfStudents`, removing each individually, you can also use the compiler as a tool. Delete the field declaration, then recompile. The compiler will generate errors on all the locations where the field is still referenced. You can use this information to navigate directly to the code you need to delete.

The final version of the class should look like the following code:

```
class CourseSession {
    private String department;
    private String number;
    private java.util.ArrayList<Student> students =
        new java.util.ArrayList<Student>();

    CourseSession(String department, String number) {
        this.department = department;
        this.number = number;
    }

    String getDepartment() {
        return department;
    }

    String getNumber() {
        return number;
    }

    int getNumberOfStudents() {
        return students.size();
    }

    void enroll(Student student) {
        students.add(student);
    }

    java.util.ArrayList<Student> getAllStudents() {
        return students;
    }
}
```

Objects in Memory

Objects in Memory

In `testEnrollStudents`, you send the `getAllStudents` message to `session` and store the result in a `java.util.ArrayList` reference that is bound to the `Student` class—it can only contain `Student` objects. Later in the test, after enrolling the second

student, `allStudents` now contains both students—you don't need to ask the `session` object for it again.

```
public void testEnrollStudents() {
    CourseSession session = new CourseSession("ENGL", "101");

    Student student1 = new Student("Cain DiVoe");
    session.enroll(student1);
    assertEquals(1, session.getNumberOfStudents());
    java.util.ArrayList<Student> allStudents = session.getAllStudents();
    assertEquals(1, allStudents.size());
    assertEquals(student1, allStudents.get(0));

    Student student2 = new Student("Coralee DeVaughn");
    session.enroll(student2);
    assertEquals(2, session.getNumberOfStudents());
    assertEquals(2, allStudents.size());
    assertEquals(student1, allStudents.get(0));
    assertEquals(student2, allStudents.get(1));
}
```

Objects in
Memory

The reason is illustrated in Figure 2.4. The `CourseSession` object holds on to the `students` field as an attribute. This means that the `students` field is available throughout the lifetime of the `CourseSession` object. Each time the `getNumberOfStudents` message is sent to the `session`, a reference to the very same `students` field is returned. This reference is a memory address, meaning that

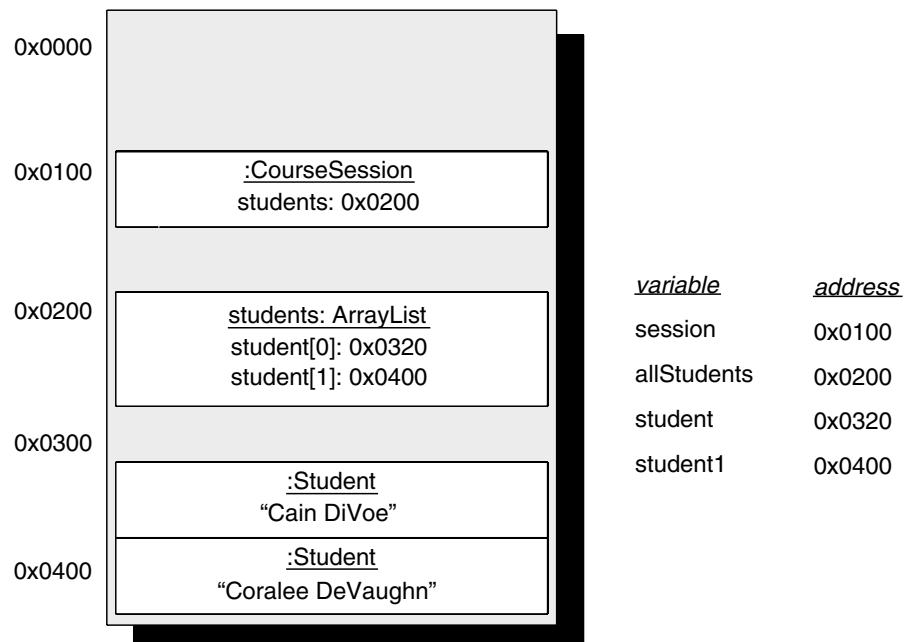


Figure 2.4 Memory Diagram

any code using the reference ends up at the same memory location at which the students field is stored.

Packages and the `import` Statement

Up until now, you have been using the *fully qualified* name of the class `java.util.ArrayList`. The fully qualified name of a class includes its package name (`java.util` in this example) followed by the class name (`ArrayList`).

Packages provide a way for developers to group related classes. They can serve several needs: First, grouping classes into packages can make it considerably easier on developers, saving them from having to navigate dozens, hundreds, or even thousands of classes at a time. Second, classes can be grouped into packages for distribution purposes, perhaps to allow easier reuse of modules or subsystems of code.

Third, packages provide *namespaces* in Java. Suppose you have built a class called `Student` and you purchase a third-party API to handle billing students for tuition. If the third-party software contains a class also named `Student`, any references to `Student` will be ambiguous. Packages provide a way to give a class a more unique name, minimizing the potential for class name conflicts. Your class might have a fully qualified name of `com.mycompany.studentinfosystem.Student`, and the third-party API might use the name `com.thirdpartyco.expensivepackage.Student`.

I will usually refer to the Java system classes without the package name unless the package is not clear from the class name. For example, I will use `ArrayList` in place of `java.util.ArrayList`, and `Object` instead of `java.lang.Object`.

Typing `java.util.ArrayList` throughout code can begin to get tedious, and it clutters the code as well. Java provides a keyword—`import`—that allows identification of fully qualified class names and/or packages at the source file level. Use of `import` statements allows you to specify simple class names throughout the remainder of the source file.

Update `CourseSessionTest` to include `import` statements as the very first lines in the source file. You can now shorten the phrase `extends junit.framework.TestCase` to `extends TestCase`. You can change the reference defined as `java.util.ArrayList<Student>` to `ArrayList<Student>`.

```
import junit.framework.TestCase;  
import java.util.ArrayList;  
  
public class CourseSessionTest extends TestCase {  
    ...
```

```

public void testEnrollStudents() {
    CourseSession session = new CourseSession("ENGL", "101");

    Student student1 = new Student("Cain DiVoe");
    session.enroll(student1);
    assertEquals(1, session.getNumberOfStudents());
    ArrayList<Student> allStudents = session.getAllStudents();
    ...
}

```

(Make sure your tests still run! I'll keep reminding you for a while.)

Update AllTests, StudentTest, and CourseSession to use `import` statements. Your code will look so much cleaner!

The `java.lang` Package

The Default Package and the package Statement

The `String` class is also part of the system class library. It belongs to the package named `java.lang`. So why haven't you had to either use its fully qualified class name (`java.lang.String`) or provide an `import` statement?

The Java library contains classes so fundamental to Java programming that they are needed in many, if not all, classes. The classes `String` and `Object` are two such classes. The ubiquitous nature of these classes is such that the designers of Java wanted to save you from the nuisance of having to specify an `import` statement for them everywhere.

If you refer to the `String` class, then the statement:

```
import java.lang.String;
```

is implicit in each and every Java source file.

When you learn about inheritance (Lesson 6), you will find that every class implicitly extends from the class `java.lang.Object`. In other words, if a class declaration does not contain an `extends` keyword, it is as if you coded:

```
class ClassName extends java.lang.Object
```

This is another shortcut provided by the Java compiler.

The Default Package and the package Statement

None of the classes you have built—`AllTests`, `StudentTest`, `Student`, `CourseSessionTest`, and `CourseSession`—specify a package. This means that they go into something called the *default package*. The default package is fine for

sample or academic programs. But for any real software development you do, all of your classes should belong to some package other than the default package. In fact, if you place classes in the default package, you will not be able to reference these classes from other packages.

If you are working in a modern IDE, moving a class into a package can be as easy as dragging the class name onto a package name. If you are not working in an IDE, setting up packages and understanding related classpath issues can be somewhat complex. Even if you are working in an IDE, it is important to understand the relationship between packages and the underlying file system directory structure.

Let's try moving your classes into a package named `studentinfo`. Note that package names are by convention comprised of lowercase letters. While you should still avoid abbreviations, package names can get unwieldy fairly quickly. The judicious use of abbreviations might help keep the package name from being unmanageable.

You should also not start your package names with `java` or `javax`, both of which should be used exclusively by Sun.

If your IDE supports moving classes into packages as a simple operation, go ahead and take advantage of it. However, make sure you understand how the classes relate to a package structure, as this will be important when you construct deployable libraries. Consider copying the source files into a different directory structure and following the exercise anyway.

From the directory in which your class files are located, create a new subdirectory called `studentinfo`. Case is important—ensure that you name this subdirectory using all lowercase letters. If your source files are currently located in `c:\source`, then you should now have a directory named `c:\source\studentinfo`. Starting with a Unix directory named `/usr/src`, you should have a directory named `/usr/src/studentinfo`.

First carefully delete all of the class files that have been generated. Class files, remember, end with a `.class` extension. Don't lose all of your hard work—back up your directory if you are unsure about this step. Under Windows, the command:

```
del *.class
```

will suffice. Under Unix, the equivalent command is:

```
rm *.class
```

After deleting the class files, move the five source files (`AllTests.java`, `StudentTest.java`, `Student.java`, `CourseSessionTest.java`, and `CourseSession.java`) into the `studentinfo` directory. You need to store classes in a directory structure that corresponds to the package name.

The Default Package and the package Statement

Edit each of the five Java source files. Add a package statement to each that identifies the class as belonging to the `studentinfo` package. The package statement must be the very first statement within a Java class file.

```
package studentinfo;

class Student {
...
```

You should be able to compile the classes from within the `studentinfo` subdirectory.

However, two things will have to change when you run JUnit against `AllTests`. First, the full name of the class will have to be passed into the `TestRunner`, otherwise JUnit will not find it. Second, if you are in the `studentinfo` directory, the `TestRunner` will not be able to find `studentinfo.AllTests`. You must either go back to the parent directory or, better yet, alter the classpath to explicitly point to the parent directory. The following command shows the altered classpath and the fully qualified test class name.

```
java -cp c:\source;c:\junit3.8.1\junit.jar junit.awtui.TestRunner studentinfo.AllTests
```

The `setUp`
Method

In fact, if you explicitly point to `c:\source` in the classpath, you can be in any directory when you execute the `java` command. You will also want to use a similar classpath in `javac` compilations:

```
javac -classpath c:\source;c:\junit3.8.1\junit.jar studentinfo\*.java
```

Another way to look at this: The classpath specifies the starting, or “root,” directory in which to find any classes that you will use. The class files must be located in the subdirectory of one of the roots that corresponds to the package name. For example, if the classpath is `c:\source` and you want to include the class `com.mycompany.Bogus`, then the file `Bogus.class` must appear in `c:\source\com\mycompany`.

A package-naming standard is something that your development team will want to agree upon. Most companies name their packages starting with the reverse of their web domain name. For example, software produced by a company named Minderbinder Enterprises might use package names starting with `com.minderbinder`.

The `setUp` Method

The test code in `CourseSessionTest` needs a bit of cleanup work. Note that both tests, `testCreate` and `testEnrollStudents`, instantiate a new `CourseSession` object and store a reference to it in a local variable named `session`.

JUnit provides you with a means to eliminate this duplication—a `setUp` method. If you provide code in this `setUp` method, JUnit will execute that code prior to executing each and every test method. You should put common test initialization code in this method.

```
public class CourseSessionTest extends TestCase {
    private CourseSession session;

    public void setUp() {
        session = new CourseSession("ENGL", "101");
    }

    public void testCreate() {
        assertEquals("ENGL", session.getDepartment());
        assertEquals("101", session.getNumber());
        assertEquals(0, session.getNumberOfStudents());
    }

    public void testEnrollStudents() {
        Student student1 = new Student("Cain DiVoe");
        session.enroll(student1);
        ...
    }
}
```

The `setUp` Method

Caution!

It's easy to make the mistake of coding `setUp` so that it declares `session` as a local variable:

```
public void setUp() {
    CourseSession session =
        new CourseSession("ENGL", "101");
}
```

It is legal to define a local variable with the same name as an instance variable, but it means that the *instance* variable `session` will not get properly initialized. The result is an error known as a `NullPointerException`, which you'll learn about in Lesson 4.

In `CourseSessionTest`, you add the instance variable `session` and assign to it a new `CourseSession` instance created in the `setUp` method. The test methods `testCreate` and `testEnrollStudents` no longer need this initialization line. Both test methods get their own separate `CourseSession` instance.

Even though you could create a constructor and supply common initialization code in it, doing so is considered bad practice. The preferred idiom for test initialization in JUnit is to use the `setUp` method.

More Refactoring

The method `testEnrollStudents` is a bit longer than necessary. It contains a few too many assertions that track the number of students. Overall, the method is somewhat difficult to follow.

Instead of exposing the entire list of students to *client* code (i.e., other code that works with `CourseSession` objects), you can instead ask the `CourseSession` to return the `Student` object at a specific index. You currently have no need for the entire list of students as a whole, so you can eliminate the `getAllStudents` method. This also means that you no longer have to test the size of the `ArrayList` returned by `getAllStudents`. The test method can be simplified to this:

```
public void testEnrollStudents() {
    Student student1 = new Student("Cain DiVoe");
    session.enroll(student1);
    assertEquals(1, session.getNumberOfStudents());
    assertEquals(student1, session.get(0));

    Student student2 = new Student("Coralee DeVaughn");
    session.enroll(student2);
    assertEquals(2, session.getNumberOfStudents());
    assertEquals(student1, session.get(0));
    assertEquals(student2, session.get(1));
}
```

More
Refactoring

Add the `get` method to `CourseSession` and remove the method `getAllStudents`. This refactoring shows how you can move common code from the test class directly into the production class in order to remove duplication.

```
class CourseSession {
    ...
    ArrayList<Student> getAllStudents() {
        return students;
    }

    Student get(int index) {
        return students.get(index);
    }
}
```

Another significant benefit of this refactoring is that you have hidden a detail of `CourseSession` that was unnecessarily exposed. You have *encapsulated* the students collection, only selectively allowing the `get`, `add`, and `size` operations to be performed on it.

This encapsulation provides two significant benefits: First, you currently store the list of in an `ArrayList`. An `ArrayList` is one specific kind of data

structure with certain use and performance characteristics. If you expose this list directly to client code, their code now depends upon the fact that the students are available as an `ArrayList`. This dependency means that later you cannot easily change the representation of how students are stored. Second, exposing the entire collection means that other classes can manipulate that collection—add new `Student` objects, remove `Student` objects, and so on—without your `CourseSession` class being aware of these changes. The integrity of your `CourseSession` object could be violated!

Class Constants

It is never a good idea to embed literals directly in code, as mentioned earlier. Declaring a local variable as `final`, which prevents other code from reassigning its value, is a good idea. The `final` keyword tells other developers that “I don’t intend for you to be able to change the value of this variable.”

Idiosyncracies: The Final Keyword

You can mark local object and primitive variables as `final`:

```
public void testCreate() {  
    final String firstStudentName =  
        "Jane Doe";  
    final Student firstStudent =  
        new Student(firstStudentName);
```

You can also mark arguments as `final`:

```
Student(final String name) {  
    this.name = name;  
}
```

Doing so can provide you with an additional level of protection from other code in the method that assigns a different value to the local variable or argument. Many developers insist on doing so, and it’s not a bad practice.

I choose not to, but I recommend that you try it and see how it helps you. My general reason to mark fields as `final` is for readability, not protection. As a rule, you should never assign values to arguments. And you should rarely reassign values to local object reference variables. I follow both of these rules and don’t feel the need to demonstrate that fact by marking the field as `final`. Instead, I use `final` to emphasize the fact that a local declaration is a literal constant and not just an initialized variable. Your mileage may vary.

Frequently you will need more than one class to use the same literal. In fact, if you are doing test-driven development properly, any time you create a literal in code, it is there by virtue of some assertion in a test method against that same literal. A test might say, “assert that these conditions produce an error and that the error message is such-and-such a message.” Once this test is written, you then write the production code that produces such-and-such an error message under the right conditions. Now you have code duplication—“such-and-such an error message” appears in both the test and the production code.

While a few duplicated strings here and there may seem innocuous, overlooking the need to refactor out this duplication may prove costly in the future. One possibility is related to the growth of your software. Initially you may only deploy your system to local customers. Later, as your software achieves more success, the business may decide that it needs to deploy the software in another country. You will then need to *internationalize* your software, by deploying it with support for other languages and with consideration for other cultures.

Many software developers have encountered this problem. The software they worked on for months or years has hundreds or thousands of literal Strings strewn throughout the code. It is a major effort to retrofit internationalization support into this software.

As a craftsman of Java code, it is your job to stay vigilant and ensure that duplication is eliminated as soon as you recognize it.⁵

Class
Constants



Replace common literals with class constants.

A class constant is a field that is declared with the static and final keywords. As a reminder, the final keyword indicates that the field reference cannot be changed to point to a different value. The static keyword means that the field is available for use without having to first construct an instance of the class in which it is declared. It also means that there is one and only one field in memory instead of one field for each object created. The following example declares a class constant:

```
class ChessBoard {  
    static final int SQUARES_PER_SIDE = 8;  
}
```

⁵For the problem of duplicated strings, a more robust solution involving *resource bundles* is more appropriate. See Additional Lesson III for a brief discussion of resource bundles.

By convention, class constants appear in uppercase letters. When all letters are uppercase, camel notation is not possible, so the standard is to use underscores to separate words.

You can refer to the class constant by first specifying the class name, followed by the dot (.)operator, followed by the name of the constant.

```
int numberOfSquares =  
    ChessBoard.SQUARES_PER_SIDE * ChessBoard.SQUARES_PER_SIDE;
```

You will use class constants already defined in the Java class library in the next section on Dates. Shortly thereafter you will define your own class constant in the CourseSession code.

Overloaded Constructors

Dates

If you browse through the J2SE API documentation for the package `java.util`, you will see several classes that are related to times and dates, including `Calendar`, `GregorianCalendar`, `Date`, `TimeZone`, and `SimpleTimeZone`. The `Date` class provides a simple timestamp mechanism. The other, related classes work together with `Date` to provide exhaustive support for internationalized dates and for working with components of a timestamp.

Initial versions of Java shipped with the `Date` class as the sole provider of support for dates and times. The `Date` class was designed to provide most of the functionality needed. It is a simple implementation: Internally, a date is represented by the number of milliseconds (thousandths of a second) since January 1, 1970 GMT at 00:00:00 (known as the “epoch”).

Overloaded Constructors

The `Date` class provides a handful of constructors. It is possible and often desirable to provide a developer with more than one way to construct new objects of a class type. You will learn how to create multiple constructors for your class in this lesson.

In the `Date` class, three of the constructors allow you to specify time parts (year, month, day, hour, minute, or second) in order to build a `Date` object for a specific date and time. A fourth constructor allows you to construct a date from an input `String`. A fifth constructor allows you to construct a date using the number milliseconds since the epoch. A final constructor, one that

takes no parameters, constructs a timestamp that represents “now,” the time at which the Date object was created.

There are also many methods that allow getting and setting of fields on the Date, such as `setHour`, `setMinutes`, `getDate`, and `getSeconds`.

The Date class was not designed to provide support for internationalized dates, however, so the designers of Java introduced the Calendar classes in J2SE 1.1. The intent was for the Calendar classes to supplement the Date class. The Calendar class provides the ability to work with dates by their constituent parts. This meant that the constructors and getters/setters in the Date class were no longer needed as of J2SE 1.1. The cleanest approach would have been for Sun to simply remove the offending constructors and methods.

However, if Sun were to have changed the Date class, large numbers of existing applications would have had to have been recoded, recompiled, retested, and redeployed. Sun chose to avoid this unhappy circumstance by instead *deprecating* the constructors and methods in Date. This means that the methods and constructors are still available for use, but the API developers are warning you that they will remove the deprecated code from the next major release of Java. If you browse the Date class in the API documentation, you will see that Sun has clearly marked the deprecated methods and constructors.

In this exercise, you will actually use the deprecated methods. You’ll see that their use generates warnings from the compiler. Warnings are for the most part bad—they indicate that you’re doing something in code that you probably should not be. There’s always a better way that does not generate a warning. As the exercise progresses, you will eliminate the warnings using an improved solution.



In the student information system, course sessions need to have a start and end date, marking the first and last days of class. You could provide both start and end dates to a CourseSession constructor, but

you have been told that sessions are always 16 weeks (15 weeks of class, with a one-week break after week 7). With this information, you decide to design the CourseSession class so that users of the class need only supply the start date—your class will calculate the end date.

Here’s the test to be added to CourseSessionTest.

```
public void testCourseDates() {
    int year = 103;
    int month = 0;
    int date = 6;
    Date startDate = new Date(year, month, date);

    CourseSession session =
        new CourseSession("ABCD", "200", startDate);
```

Overloaded Constructors

```

year = 103;
month = 3;
date = 25;
Date sixteenWeeksOut = new Date(year, month, date);
assertEquals(sixteenWeeksOut, session.getEndDate());
}

```

You will need an `import` statement at the top of `CourseSessionTest`:

```
import java.util.Date;
```

This code uses one of the deprecated constructors for `Date`. Also of note are the odd-looking parameter values being passed to the `Date` constructor. Year 103? Month 0?

The API documentation, which should be your first reference for understanding a system library class, explains what to pass in for the parameters. Specifically, documentation for the `Date` constructor says that the first parameter represents “the year minus 1900,” the second parameter represents “the month between 0 and 11,” and the third parameter represents “the day of the month between 1-31.” So `new Date(103, 0, 6)` would create a `Date` representation for January 6, 2003. Lovely.

Since the start date is so critical to the definition of a course session, you want the constructor to require the start date. The test method constructs a new `CourseSession` object, passing in a newly constructed `Date` object in addition to the department and course number. You will want to change the instantiation of `CourseSession` in the `setUp` method to use this modified constructor. But as an interim, incremental approach, you can instead supply an additional, *overloaded* constructor.

The test finally asserts that the session end date, returned by `getEndDate`, is April 25, 2003.

You will need to make the following changes to `CourseSession` in order to get the test to pass:

- Add `import` statements for `java.util.Date`, `java.util.Calendar`, and `java.util.GregorianCalendar`;
- add a `getEndDate` method that calculates and returns the appropriate session end date; and
- add a new constructor that takes a starting date as a parameter.

The corresponding production code:

```

package studentinfo;

import java.util.ArrayList;
import java.util.Date;

```

Overloaded Constructors

```

import java.util.Calendar;
import java.util.GregorianCalendar;

class CourseSession {
    private String department;
    private String number;
    private ArrayList<Student> students = new ArrayList<Student>();
    private Date startDate;

    CourseSession(String department, String number) {
        this.department = department;
        this.number = number;
    }

    CourseSession(String department, String number, Date startDate) {
        this.department = department;
        this.number = number;
        this.startDate = startDate;
    }

    ...
    Date getEndDate() {
        GregorianCalendar calendar = new GregorianCalendar();
        calendar.setTime(startDate);
        int numberOfDays = 16 * 7 - 3;
        calendar.add(Calendar.DAY_OF_YEAR, numberOfDays);
        Date endDate = calendar.getTime();
        return endDate;
    }
}

```

Overloaded Constructors

As I describe what `getEndDate` does, try to follow along in the J2SE API documentation for the classes `GregorianCalendar` and `Calendar`.

In `getEndDate`, you first construct a `GregorianCalendar` object. You then use the `setTime`⁶ method to store the object representing the session start date in the calendar. Next, you create the local variable `numberOfDays` to represent the number of days to be added to the start date in order to come up with the end date. The appropriate number is calculated by multiplying 16 weeks by 7 days per week, then subtracting 3 days (since the last day of the session is on the Friday of the 16th week).

The next line:

```
calendar.add(Calendar.DAY_OF_YEAR, numberOfDays);
```

sends the `add` message to the calendar object. The `add` method in `GregorianCalendar` takes a field and an amount. You will have to look at the J2SE API documentation for `Calendar`—in addition to the documentation for `GregorianCalendar`—to fully understand how to use the `add` method. `Gregorian-`

⁶Don't use this as a good example of method naming!

Calendar is a *subclass* of Calendar, which means that the way it works is tied tightly to Calendar. The field that represents the first parameter tells the calendar object what you are adding *to*. In this case, you want to add a number to the day of year. The Calendar class defines DAY_OF_YEAR as well as several other class constants that represent date parts, such as YEAR.

The calendar now contains a date that represents the end date of the course session. You extract this date from the calendar using the `getTime` method and finally return the end date of the course session as the result of the method.

You might wonder if the `getEndDate` method will work, then, when the start date is close to the end of the year. If that can possibly happen, it's time to write a test for it. However, the student information system you are writing is for a university that has been around for 200 years. No semester has ever begun in one year and ended in the next, and it will never happen. In short, you're not going to need to worry about it . . . yet.

Overloaded Constructors

The second constructor will be short-lived, but it served the purpose of allowing you to quickly get your new test to pass. You now want to remove the older constructor, since it doesn't initialize the session start date. To do so, you will need to modify the `setUp` method and `testCourseDates`. You should also amend the creation test to verify that the start date is getting stored properly.

```
package studentinfo;

import junit.framework.TestCase;
import java.util.ArrayList;
import java.util.Date;

public class CourseSessionTest extends TestCase {
    private CourseSession session;
    private Date startDate;

    public void setUp() {
        int year = 103;
        int month = 0;
        int date = 6;
        startDate = new Date(year, month, date);
        session = new CourseSession("ENGL", "101", startDate);
    }

    public void testCreate() {
        assertEquals("ENGL", session.getDepartment());
        assertEquals("101", session.getNumber());
        assertEquals(0, session.getNumberofStudents());
        assertEquals(startDate, session.getStartDate());
    }

    ...
}
```

```

public void testCourseDates() {
    int year = 103;
    int month = 3;
    int date = 25;
    Date sixteenWeeksOut = new Date(year, month, date);
    assertEquals(sixteenWeeksOut, session.getEndDate());
}
}

```

You can now remove the older constructor from CourseSession. You'll also need to add the getStartDate method to CourseSession:

```

class CourseSession {
    ...
    Date getStartDate() {
        return startDate;
    }
}

```

**Deprecation
Warnings**

Deprecation Warnings

The above code will compile and pass the tests, but you will see a warning or note in the compilation output. If you are using an IDE, you might not see the warnings. Find out how to turn on the warnings—you don't want to hide them.



Eliminate all warnings.

Ignoring a compiler warning is like ignoring a sore tooth—you'll pay for it sooner or later, and it may end up costing you dearly to ignore it for too long.⁷

Note: CourseSessionTest.java uses or overrides a deprecated API.

Note: Recompile with -Xlint:deprecation for details.

If you are compiling from the command line, you should do just what the message says: Enter the compilation command again, but modify it to include the compilation switch `-Xlint:deprecation` option.

```
javac -classpath c:\junit3.8.1\junit.jar -Xlint:deprecation *.java
```

The new compile output should look something like this:

```
CourseSessionTest.java:15: warning: Date(int,int,int) in java.util.Date has been deprecated
    startDate = new Date(year, month, date);
                           ^

```

⁷He writes, as he sits here with a corrupt molar . . .

```
CourseSessionTest.java:43: warning: Date(int,int,int) in java.util.Date has been deprecated  
    Date sixteenWeeksOut = new Date(year, month, date);  
                           ^  
2 warnings
```

If you are running in an IDE, it will have a setting somewhere to allow you to turn warnings on or off. You might already have seen similar deprecation messages, since warnings are usually “on” by default. In any case, the warnings should be disturbing to your craftsperson sensibilities. May they nag at you incessantly. You will soon learn a different way to construct dates that does not result in the warnings.

The test should run just fine. But you have lots of little ugliness in the code that can be cleaned up.

Refactoring

Refactoring

A quick improvement is to remove the unnecessary local variable, `endDate`, that appears at the end of the method `getEndDate` in `CourseSession`. Declaring the temporary variable aided in your understanding of the `Calendar` class:

```
Date endDate = calendar.getTime();  
return endDate;
```

A more concise form is to simply return the `Date` object returned by the call to `getTime`.

```
return calendar.getTime();
```

Import Refactorings

The `CourseSession` class now must import four different classes from the package `java.util`:

```
import java.util.ArrayList;  
import java.util.Date;  
import java.util.GregorianCalendar;  
import java.util.Calendar;
```

This is still reasonable, but you can see that the list could quickly get out of hand as more system library classes are used. It is also a form of duplication—the package name is duplicated in each `import` statement. Remember that your primary refactoring job is to eliminate as much duplication as possible.

Java provides a shortcut version of the `import` statement to declare that you are importing all classes from a specified package:

```
import java.util.*;
```

This form of import is known as a *package import*. The asterisk (*) acts as a wildcard character.

After making this change, you can use any additional classes from the `java.util` package without having to modify or add to the `import` statements. Note that there is no runtime penalty for using either this form or the singular class form of the `import` statement. An `import` statement merely declares that classes *might* be used in the class file. An `import` statement does not guarantee that any classes from a package are used in the class file.

There is no consensus about which form is more proper. Most shops use the * form of the `import` statement either all the time or when the number of `import` statements begins to get unwieldy. Some shops insist that all classes must be explicitly named in the `import` statements, which makes it easier to determine the package a class comes from. Modern Java IDEs can enforce the `import` convention your shop decides on and even switch back and forth between the various forms. An IDE makes the choice of which form to use less significant.

It is possible to import a class or package but not use the class or any classes from the package within your source code. The Java compiler will not warn you about these unnecessary `import` statements. Most modern IDEs have optimizing facilities that will help you remove them.

Refactoring

Improving Understanding with a Factory Method

The `getEndDate` method in `CourseSession` is the most complex method you've written yet. It's about as long as you want your methods to get. Most of the methods you write should be between one and half a dozen lines. Some methods might be between half a dozen and a dozen or so lines. If your methods are regularly this length or even longer, you should work on refactoring them. The primary goal is to ensure that methods can be rapidly understood and maintained.

If your methods are short enough, it will be easy to provide a meaningful, concise name for them. If you are finding it difficult to name a method, consider breaking it up into smaller methods, each of which does only one succinctly nameable thing.

Another bit of duplication and lack of clarity that you should be unhappy with lies in the test method. For the time being, you are using the deprecated Date constructors, a technique that is considerably simpler than using the

Calendar class. However, the code is somewhat confusing because the year has to be specified as relative to 1900, and the months have been numbered from 0 through 11 instead of 1 through 12.

In CourseSessionTest, code a new method called `createDate` that takes more sensible inputs.

```
Date createDate(int year, int month, int date) {  
    return new Date(year - 1900, month - 1, date);  
}
```

This will allow you to create dates using a 4-digit year and a number from 1 through 12 for the month.

You can now refactor `setUp` and `testCourseDates` to use this utility method. With the introduction of the utility method, defining the local variables `year`, `month`, and `date` adds little to the understanding of the code, since the *factory method*⁸ `createDate` encapsulates some of the confusion. You can eliminate the local variables and embed them directly as parameters in the message send to `createDate`:

```
public void setUp() {  
    startDate = createDate(2003, 1, 6);  
    session = new CourseSession("ENGL", "101", startDate);  
}  
...  
public void testCourseDates() {  
    Date sixteenWeeksOut = createDate(2003, 4, 25);  
    assertEquals(sixteenWeeksOut, session.getEndDate());  
}
```

Some of you may be shaking your head at this point. You have done a lot of work, such as introducing local variables, and then you have undone much of that same work shortly thereafter.

Part of crafting software is understanding that code is a very malleable form. The best thing you can do is get into the mindset that you are a sculptor of code, shaping and molding it into a better form at all times. Once in a while, you'll add a flourish to make something in the code stand out. Later, you may find that the modification is really sticking out like a sore thumb, asking for someone to soothe it with a better solution. You will learn to recognize these trouble spots in your code ("code smells," as Martin Fowler calls them.)⁹

⁸A method responsible for creating and returning objects. Another, perhaps more concise term, is "creation method."

⁹[Wiki2004].

You will also learn that it is cheaper to fix code problems now rather than wait until the code is too intertwined with the rest of the system. It doesn't take long!



Keep your code clean at all times!

Creating Dates with Calendar

You still receive deprecation warnings each time you compile. Bad, bad. You should get rid of the warnings before someone complains. The benefit of having moved the date creation into a separate method, `createDate`, is that now you will only have to make a change in one place in your code, instead of two, to eliminate the warnings.

Instead of creating a `Date` object using the deprecated constructor, you will use the `GregorianCalendar` class. You can build a date or timestamp from its constituent parts by using the `set` method defined in `Calendar`. The API documentation for `Calendar` lists the various date parts that you can set. The `createDate` method builds a date by supplying a year, a month, and a day of the month.

**Creating Dates
with Calendar**

```
Date createDate(int year, int month, int date) {
    GregorianCalendar calendar = new GregorianCalendar();
    calendar.clear();
    calendar.set(Calendar.YEAR, year);
    calendar.set(Calendar.MONTH, month - 1);
    calendar.set(Calendar.DAY_OF_MONTH, date);
    return calendar.getTime();
}
```

The `GregorianCalendar` is a bit more sensible than `Date` in that a year is a year. If the real year is 2005, you can pass 2005 to the `calendar` object instead of 105.

You will need to modify the `import` statements in `CourseSessionTest` in order to compile and test these changes. The simplest way is to import everything from the `java.util` package:

```
import java.util.*;
```

Compile and test. Congratulations—no more embarrassing deprecation warnings!

Comments

One part of the `getEndDate` method that could use clarification is the calculation for the number of days to add to the session start date.

```
int numberOfDay = 16 * 7 - 3;
```

To another developer that has to maintain this method, the mathematical expression is not immediately obvious. While the maintainer can probably figure it out in a few minutes, it's far less expensive for you as the original developer to explain what you were thinking.

Java allows you to add free-form explanatory text within the source file in the form of *comments*. The compiler ignores and discards comments encountered when it reads the source file. It is up to you to decide where and when comments are appropriate.

Comments

You can add a *single-line comment* to the `numberOfDays` calculation. A single-line comment begins with two forward slashes (`//`) and continues to the end of the current source line. Anything from the first slash to the end of the line is ignored by the compiler.

```
int numberOfDay = 16 * 7 - 3; // weeks * days per week - 3 days
```

You can place single-line comments on a separate line:

```
// weeks * days per week - 3 days  
int numberOfDay = 16 * 7 - 3;
```

However, comments are notorious for being incorrect or misleading. The above comment is a perfect example of an unnecessary comment. A better solution is to figure out a clearer way to express the code.



Replace comments with more expressive code.

One possible solution:

```
final int sessionLength = 16;  
final int daysInWeek = 7;  
final int daysFromFridayToMonday = 3;  
int numberOfDay =  
    sessionLength * daysInWeek - daysFromFridayToMonday;
```

Hmmm. Well, it's more expressive, but I'm not sure that `daysFromFridayToMonday` expresses exactly what's going on. This should demonstrate that there's not always a perfect solution. Refactoring is not an exact science. That shouldn't stop you from trying, however. Most changes *do* improve the code, and

someone (maybe you) can always come along after you and figure out an even better way. For now, it's your call.

Java supplies another form of comment known as a *multiline comment*. A multiline comment starts with the two characters /* and ends with the two characters */. Everything from the beginning slash through to the ending slash is ignored by the compiler.

Note that while multiline comments can nest single line comments, multiline comments cannot nest other multiline comments.

As an example, the Java compiler allows the following:

```
int a = 1;
/* int b = 2;
// int c = 3;
*/
```

while the following code will not compile:

```
int a = 1;
/* int b = 2;
/* int c = 3; */
*/
```

Javadoc
Comments

You should prefer single-line comments for the few places that you need to annotate the code. The multiline comment form can then be used for rapidly “commenting out” (turning off the code so that it is not read by the compiler) large blocks of code.

Javadoc Comments

Another use for multiline comments is to provide formatted code documentation that can later be used for automated generation of nicely formatted API documentation. These comments are known as javadoc comments, since there is a javadoc tool that will read your source files, look for javadoc comments, and extract them along with other necessary information in order to build documentation web pages. Sun’s documentation for the Java APIs themselves was produced using javadoc.

A javadoc comment is a multiline comment. The distinction is that a javadoc comment starts with /** instead of /*. To the javac compiler, there is no difference between the two, since both start with /* and end with */. The javadoc tool understands the difference, however.

Javadoc comments appear directly before the Java element they are documenting. Javadoc comments can appear before fields, but most typically they are used to document classes and methods. There are rules for how a javadoc comment must be formatted in order to be parsed correctly by the javadoc compiler.

The primary reason for producing javadoc web pages is to document your code for external consumption by other project teams or for public distribution. While you can code javadoc comments for every Java element (fields, methods, classes, etc.), you should only expend the time to produce javadoc comments for things you want to expose. Javadoc comments are intended for telling a client developer how to work with a class.

In a team doing test-driven development, Javadoc comments are of lesser value. If done properly, the tests you produce using test-driven development provide far better documentation on the capabilities of a class. Practices such as pair programming and *collective code ownership*, where developers work on all parts of the system, also minimize the need for producing javadoc comments.

If you code concise, well-named methods, with well-named parameters, the amount of additional text that you should have to write in a Javadoc comment should be minimal. In the absence of text, the Javadoc compiler does a fine job of extracting and presenting the names you chose.

As a brief exercise, provide a Javadoc comment for the CourseSession class—the single-argument constructor—and for one of the methods within the class.

Here are the Javadoc comments I came up with:

```
package studentinfo;

import java.util.*;

/**
 * Provides a representation of a single-semester
 * session of a specific university course.
 * @author Administrator
 */
class CourseSession {
    private ArrayList<Student> students = new ArrayList<Student>();
    private Date startDate;

    CourseSession() {
    }

    /**
     * Constructs a CourseSession starting on a specific date
     *
     * @param startDate the date on which the CourseSession begins
    }
```

```

*/
CourseSession(Date startDate) {
    this.startDate = startDate;
}

/**
 * @return Date the last date of the course session
 */
Date getEndDate() {
...
}

```

Take particular note of the @ keywords in the javadoc comments. When you look at the web pages produced by the Javadoc command, it will be apparent what the Javadoc compiler does with the tagged comments. The Javadoc keywords that you will want to use most frequently are @param, to describe a parameter, and @return, to describe what a method returns.

There are lots of rules and many additional @ keywords in Javadoc. Refer to the Javadoc documentation for further information. The Javadoc documentation can be found in the API documentation for the Java SDK (either online or downloaded), under the tool docs for your platform.

Once you've added these or similar comments to your code, go to the command line. (If you are using an IDE, you may be able to generate the documentation from within the IDE.) You will want to create a new empty directory in which to store the generated docs, so that you can easily delete it when you regenerate the Javadoc. Navigate to this empty directory¹⁰ and enter the following command:

```
javadoc -package -classpath c:\source;c:\junit3.8.1\junit.jar studentinfo
```

The javadoc program will produce a number of .html files and a stylesheet (.css) file. Open the file index.html in a web browser and take a few minutes to see what this simple command has produced (Figure 2.5). Fairly impressive, no?

Well, yes and no. I'm embarrassed by the comments I had you put in for the methods. They really added nothing that the code didn't already state. The @param keyword simply restates information that can be gleaned from the parameter type and name. The @return keyword restates information that can be derived from the method name and return type. If you find a need for @return and @param keywords, try to rename the parameters and the method to eliminate this need.

javadoc
Comments

¹⁰ Instead of navigating to the empty directory, you can also redirect the output from the javadoc command using its -d switch.

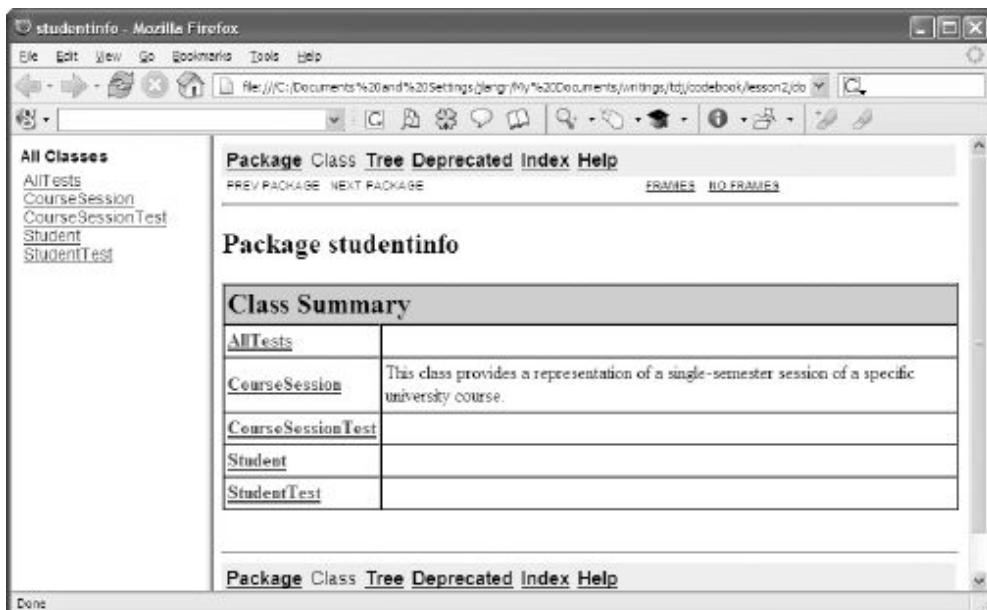


Figure 2.5 Your Own API Pages

Remove the comments for the constructor and the methods completely, but leave the comment for the class—it does provide a bit more value to the reader. Then rerun the javadoc command, bring up the web pages again, and see if you feel any useful information was lost. Your opinion may differ.

Exercises

1. Add a test to TestPawn that creates a pawn without a color. Why does this generate a compile error? (Hint: Think about default constructors.) Fix the compiler error by adding a second constructor that constructs a white pawn by default.
2. Make the constants for the two colors static and move them onto the Pawn class.
3. Pawns aren't very useful without a board. Use a test to define a Board class. Assert that the board starts with zero pieces on it. Follow the TDD sequence: Write the smallest test possible. Prove failure with a red bar or a compile error. Incrementally add small bits of code to obtain a clean compile or green bar.
4. Develop code to allow pawns to be added to the board. In a test, add a black and white pawn to the board. Each time you add a pawn, as-

sert that the piece count is correct. Also, each time you add a pawn, obtain a list of pieces from the board and ensure that it contains the expected pawn objects.

5. Write javadoc for each of the production classes and methods you have created so far. Be careful in doing so: Do not duplicate information that your methods already impart! The javadoc should be supplementary information.
6. Move the four tests and classes you have created into a package of their own. Name this package chess. Resolve compilation failures and get to a green bar again. Also, replace fully qualified class names for List and ArrayList by using an import statement.
7. Move TestPawn and Pawn into a package named pieces and resolve any problems discovered along the way.
8. Ensure that nothing other than a Pawn can be added to the board. Try adding a new Integer("7") to the list of pawns and see the resulting compiler error.
9. Create a test suite that runs each test class.
10. Look through the code you have written so far. Ensure there is no duplication anywhere in the code. Remember that the test code is code as well. Use the `setUp` method if appropriate.

Exercises

This page intentionally left blank

Lesson 3

Strings and Packages

In this lesson, you will:

- learn more about the String class
- learn how characters are represented in Java
- use a system property to ensure platform-independent code
- use StringBuilder objects to dynamically construct strings
- learn how to iterate through a collection to operate on each of its objects
- use System.out to display report output
- organize your classes into packages
- increase your understanding of the access modifiers public and private

Characters and
Strings

Characters and Strings

Strings, or pieces of text, account for up to 50 percent and more of the objects created during the execution of a typical Java application. While Strings are objects, they are made up of sequences of individual characters. Java represents a character as a primitive type known as `char`. Since a `char` value is of a primitive type (like an `int` value), remember that you cannot send messages to it.

Characters

Java includes a `char` type that represents letters, digits, punctuation marks, diacriticals, and other special characters. Java bases its character set on a standard known as Unicode 4.0 for representation of its characters. The

Unicode standard is designed to accommodate virtually all character variations in the major languages of the world. More information regarding the standard can be found at <http://www.unicode.org>.

Java uses two bytes to store each character. Two bytes is 16 bits, which means that Java can represent 2^{16} , or 65,536, characters. While that may seem like a lot, it's not enough to support everything in the Unicode standard. You probably won't need to concern yourself with supporting anything over the two-byte range, but if you do, Java allows you to work with characters as int values. An int is four bytes, so the billions of characters it can support should be sufficient until the Federation requires us to incorporate the Romulan alphabet.

You can represent character literals in Java in a few ways. The simplest form is to embed the actual character between single quotes (tics).

```
char capitalA = 'A';
```

Language Tests

Characters and Strings

While I present most of the code in Agile Java as part of the ongoing student example, I show some Java syntactical details and variations as brief code snippets and assertions. The single-line assertions in this section on characters provide an example. I refer to these as language tests—you write them to learn the language. You might keep them to reinforce your understanding of the language for later use.

You can code these tests wherever you like. I typically code them as separate test methods in the current test class, then delete them once I have an understanding of the element I was working on.

You may choose to create a separate class to contain these “scratch” tests. Ultimately, you might even create a scratch package, suite, and/or project of such tests.

You might get some reuse out of storing these tests: Some of the language tests you build may end up being the basis for utility methods that encapsulate and simplify use of a language feature.

Characters are essentially numerics. Each character maps to a corresponding positive integer from 0 through 65,535. Here is a test snippet that shows how the character 'A' has a numeric value of 65 (its Unicode equivalent).

```
assertEquals(65, capitalA);
```

Not all characters can be directly entered via the keyboard. You can represent Unicode characters using the Unicode escape sequence, \u or \U, followed by a 4-digit hex number.

```
assertEquals('\u0041', capitalA);
```

Additionally, you may represent characters as a 3-digit octal (base 8) escape sequence.

```
assertEquals('\101', capitalA);
```

The highest possible character literal that you may represent as an octal sequence is '\377', which is equivalent to 255.

Most older languages (for example, C) treat characters as single bytes. The most well-known standard for representing characters in a single-byte character set (SBCS), the American Standard Code for Information Interchange (ASCII), is defined by ANSI X3.4.¹ The first 128 characters of Unicode map directly to their ASCII correspondents.

Special Characters

Java defines several special characters that you can use for things such as output formatting. Java represents the special characters with an *escape sequence* that consists of the backslash character (\) followed by a mnemonic. The table below summarizes the char literals that represent these special characters.

Carriage return	'\r'
Line feed	'\n'
Tab	'\t'
Form feed	'\f'
Backspace	'\b'

Characters and
Strings

Since the tic character and the backslash character have special meanings with respect to char literals, you must represent them with an escape sequence. You may also escape (i.e., prefix with the escape character \) the double quote character, but you are not required to do so.

Single quote	'\'
Backslash	'\\'
Double quote	'\"'

¹In fact, ASCII is only a true standard for seven of the single byte's eight bits. Characters from 0 through 127 are consistently represented, but there are several competing standards for characters 128 through 255.

Strings

A String object represents a sequence of char values of fixed length. The String class in Java is probably the most frequently used class in any Java application. Even in a small application, thousands of String objects will be created over and over.

The String class supplies dozens of methods. It has special performance characteristics that make it different from most other classes in the system. Finally, even though String is a class like any other class in the system, the Java language provides special syntactical support for working with String objects.

You can construct Strings in a number of ways. Any time you create a new String literal, the Java VM constructs a String object behind the scenes. Here are two ways to construct a String object and assign it to a reference variable:

```
String a = "abc";
String b = new String("abc"); // DON'T DO THIS
```

Strings

Avoid the second technique.² It creates two String objects, which can degrade performance: First, the VM creates the literal String object "abc". Second, the VM constructs a new String object, passing the literal "abc" to its constructor. Equally as important, it is an unnecessary construct that makes your code more difficult to read.

Since strings are sequences of characters, they can embed special characters. The string literal in the following line of code contains a tab character followed by a line feed character.

```
String z = "\t\n";
```

String Concatenation

You may *concatenate* a string to another string to produce a third string.

```
assertEquals("abcd", "ab".concat("cd"));
```

String concatenation is such a frequent operation in Java that you can use the plus sign (+) as a shortcut for concatenating strings. In fact, most Java concatenations are written this way:

```
assertEquals("abcdef", "abc" + "def");
```

²[Bloch2001].

Since the result of concatenating two strings together is another string, you can code many + operations to concatenate several strings into a single string.

```
assertEquals("123456", "12" + "3" + "456");
```

In the previous lesson, you used + for addition of integers. Java also allows you to use the plus sign, known as an *operator*, to concatenate strings. Since the + operator has different meanings, depending on what you use it with, it is known as an *overloaded operator*.

String Immutability

As you browse through the Java API documentation for String, you will notice that there are no methods that change a string. You cannot alter the length of a string, nor can you alter any of the characters contained within a string. String objects are thus *immutable*. If you want to perform any manipulations on a string, you must create a new string. For example, when you concatenate two strings using +, the Java VM alters neither string. Instead, it creates a new string object.

Sun designed String to be immutable to allow it to act in a very optimized manner. This optimization is crucial due to the very heavy use of Strings in most applications.

StringBuilder

StringBuilder

Often you will need to be able to construct strings dynamically. The class `java.lang.StringBuilder` provides this capability. A newly created `StringBuilder` represents an empty sequence, or collection, of characters. You can add to this collection by sending the `append` message to the `StringBuilder` object.

Just as Java overloads the + operator to support both adding int values and concatenating strings, the `StringBuilder` class overloads the `append` method to take arguments of any base type. You can pass a character, a String, an int, or other types as arguments to an `append` message. Refer to the Java API documentation for the list of overloaded methods.

When you finish appending to the `StringBuilder`, you obtain a concatenated String object from the `StringBuilder` by sending it the `toString` message.



Users of the student information system need to produce a report showing the roster of a course session. For now, a simple text report with just a list of the student names in any order will suffice.

Code the following test in CourseSessionTest. The assertion shows that the report requires a simple header and a footer showing the count of students.

```
public void testRosterReport() {
    session.enroll(new Student("A"));
    session.enroll(new Student("B"));

    String rosterReport = session.getRosterReport();
    assertEquals(
        CourseSession.ROSTER_REPORT_HEADER +
        "A\nB\n" +
        CourseSession.ROSTER_REPORT_FOOTER + "2\n", rosterReport);
}
```

(Remember that `testRosterReport` uses the `CourseSession` object created in the `setUp` method in `CourseSessionTest`.) Update `CourseSession` with the corresponding code:

```
String getRosterReport() {
    StringBuilder buffer = new StringBuilder();

    buffer.append(ROSTER_REPORT_HEADER);

    Student student = students.get(0);
    buffer.append(student.getName());
    buffer.append('\n');

    student = students.get(1);
    buffer.append(student.getName());
    buffer.append('\n');

    buffer.append(ROSTER_REPORT_FOOTER + students.size() + '\n');

    return buffer.toString();
}
```

StringBuilder

For each of the two students, you pass a `String` (the student's name) to `append`, then you pass a `char` (line feed) to `append`. You also append header and footer information to the `StringBuilder` object stored in `buffer`. The name `buffer` implies that the `StringBuilder` holds on to a collection of characters that will be used later. The line that constructs the footer demonstrates how you can pass a concatenated string as the parameter to the `append` method.

You defined the `getRosterReport` method in the `CourseSession` class. Code within a class can refer directly to static variables. So instead of:

`CourseSession.ROSTER_REPORT_HEADER`

the `CourseSession` code uses:

`ROSTER_REPORT_HEADER`

When you learn more about the static keyword in Lesson 4, you will be told to refer to static variables and static methods only by scoping them with the class name (as in `CourseSession.ROSTER_REPORT_HEADER`), even from within the class they are defined. Doing otherwise obscures the fact that you are using static elements, which can lead to troublesome defects. In the case of class constants, however, the naming convention (`UPPERCASE_WITH_UNDERSCORES`) makes it explicitly clear that you are referring to a static element. The second, unscoped, form is thus acceptable (although some shops may prohibit it) and is an exception to the rule.

If you look at older Java code, you will see use of the class `java.lang.StringBuffer`. You interact with a `StringBuffer` object the same as with a `StringBuilder` object. The distinction between the two is that the `StringBuilder` class has better performance characteristics. It does not need to support *multithreaded* applications, where two pieces of code could be working with a `StringBuffer` simultaneously. See Lesson 13 for a discussion of multithreading.

System Properties

System
Properties

Both the method `getRosterReport` and its test contain the use of '`\n`' to represent a line feed in many places. Not only is this duplication, it is not portable—different platforms use different special character sequences for advancing to a new line on output. The solution to this problem can be found in the class `java.lang.System`. As usual, refer to the J2SE API documentation for a more detailed understanding of the `System` class.

The `System` class contains a method named `getProperty` that takes a system property key (a `String`) as a parameter and returns the system property value associated with the key. The Java VM sets several system properties upon startup. Many of these properties return information about the VM and execution environment. The API documentation method detail for `getProperties` shows the list of available properties.

One of the properties is `line.separator`. According to the Java API documentation, the value of this property under Unix is '`\n`'. However, under Windows, the value of the property is '`\r\n`'. You will use the `line.separator` system property in your code to compensate for the differences between platforms.

The following changes to the test and to `CourseSession` demonstrate use of the `System` method `getProperty`.

Test code:

```
public void testRosterReport()
{
    Student studentA = new Student("A");
```

```

Student studentB = new Student("B");
session.enroll(studentA);
session.enroll(studentB);

String rosterReport = session.getRosterReport();
assertEquals(
    CourseSession.ROSTER_REPORT_HEADER +
    "A" + CourseSession.NEWLINE +
    "B" + CourseSession.NEWLINE +
    CourseSession.ROSTER_REPORT_FOOTER + "2" +
    CourseSession.NEWLINE, rosterReport);
}

```

Production code:

```

class CourseSession {
    static final String NEWLINE =
        System.getProperty("line.separator");
    static final String ROSTER_REPORT_HEADER =
        "Student" + NEWLINE +
        "---" + NEWLINE;
    static final String ROSTER_REPORT_FOOTER =
        NEWLINE + "# students = ";

    ...
    String getRosterReport() {
        StringBuilder buffer = new StringBuilder();

        buffer.append(ROSTER_REPORT_HEADER);

        Student student = students.get(0);
        buffer.append(student.getName());
        buffer.append(NEWLINE);

        student = students.get(1);
        buffer.append(student.getName());
        buffer.append(NEWLINE);

        buffer.append(ROSTER_REPORT_FOOTER + students.size() + NEWLINE);

        return buffer.toString();
    }
}

```

**Looping
through All
Students**

Looping through All Students

The test method `testRosterReport` demonstrates how to produce a report for two students. You know that the code to construct the report is written with the assumption that there will be only two students.



You need to produce code that supports an unlimited number of students. To do this, you will modify your test to enroll additional students. Subsequently, you will recognize that the production class contains duplicate code—and the amount of duplication will only get worse—since the same three lines of code are repeated for each student, with the only variance being the index of the student.

What you would like to be able to do is to execute the same three lines of code for *each* of the students in the ArrayList, regardless of how many students the ArrayList contains. There are several ways of doing this in Java. The most straightforward means in J2SE 5.0 is to use a for-each loop.³

There are two forms of the for-each loop. The first form, which uses opening and closing braces following the loop declaration, allows you to specify multiple statements as the body of the for-each loop.

```
for (Student student: students) {
    // ... statements here ...
}
```

The second form allows you to define the body as a single statement only, and thus requires no braces:

```
for (Student student: students)
    // ... single statements here;
```

In Lesson 7, you will learn about another kind of for loop that allows you to loop a certain number of times instead of looping through every element in a collection.

The Java VM executes the body of the for loop once for each student in the collection students.

```
String getRosterReport() {
    StringBuilder buffer = new StringBuilder();

    buffer.append(ROSTER_REPORT_HEADER);

    for (Student student: students) {
        buffer.append(student.getName());
        buffer.append(NEWLINE);
    }

    buffer.append(ROSTER_REPORT_FOOTER + students.size() + NEWLINE);

    return buffer.toString();
}
```

**Looping
through All
Students**

A reading of the above for-each loop in English-like prose: Assign each object in the collection students to a reference of the type Student named student and execute the body of the for loop with this context.

³Also known as an enhanced for loop.

Single-Responsibility Principle

 New reports are continually needed in the student information system. You have been told that you must now produce three additional reports. And you can surmise that new reports will continue to be requested. You foresee the need to change the CourseSession class constantly as the reports are added.

One of the most basic design principles in object-oriented programming is that a class should do one thing and do it well. By virtue of doing this one thing, the class should have only one reason to change. This is known as the Single-Responsibility Principle.⁴



Classes should have only one reason to change.

The one thing CourseSession should be doing is tracking all information pertinent to a course session. Adding the capability to store professor information for the course session is a motivation that is in line with the primary goal of the class. Producing reports such as the roster report, however, is a different motivation for changing the CourseSession class and as such violates the Single-Responsibility Principle.

Single-Responsibility Principle

Create a test class, RosterReporterTest, to demonstrate how a new, separate class named RosterReporter can be used to produce a roster report.

```
package studentinfo;

import junit.framework.TestCase;
import java.util.*;

public class RosterReporterTest extends TestCase {
    public void testRosterReport() {
        CourseSession session =
            new CourseSession("ENGL", "101", createDate(2003, 1, 6));

        session.enroll(new Student("A"));
        session.enroll(new Student("B"));

        String rosterReport = new RosterReporter(session).getReport();
        assertEquals(
            RosterReporter.ROSTER_REPORT_HEADER +
            "A" + RosterReporter.NEWLINE +
            "B" + RosterReporter.NEWLINE +

```

⁴[Martin2003].

```

    RosterReporter.ROSTER_REPORT_FOOTER + "2" +
    RosterReporter.NEWLINE, rosterReport);
}

Date createDate(int year, int month, int date) {
    GregorianCalendar calendar = new GregorianCalendar();
    calendar.clear();
    calendar.set(Calendar.YEAR, year);
    calendar.set(Calendar.MONTH, month - 1);
    calendar.set(Calendar.DAY_OF_MONTH, date);
    return calendar.getTime();
}
}

```

The method `testRosterReport` is almost the same as it appeared in `CourseSessionTest`. The chief differences (highlighted in **bold**):

- You construct an instance of `RosterReporter` with a `CourseSession` object as a parameter.
- You now use class constants declared in `RosterReporter` instead of `CourseSession`.
- `testReport` constructs its own `CourseSession` object.

You should also recognize and make note of the duplication—both `CourseSessionTest` and `RosterReporterTest` require the `createDate` method. You will soon refactor this duplication away.

Add the new test to `AllTests`:

```

package studentinfo;

import junit.framework.TestSuite;

public class AllTests {
    public static TestSuite suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(StudentTest.class);
        suite.addTestSuite(CourseSessionTest.class);
        suite.addTestSuite(RosterReporterTest.class);
        return suite;
    }
}

```

**Single-
Responsibility
Principle**

Much of the work of getting the test to pass involves moving the code over from `CourseSession`. Do this incrementally—don't make any changes to `CourseSession` or `CourseSessionTest` until everything is working in `RosterReporter`.

```

package studentinfo;

import java.util.*;

class RosterReporter {
    static final String NEWLINE =
        System.getProperty("line.separator");
    static final String ROSTER_REPORT_HEADER =
        "Student" + NEWLINE +
        "----" + NEWLINE;
    static final String ROSTER_REPORT_FOOTER =
        NEWLINE + "# students = ";

    private CourseSession session;

    RosterReporter(CourseSession session) {
        this.session = session;
    }

    String getReport() {
        StringBuilder buffer = new StringBuilder();

        buffer.append(ROSTER_REPORT_HEADER);

        for (Student student: session.getAllStudents()) {
            buffer.append(student.getName());
            buffer.append(NEWLINE);
        }

        buffer.append(
            ROSTER_REPORT_FOOTER + session.getAllStudents().size() +
            NEWLINE);

        return buffer.toString();
    }
}

```

Single-Responsibility Principle

The **bold** code in the example above shows the significant differences between RosterReporter and the corresponding code in CourseSession.

To arrive at the code above, first paste the body of `getReport` from CourseSession directly into the corresponding method in RosterReporter. Then modify the pasted code to request the collection of students from CourseSession by sending the `getAllStudents` message instead of accessing it directly (since the method no longer executes in CourseSession). Since you removed `getAllStudents` in the previous lesson, you'll just have to add it back to CourseSession.

```

class CourseSession {
    ...
    ArrayList<Student> getAllStudents() {
        return students;
    }
    ...
}

```

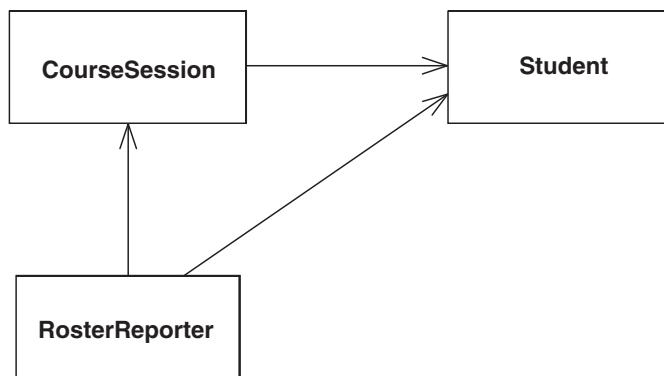


Figure 3.1 *Class Diagram*

Also, in order for code in `RosterReporter` to be able to send messages to the `CourseSession` object, it must store a `CourseSession` reference. You do this by assigning the `CourseSession` passed to the constructor of `RosterReporter` in the instance variable `session`.

Next, remove the report-related code from `CourseSessionTest` and `CourseSession`. This includes the test method `testRosterReport`, the production method `getRosterReport`, and the class constants defined by `CourseSession`. Rerun all tests.

The current class structure is show in Figure 3.1.

Refactoring

Refactoring

Both `CourseSessionTest` and `RosterReporterTest` require the `createDate` utility method. The code in `createDate` has nothing to do with course sessions or roster reports; it deals solely with constructing date objects. Including minor utility methods in classes is a mild violation of the Single-Responsibility Principle. You can tolerate small doses of duplication, but in doing so you quickly open the door to excessive, costly duplication in your system. In a larger system, there might be half a dozen methods that construct dates, all with pretty much the same code.

Here the duplication is obvious, since you directly created (and hopefully noted) it. An alternate approach is to not even let the duplication occur: As soon as you recognize that you might be introducing duplicate code, do the necessary refactoring *first* to stave off the potential duplication.

You will create a new test class and production class. You must update AllTests to reference the new test class. The code for all three follows.

```
// DateUtilTest.java
package studentinfo;

import java.util.*;
import junit.framework.*;

public class DateUtilTest extends TestCase {
    public void testCreateDate() {
        Date date = new DateUtil().createDate(2000, 1, 1);
        Calendar calendar = new GregorianCalendar();
        calendar.setTime(date);
        assertEquals(2000, calendar.get(Calendar.YEAR));
        assertEquals(Calendar.JANUARY, calendar.get(Calendar.MONTH));
        assertEquals(1, calendar.get(Calendar.DAY_OF_MONTH));
    }
}

// DateUtil.java
package studentinfo;

import java.util.*;

class DateUtil {
    Date createDate(int year, int month, int date) {
        GregorianCalendar calendar = new GregorianCalendar();
        calendar.clear();
        calendar.set(Calendar.YEAR, year);
        calendar.set(Calendar.MONTH, month - 1);
        calendar.set(Calendar.DAY_OF_MONTH, date);
        return calendar.getTime();
    }
}

// AllTests.java
package studentinfo;

import junit.framework.TestSuite;

public class AllTests {
    public static TestSuite suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(StudentTest.class);
        suite.addTestSuite(CourseSessionTest.class);
        suite.addTestSuite(RosterReporterTest.class);
        suite.addTestSuite(DateUtilTest.class);
        return suite;
    }
}
```

Refactoring

Previously, no tests existed for the createDate method, since it was only a utility for use in test classes themselves. When extracting code from one class

to a new class, you should always move along any tests that exist into the corresponding new test class. When tests do not exist, you should expend the time to create them. This will maintain the sustainability of your system.

Now that you have created and tested the DateUtil class, you want to update your code to refer to it. At the same time, you want to remove the createDate method from both CourseSessionTest and RosterReporterTest. One solid approach is to remove the createDate method from both places and recompile. The compiler will tell you precisely which lines of code refer to the nonexistent createDate method.



Use the compiler to help you refactor code.

Change these lines.

```
// CourseSessionTest
package studentinfo;

import junit.framework.TestCase;
import java.util.*;

public class CourseSessionTest extends TestCase {
    ...
    public void setUp() {
        startDate = new DateUtil().createDate(2003, 1, 6);
        session = new CourseSession("ENGL", "101", startDate);
    }
    ...
    public void testCourseDates() {
        Date sixteenWeeksOut = new DateUtil().createDate(2003, 4, 25);
        assertEquals(sixteenWeeksOut, session.getEndDate());
    }
}

// RosterReporterTest.java
package studentinfo;

import junit.framework.TestCase;

public class RosterReporterTest extends TestCase {
    public void testRosterReport() {
        CourseSession session =
            new CourseSession("ENGL", "101",
                new DateUtil().createDate(2003, 1, 6));
        ...
    }
}
```

Refactoring

In order to use the createDate utility method, you must construct a DateUtil object each time. In the case of CourseSession test, you construct a DateUtil object twice—a prime candidate for refactoring. You could create an instance

variable to hold onto a DateUtil instance. A better solution, however, is to convert the DateUtil to a static method—a method you can call without creating instances of DateUtil. You will learn how to do this in Lesson 4.

System.out

The getReport method returns a String that contains a report of all the students enrolled in a course session. In the production student information system, the String isn't going to be of much use to anyone unless you print it out or display it somewhere. Java provides output facilities to allow you to redirect information to the console, to files, and to other destinations. You will learn about these output facilities in depth in Lesson 11.

In this exercise you will modify the test so that the report displays on the console. It isn't yet a requirement, but sometimes you need to be able to display things for various reasons. The next section goes into some of these reasons.

System.out

In the Setup section of this book, you coded and ran a “Hello World” application that printed text to your console. The line of code to print the text on the console was:

```
System.out.println("hello world");
```

Look at the J2SE API documentation for the class named System, located in the package java.lang. You will see that out is a static variable, of the type PrintStream, that represents the standard output stream, also known as std-out or simply “the console.” You can directly access this console object using the following static variable reference:

```
System.out
```

Once you have this console object, you may send it a number of messages, including the message println. The println method takes a String (among other things) and writes it to the underlying output stream.

Add a line to RosterReporterTest that displays the report on the console by using System.out:

```
package studentinfo;  
  
import junit.framework.TestCase;  
  
public class RosterReporterTest extends TestCase {  
    public void testRosterReport() {  
        CourseSession session =
```

```

        new CourseSession("ENGL", "101",
            new DateUtil().createDate(2003, 1, 6));

    session.enroll(new Student("A"));
    session.enroll(new Student("B"));

    String rosterReport = new RosterReporter(session).getReport();
System.out.println(rosterReport);
    assertEquals(
        RosterReporter.ROSTER_REPORT_HEADER +
        "A" + RosterReporter.NEWLINE +
        "B" + RosterReporter.NEWLINE +
        RosterReporter.ROSTER_REPORT_FOOTER + "2" +
        RosterReporter.NEWLINE, rosterReport);
    }
}

```

Rerun your tests. You should see the actual report displayed onscreen. If you are running in an IDE, you may need to use `System.err` (the standard error output stream, also known as `syserr`), instead of `System.out`, in order to view the results.⁵

You'll note that I placed the additional line of code all the way to the left margin. I use this convention to remind me that the code is intended for temporary use. It makes such statements easy to locate and remove.

Revert these changes and rerun all tests once you have finished viewing the output.

Using
`System.out`

Using `System.out`

The most frequent use of `System.out` is to post messages to the console in an effort to locate defects in a program. You insert `System.out.println` statements to display useful information at judicious points in your code. When you execute the application, the output from these *trace statements* can help you understand the flow of messages and data through the objects interacting in the system.

Debuggers are far more sophisticated tools that accomplish the same goal and much more, but simple trace statements can occasionally be a more rapid and effective solution. Also, in some environments it is not feasible to use a debugger.

Regardless, you should find minimal need to debug your code, or even insert trace statements into it, if you do TDD properly. If you do the small

⁵The results might appear in a window named “console.”

steps that TDD prescribes, you will introduce very small amounts of code into your application before finding out you have a problem. Instead, the better solution is to discard the small amount of newly introduced code and start again, using even smaller verified steps.



Build your system in small increments of test and code. Discard an increment and start over with smaller steps if you have a problem.

Most developers do not write console-based applications, although you are probably familiar with many of them. The compiler `javac` itself is a console-based application. Simple server applications are often coded as console applications so that developers can easily monitor their output.

Refactoring

Refactoring

If you haven't already done so, remove the `testReport` method from `CourseSessionTest` and remove the corresponding production code from `CourseSession`.

The `writeReport` method is still short, but conceptually it is doing three things. To make understanding even more immediate, you can decompose the code in `writeReport` into three smaller methods, one each to construct the header, body, and footer of the report:

```
String getReport() {
    StringBuilder buffer = new StringBuilder();
    writeHeader(buffer);
    writeBody(buffer);
    writeFooter(buffer);

    return buffer.toString();
}

void writeHeader(StringBuilder buffer) {
    buffer.append(ROSTER_REPORT_HEADER);
}

void writeBody(StringBuilder buffer) {
    for (Student student: session.getAllStudents()) {
        buffer.append(student.getName());
        buffer.append(NEWLINE);
    }
}

void writeFooter(StringBuilder buffer) {
    buffer.append(
        ROSTER_REPORT_FOOTER + session.getAllStudents().size() + NEWLINE);
}
```

Package Structure

You use packages to arbitrarily group your classes. This grouping of classes, known as the *package structure*, will change over time as your needs change. Initially, your concern will be ease of development. As the number of classes grows, you will want to create additional packages for manageability reasons. Once you deploy the application, your needs may change: You may want to organize the packages to increase the potential for reuse or perhaps to minimize maintenance impact to consumers of the package.

So far, your classes have all ended up in one package, `studentinfo`. A typical way to start organizing packages is to separate the *user interface*—the part of the application that the end user interacts with—from the underlying classes that represent business objects and infrastructural objects. The `RosterReporter` class in the previous example could be construed as part of the user interface, as it produces output that the end user will see.

Your next task will be to first move the `studentinfo` package down a level so that it is in a package named `sis.reportinfo`. You will then separate the `RosterReporter` and `RosterReporterTest` classes into their own package named `report`.

First create a new subdirectory named `sis` (for “Student Information System”) at the same directory level as `studentinfo`. Beneath this directory, create a new subdirectory named `report`. Move the `studentinfo` subdirectory into the `sis` subdirectory. Move the `RosterReporter` and `RosterReporterTest` classes into the `report` subdirectory. Your directory structure should look something like:

```
source
|---sis
    |---studentinfo
    |---report
```

Next, you will change the package statements of all your classes. For the packages in the `report` subdirectory, use this package statement:

```
package sis.report;
```

For the packages in the `studentinfo` subdirectory, use this package statement:

```
package sis.studentinfo;
```

As you did in Lesson 2, remove all the class files (`*.class`), then recompile all your code. You will receive several errors. The problem is that the `RosterReporter` and `RosterReporterTest` classes are now in a separate package from

Package
Structure

the CourseSession and Student classes. They no longer have appropriate access to the classes in the other package.

Access Modifiers

You have already used the keyword `public` for JUnit classes and methods without an understanding of the full meaning of the keyword, other than that JUnit requires that test classes and methods be declared `public`. You also learned that instance variables can be declared `private` so that objects of other classes cannot access them.

The `public` and `private` keywords are known as *access modifiers*. You use access modifiers to control access to Java elements, including fields, methods, and classes. The access modifiers that are appropriate for a class are different than those that are appropriate for methods and fields.

By declaring a class as `public`, you allow classes in other packages to be able to `import` and refer directly to the class. The JUnit framework classes are located in various packages whose name starts with `junit`. In order for these JUnit classes to be able to instantiate your test classes, you must declare them as `public`.

Neither the CourseSession nor the Student class you built specified an access modifier. In the absence of an access modifier, a class has an access level of *package*, also known as default access. You can refer to a class with package-level access from other classes within the same package; however, classes in a different package cannot refer to the class.

For “safer” programming, the preferred tactic is to start at the most restrictive level and then open up access as needed. Exposing your classes too much can mean that clients can become unnecessarily dependent on the details of how you’ve put the system together. If you change the details, the clients could break. Also, you open your code up to being corrupted by providing too much access.



Protect your code as much as possible. Relax access modifiers only when necessary.

The CourseSession and Student classes currently have package-level access. You can keep them at that level until a class in another package requires access to them.

In order to get your code to compile, you will first have to add an `import` statement so that the compiler knows to look in the `studentinfo` package for the Student and CourseSession classes. The modification to RosterReporterTest is shown here:

```
package sis.report;

import junit.framework.*;
import sis.studentinfo.*;

public class RosterReporterTest extends TestCase {
    ...
}
```

Add the same `import` statement to `RosterReporter`.

The classes in the `studentinfo` package still have package-level access, so classes in the `reports` package will not be visible to them. Change the class declaration to be `public` for `Student`, `CourseSession`, and `DateUtil`, as shown in the example for `Student`:

```
package sis.studentinfo;

public class Student {
    ...
}
```

You will also receive a compilation error for `AllTests.java`. It no longer recognizes the class `RosterReporterTest`, since `RosterReporterTest` has been moved to a different package. For now, comment out that line in `AllTests.java`:

```
package sis.studentinfo;

import junit.framework.TestSuite;

public class AllTests {
    public static TestSuite suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(StudentTest.class);
        suite.addTestSuite(CourseSessionTest.class);
//        suite.addTestSuite(RosterReporterTest.class);
        suite.addTestSuite(DateUtilTest.class);
        return suite;
    }
}
```

**Access
Modifiers**

You will soon create a new `AllTests` for the `reports` package. Be careful when commenting out code—it's easy to forget why the code is commented out.

After recompiling, you will receive lots of errors for each message sent from the code in the `reports` package to `Student` and `CourseSession` objects. Like classes, the default access level for constructors (and methods) is package. Just as classes need to be `public` in order to be accessed from outside the package, methods and constructors also must be declared as `public`. Do so judiciously—you should never make blanket declarations of every method as `public`.

As a matter of style and organization, you may also want to move public methods so they appear before the non-public methods in the source file. The idea is that a client developer interested in your class will find the public methods—the methods they should be most interested in—first. With IDEs, this organization is not as necessary, as most IDEs provide a better way to organize and navigate through source for a class.

When finished, the production classes in `studentinfo` should look something like the following.

`Student.java:`

```
package studentinfo;

public class Student {
    private String name;

    public Student(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Access
Modifiers

`CourseSession.java:`

```
package studentinfo;

import java.util.*;

/**
 * This class provides a representation of a single-semester
 * session of a specific university course.
 * @author Administrator
 */
public class CourseSession {
    private String department;
    private String number;
    private ArrayList<Student> students = new ArrayList<Student>();
    private Date startDate;

    /**
     * Constructs a CourseSession starting on a specific date
     * @param startDate the date on which the CourseSession begins
     */
    public CourseSession(
        String department, String number, Date startDate) {
        this.department = department;
        this.number = number;
        this.startDate = startDate;
    }
}
```

```

String getDepartment() {
    return department;
}

String getNumber() {
    return number;
}

int getNumberOfStudents() {
    return students.size();
}

public void enroll(Student student) {
    students.add(student);
}

Student get(int index) {
    return students.get(index);
}

Date getStartDate() {
    return startDate;
}

public ArrayList<Student> getAllStudents() {
    return students;
}

/*
 * @return Date the last date of the course session
 */
Date getEndDate() {
    GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(startDate);
    final int sessionLength = 16;
    final int daysInWeek = 7;
    final int daysFromFridayToMonday = 3;
    int numberOfDays =
        sessionLength * daysInWeek - daysFromFridayToMonday;
    calendar.add(Calendar.DAY_OF_YEAR, numberOfDays);
    return calendar.getTime();
}
}

```

DateUtil.java:

```

package studentinfo;

import java.util.*;

public class DateUtil {
    public Date createDate(int year, int month, int date) {
        GregorianCalendar calendar = new GregorianCalendar();

```

Access
Modifiers

```

        calendar.clear();
        calendar.set(Calendar.YEAR, year - 1900);
        calendar.set(Calendar.MONTH, month - 1);
        calendar.set(Calendar.DAY_OF_MONTH, date);
        return calendar.getTime();
    }
}

```

Where Do the Tests Go?

You have been putting your test classes in the same package as your production classes so far; for example, `StudentTest` and `Student` both appear in the same `studentinfo` package. This is the easiest approach, but not the only one. Another approach is to create an equivalent test package for every production package. For example, you might have a package named `test.studentinfo` that contains tests for classes in `studentinfo`.

One advantage of putting the tests in the same package as the production code is that the tests have access to package-level details of the class they test. However, that level of visibility could be construed as a negative: As much as possible, you should design your tests to test a class using its public interface—what it makes publicly available. The more your test requires access to private information, the more tightly *coupled*, or dependent, it becomes to the actual implementation. The tight coupling means that it is more difficult to make modifications to the production class without adversely impacting the test.

You will still find occasion to assert against information from an object that you wouldn't otherwise expose publicly. You may want to combine test and production classes in the same package for that reason. If you find that this scheme results in too many classes in one directory, you can use Java's classpath to your advantage: Create the same directory structure in two different subdirectories and have the classpath refer to both subdirectories.

For example, suppose you compile your classes into `c:\source\sis\bin`. You can create a second class file location, `c:\source\sis\test\bin`. Subsequently, you can modify your build scripts (Ant makes this very easy) to compile *only* test classes into the `c:\source\sis\test\bin` directory. Everything else will compile into `c:\source\sis\bin`. You can then put both `c:\source\sis\bin` and `c:\source\sis\test\bin` on the classpath.

Using this scheme, the class file for `Student` would end up as `c:\source\sis\bin\studentinfo\Student.class`, and the class file for `StudentTest` would end up as `c:\source\sis\test\bin\studentinfo\StudentTest.class`. Both classes remain in the package `studentinfo`, yet each is in a separate directory.

**Access
Modifiers**

At this point, everything should compile. Your tests should also run, but don't forget that you commented out `RosterReporterTest`. It's time to add it back in.

Create a new class named AllTests in the sis.report package. Generally you will want a test suite in each package to ensure that all classes in the package are tested.⁶

```
package sis.report;

import junit.framework.TestSuite;

public class AllTests {
    public static TestSuite suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(RosterReporterTest.class);
        return suite;
    }
}
```

You can now remove the commented-out line from the class studentinfo.AllTests.

Create a class named AllTests in the sis package by placing its source file in the sis directory. This class will produce the combined test suite that ensures all classes in the application are tested.

**Access
Modifiers**

```
package sis;

import junit.framework.TestSuite;

public class AllTests {
    public static TestSuite suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(sis.report.AllTests.suite());
        suite.addTest(sis.studentinfo.AllTests.suite());
        return suite;
    }
}
```

Instead of sending the message addTestSuite to the suite, you send the message addTest. As a parameter, you pass along the results of sending the message suite to the appropriate AllTests class. Sending a message to a class instead of to an object will result in a *static method* being called. I will discuss static methods in the next lesson.

You will want to pass sis.AllTests to JUnit in order to run your entire test suite.

⁶There are other ways of managing test suites; your IDE may provide some assistance here. Also, refer to Lesson 12 for a dynamic way of gathering tests.

Using Ant

From here on out, I will be using an Ant script to do my compilations, now that I have more than two directories to compile. Ant is a platform-independent tool that allows you to create specifications of how your project should be built and deployed.

If you are using an IDE, you should be able to get it to build your entire codebase easily. Under Eclipse, for example, all of your source code is compiled automatically each time you save changes to Java source.

Regardless of whether or not you are using an IDE, you may want to use Ant in order to obtain IDE and platform independence. Other alternatives are to build a shell script or batch file as I demonstrated in Lesson 1. You can also use one of a number of available make tools. A make tool is a build tool very similar to Ant, but most make tools are very tightly bound to a specific operating system. Few make tools provide the ease of building Java applications that Ant does. Ant is the most effective way of doing builds in Java.

I highly recommend that you learn how to use Ant. Your IDE may suffice for your own personal needs, but it might not be sufficient in a team environment. If you work in a team environment, you will want a standardized way of building and deploying your application. Most development shops have standardized on Ant as a way of ensuring the system is built and deployed consistently and correctly.

See the sidebar “Getting Started with Ant” for a brief overview of using Ant.

Using Ant

Getting Started With Ant

This sidebar will help you obtain a basic understanding of how to work with the Ant build tool.

Most Java IDEs come with Ant support already built in. If you are not using an IDE, you can follow these steps to be able to use Ant for your builds.

- Download the latest version of Ant from <http://ant.apache.org>.
- Follow the instructions available with the Ant distribution to install Ant. Set the environment variable `JAVA_HOME` to the directory in which you installed the J2SE 5.0 SDK.
- Update your system’s path environment variable to include Ant’s `bin` directory.
- Create a `build.xml` file in the root directory of your project.

Regardless of whether you are using an IDE or not, you will put together a `build.xml` file that contains the instructions on how to compile, execute, and/or deploy your application.

Here is a starter build.xml for a project named agileJava.

```
<?xml version="1.0"?>

<project name="agileJava" default="junitgui" basedir=".">
    <property name="junitJar" value="\junit3.8.1\junit.jar" />
    <property name="src.dir" value="${basedir}\source" />
    <property name="build.dir" value="${basedir}\classes" />

    <path id="classpath">
        <pathelement location="${junitJar}" />
        <pathelement location="${build.dir}" />
    </path>

    <target name="init">
        <mkdir dir="${build.dir}" />
    </target>

    <target name="build" depends="init" description="build all">
        <javac
            srcdir="${src.dir}" destdir="${build.dir}"
            source="1.5"
            deprecation="on" debug="on" optimize="off" includes="**">
            <classpath refid="classpath" />
        </javac>
    </target>

    <target name="junitgui" depends="build" description="run junit gui">
        <java classname="junit.awtui.TestRunner" fork="yes">
            <arg value="sis.AllTests" />
            <classpath refid="classpath" />
        </java>
    </target>

    <target name="clean">
        <delete dir="${build.dir}" />
    </target>

    <target name="rebuildAll" depends="clean,build" description="rebuild all"/>
</project>
```

Using Ant

Understanding the Sample Build File

Ant allows you to define in XML how to build various *targets* within a *project*. A target has a name and may have one or more other targets as dependencies:

```
<target name="rebuildAll" depends="clean,build" />
```

The above line defines a target named rebuildAll. When you execute this target (keep reading), Ant first ensures that the targets named clean and build have been executed.

A target contains a list of commands, or tasks, to execute. The Ant software installation provides a manual describing a large number of tasks that will suffice for most of your needs. If you can't find an appropriate task, you can programmatically create your own.

The clean target contains the single task named delete. In this example, the delete task tells Ant to delete a file system directory with the name provided in quotes.

```
<target name="clean">
  <delete dir="${build.dir}" />
</target>
```

Ant allows you to define properties that provide a construct similar to constants in Java. When the delete task executes, Ant will replace \${build.dir} with the value of the property named build.dir. The property named build.dir is declared in the agileJava Ant script as:

```
<property name="build.dir" value="${basedir}\classes" />
```

This declaration sets the value of build.dir to \${basedir}\classes. The use of \${basedir} is in turn a reference to the property basedir, defined in the project element for the agileJava Ant script:

```
<project name="agileJava" default="junitgui" basedir=".">>
```

(The . indicates that basedir is set to the current directory—the directory in which Ant is executed.)

When executing Ant, you can specify a target:

```
ant rebuildAll
```

The default attribute in the project element indicates the target to execute if none is specified. In this example, the junitgui target is the default and gets run if you execute ant with no arguments:

```
ant
```

A list of targets can be obtained by executing:

```
ant -projecthelp
```

This will show main targets—those targets that specify a description attribute.

Using some built-in smarts, Ant executes tasks only when necessary. For example, if you execute the junitgui target, Ant will only run javac compiles against source that has not changed since the last time you executed junitgui. It uses the timestamps of the class files to make this determination.

To summarize the agileJava project, there are three main targets: build, junitgui, and rebuildAll. There are two subtargets, init and clean.

The build target depends on the init target, which ensures that the build output directory (./classes) exists. The build target compiles all sources in the source directory (./source) to the build output directory, using Ant's built-in javac task. The javac task specifies a number of attributes, including the attribute classpath, specified as a nested element of the javac task. The classpath attribute references a path

Using Ant

element by the name classpath; this path element includes the JUnit jar file and the classes directory.

The junitgui target depends on the build target. If the build target succeeds, the junitgui target executes the JUnit GUI using the Java VM, passing in AllTests as the parameter.⁷

The rebuildAll target depends on execution of the clean target, which removes the build output directory, and on the build target.

Refer to the Ant manual for more detailed information. There are also several books available on Ant. One very comprehensive book is *Java Development with Ant*.⁸

Exercises

1. Create a CharacterTest class. Don't forget to add it to the AllSuites class. Observe the zero tests failure. Then add a test named testWhiteSpace. This test should demonstrate that the new line character, the tab character, and the space character all return true for Character.isWhiteSpace. Express that other characters return false. Can you find another character that returns true?
2. Java has certain naming restrictions—the names you give to methods, classes, variables, and other entities. For example, you cannot use the caret (^) in an identifier name. The Character class contains methods that designate whether or not a character can be used in an identifier. Consult your API documentation to understand these methods. Then add tests to the CharacterTest class to discover some of the rules regarding Java identifiers.
3. Assert that a black pawn's printable representation is the uppercase character 'P', and a white pawn's is the lowercase character 'p'. For the time being, you can accomplish this by adding a second parameter to the Pawn constructor. However, note that this creates a redundancy in representation. You'll improve upon the solution later.
4. (This exercise and Exercise 5 are closely related. You may suspend refactoring until you have completed Exercise 5.) When a client creates a Board object, they should be able to assume that the board is already initialized, with pieces in place. You will need to modify the

Exercises

⁷There is also an optional Ant task named junit that executes a text-based JUnit.

⁸[Hatcher2002].

Board tests and code accordingly. Start by changing the assertions on the number of pieces available at time of board creation: There should be 16. Delete `testAddPawns`; it is not useful in its current form.

5. Add an `initialize` method to `Board`. The `initialize` method should add pawns to create two ranks: a rank for white pawns (the second rank) and a rank for black pawns (the seventh rank). To store a rank, use an `ArrayList` whose contents are `Pawn` objects. You declare such a list as `ArrayList<Pawn>`.

Add an assertion to `testCreate` that the second rank looks like this: "pppppppp". Assert that the seventh rank looks like this: "PPPPPPPP". Use a `StringBuilder` and a `for` loop to gather the printable representation for the pieces in each rank.

Ensure your solution is as well refactored as you are capable of. Expect a good amount of duplication in adding pawns to the ranks and in other areas of `Board`. You will learn to eliminate this duplication in later lessons.

6. Assert that the board looks like this at initial setup, with a dot character (a period) representing an empty square (rank 8 is the top row, rank 1 is the bottom row):

```
.....  
PPPPPPPP  
.....  
.....  
.....  
.....  
pppppppp  
.....
```

Remember to ensure portability in your tests and in your board-printing method by using the system properties appropriately.

7. If you have implemented your chess board code and tests to this point using String concatenation, change the code to use the `StringBuilder` class. If you have used `StringBuilder` as the primary basis for your solution, change your code to use String concatenation. Name some differences in the structure and readability of the code.
8. Modify your test to display the board on the console. Ensure that it appears as expected. If not, correct the test and fix the code.
9. You will need to revisit this code in later exercises to remove even more duplication when they learn about loops and other Java constructs.
10. Create an Ant build file to compile your entire project and run all of your tests with a single command.

Exercises

Lesson 4

Class Methods and Fields

In this lesson you will:

- refactor an instance method to a class method
- learn about class variables and methods
- use the static import facility
- learn how to use compound assignment and increment operators
- understand what simple design is
- create a utility method
- learn how to judiciously use class methods
- work with the boolean primitive type
- understand why it is important that tests act as documentation
- be exposed to exceptions and stack traces
- learn more about initialization

Class Methods

Class Methods

Objects are a combination of behavior (implemented in Java in terms of methods) and attributes (implemented in Java in terms of fields). Attributes for an object stick around as long as the object sticks around. At any given point in time, an object has a certain *state*, which is the combined snapshot of all its instance variables. For this reason, instance variables are sometimes called state variables.

Action methods in the object may operate on and change attributes of the object. In other words, action methods can alter the object state. *Query methods* return pieces of the object state.



Design your methods to either change object state or return information, not both.

Occasionally you will find the need for a method that can take parameters, operate on only those parameters, and return a value. The method has no need to operate on object state. This is known as a utility method. Utility methods in other languages are sometimes called functions. They are *global*: any client code can access them.

Sometimes, having to create an object in order to use a utility method makes little sense. For example, the DateUtil method createDate you coded in Lesson 3 is a simple function that takes month, day, and year integer arguments and returns a Date object. The createDate method changes no other data. Removing the need to construct DateUtil objects will also simplify your code a bit. Finally, since createDate is the only method in DateUtil, there is no other need to construct instances of it.

For these reasons, createDate is a good candidate for being a class method. In this exercise, you will refactor createDate to be a class method in this exercise. Start by changing the test to make a class method call:

```
package sis.studentinfo;

import java.util.*;
import junit.framework.*;

public class DateUtilTest extends TestCase {
    public void testCreateDate() {
        Date date = DateUtil.createDate(2000, 1, 1);
        Calendar calendar = new GregorianCalendar();
        calendar.setTime(date);
        assertEquals(2000, calendar.get(Calendar.YEAR));
        assertEquals(Calendar.JANUARY, calendar.get(Calendar.MONTH));
        assertEquals(1, calendar.get(Calendar.DAY_OF_MONTH));
    }
}
```

You no longer create an instance of DateUtil with the `new` operator. Instead, to call a class method, you specify the class on which the class method is defined (DateUtil), followed by the dot operator (`.`), followed by the method name and any arguments (`createDate(2000, 1, 1)`).

Changes to the DateUtil class itself are similarly minor:

```
package sis.studentinfo;

import java.util.*;

public class DateUtil {
    private DateUtil() {}
}
```

Class Methods

```

public static Date createDate(int year, int month, int date) {
    GregorianCalendar calendar = new GregorianCalendar();
    calendar.clear();
    calendar.set(Calendar.YEAR, year);
    calendar.set(Calendar.MONTH, month - 1);
    calendar.set(Calendar.DAY_OF_MONTH, date);
    return calendar.getTime();
}
}

```

You declare a class method just like a “regular,” or instance, method, except that you prefix its declaration with the keyword `static`.

In addition to making the `createDate` method static, it’s a good idea to make the constructor of `DateUtil` private. By declaring the constructor as `private`, only code in the `DateUtil` class can construct new `DateUtil` instances. No other code will be able to do so. While it wouldn’t be harmful to allow creation of `DateUtil` objects, keeping clients from doing something nonsensical and useless is a good idea.

Adding the private constructor will also make it simpler for you to pinpoint the nonstatic references to `createDate`. When you compile your code, methods that create a new `DateUtil` object will generate compilation errors. For example, the `setUp` method in `CourseSessionTest` will fail compilation:

```

public void setUp() {
    startDate = new DateUtil().createDate(2003, 1, 6);
    session = new CourseSession("ENGL", "101", startDate);
}

```

Class Methods

Update it to call `createDate` statically:

```

public void setUp() {
    startDate = DateUtil.createDate(2003, 1, 6);
    session = new CourseSession("ENGL", "101", startDate);
}

```

Fix the remainder of the failing compilation problems and rerun your tests. You now have a general-purpose utility that may find frequent use in your system.¹

The class `java.lang.Math` in the J2SE 5.0 class library supplies many mathematical functions. For example, `Math.sin` returns the sine of a double and `Math.toRadians` converts a double value from degrees to radians. The `Math` class

¹The utility is not the best-performing one. It is not necessary to create a `GregorianCalendar` object with each call to `createDate`. For sporadic use, this is probably just fine. For heavy use—say, creating 10,000 dates upon reading an input file—you’ll want to consider caching the calendar object using a class variable (see the next section).

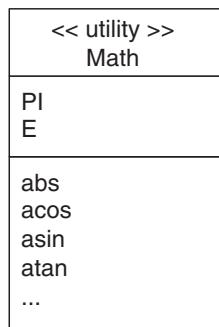


Figure 4.1 The Math Utility Class

also provides two standard mathematical constants, `Math.PI` and `Math.E`. Since each of the methods in `java.lang.Math` is a utility class method, it is known as a utility class.

In UML (Figure 4.1), you indicate a utility class using the stereotype `<<utility>>`. Stereotypes in UML define semantics beyond the limitations of what UML supplies. A utility stereotype specifies that all class behaviors and attributes may be globally accessed.

Normally you underline class behaviors and class attributes in UML. Since the `<<utility>>` stereotype declares that *all* methods and attributes in a class are global, you need not underline them.

Class Methods

Static Initialization Blocks

Constructors execute when you create instances of a class. You use constructors for more complex instance initialization.

Occasionally you may need more complex initialization to occur at a class level. You can use a *static initialization block* to accomplish this goal. Code within a static initialization block executes when the Java VM first loads a class.

```

import java.util.Date;
public class St {
    static {
        long now =
            System.currentTimeMillis();
        then = new Date(now + 86400000);
    }
    public static Date then;
}
  
```

To define a static initialization block, you precede a block of code (`{ . . . }`) with the keyword `static`. Place it within a class definition but external to any method or constructor. Virtually any code can appear within a static initialization block, but the block should not throw any exceptions (see Lesson 8).

Class Variables

You will occasionally want to track information about all instances of a class or perform an operation without first creating an instance of an object. As a simplistic example, you might want to track the total number of course sessions. As each `CourseSession` object is created, you want to bump up a counter. The question is, where should you put this counter? You could provide an instance variable on `CourseSession` to track the count, but this is awkward: Would all instances of `CourseSession` have to track the count? How would one `CourseSession` instance know when others were created so that it could update the count?

You could provide another class, `CourseSessionCounter`, whose sole responsibility is to track the `CourseSession` objects created. But a new class seems like overkill for the simple goal you are trying to accomplish.

In Java, you can use *class variables*, as opposed to instance variables, for a solution. Client code can access a class variable without first creating an instance of that class. Class variables have what is known as *static scope*: they exist as long as the class exists, which is pretty much from the time the class is first loaded until your application terminates.

You have already seen class constants in use. Class constants are class variables that you have designated as `final`.

The following test code (in `CourseSessionTest`) counts the number of `CourseSession` instances created:

```
public void testCount() {
    CourseSession.count = 0;
    createCourseSession();
    assertEquals(1, CourseSession.count);
    createCourseSession();
    assertEquals(2, CourseSession.count);
}

private CourseSession createCourseSession() {
    return new CourseSession("ENGL", "101", startDate);
}
```

Class Variables

(Don't forget to update the `setUp` method to use `createCourseSession`.)

To support the test, create a class variable in the `CourseSession` class named `count`. You use the `static` keyword to designate a variable as static in scope. Also add code to `CourseSession` to update `count` when a new `CourseSession` instance is created.

```
public class CourseSession {
    // ...
    static int count;
```

```

public CourseSession(
    String department, String number, Date startDate) {
    this.department = department;
    this.number = number;
    this.startDate = startDate;
    CourseSession.count = CourseSession.count + 1;
}
// ...

```

You access the class variable `count` similar to the way you call a class method: First specify the class name (`CourseSession`), followed by the dot (.) operator, followed by the variable name (`count`). The Java VM does not create an instance of `CourseSession` when code accesses the class variable.

As I mentioned, class variables have a different lifetime than instance variables. Instance variables stick around for the lifetime of the object that contains them. Each new `CourseSession` object that the Java VM creates manages its own set of the instance variables declared in `CourseSession`. When the VM creates a `CourseSession` object, it initializes its instance variables.

A class variable, however, comes into existence when the Java VM first loads the containing class—when code that is currently executing first references the class. There is one copy of the class variable in memory. The first time the Java VM loads a class, it initializes its class variables, and that's it. If you need to reset a class variable to an initial state at a later time, you must explicitly initialize it yourself.

As an experiment, comment out the first line in `testCount` (the line that reads `CourseSession.count = 0`). Then run the tests in JUnit. Turn off the checkbox in JUnit that says “Reload classes every run.”² If you run the tests twice (by clicking the Run button), they will fail, and you should see the actual count go up with each execution of the tests. You may even see the first run of the test fail: Other test methods in `CourseSessionTest` are creating `CourseSession` objects, which increments the `count` variable.

**Operating on
Class Variables
with Class
Methods**

Operating on Class Variables with Class Methods

Just as it is bad form to expose instance variables of your objects directly to prying clients, it is also bad form to expose class variables publicly. The notable exception is the class constant idiom—but there are even good reasons to avoid using class constants that you will learn in Lesson 5.

²This JUnit switch, when turned on, results in your test classes being physically loaded from disk and reinitialized each time the tests are run in JUnit. If you are running in an IDE such as Eclipse, you may not have control over this JUnit feature.

In addition to being able to use class methods for utility purposes, you can use class methods to operate on static data.

The CourseSession method testCount accesses the count class variable directly. Change the test code to ask for the count by making a class method call.

```
public void testCount() {
    CourseSession.count = 0;
    createCourseSession();
    assertEquals(1, CourseSession.getCount());
    createCourseSession();
    assertEquals(2, CourseSession.getCount());
}
```

Then add a class method to CourseSession that returns the class variable count.

```
static int getCount() {
    return count;
}
```

Class methods can access class variables directly. You should not specify the class name when accessing a class variable from a class method.

The Java VM creates no instances of CourseSession as a result of calling the class method. This means that class methods on CourseSession may not access any of the instance variables that CourseSession defines, such as department or students.

The test method still refers directly to the count class variable, however, since you need it initialized each time the test is run:

```
public void testCount() {
    CourseSession.count = 0;
    ...
```

Change the code in CourseSessionTest to make a static message send to reset the count:

```
public void testCount() {
    CourseSession.resetCount();
    createCourseSession();
    assertEquals(1, CourseSession.getCount());
    createCourseSession();
    assertEquals(2, CourseSession.getCount());
}
```

Add the resetCount method to CourseSession and make the count class variable private:

```
public class CourseSession {
    // ...
    private static int count;
    // ...
    static void resetCount() {
```

**Operating on
Class Variables
with Class
Methods**

```

        count = 0;
    }

    static int getCount() {
        return count;
    }
    // ...
}

```

Making the `count` variable `private` will point out (when you recompile) any other client code that directly accesses it.

The `testCount` method, which documents how a client should use the `CourseSession` class, is now complete and clean. But the `CourseSession` class itself still accesses the class variable directly in its constructor. Instead of accessing static data directly from a *member* (an instance-side constructor, field, or method), a better approach is to create a class method that you call from the instance side. This is a form of encapsulation that will give you greater control over what happens to the class variable.

Change the `CourseSession` constructor to send the `incrementCount` message instead of accessing the class variable directly:

```

public CourseSession(String department, String number, Date startDate) {
    this.department = department;
    this.number = number;
    this.startDate = startDate;
    CourseSession.incrementCount();
}

```

Then add a class method to `CourseSession` that increments the `count`. Declare this method as `private` to prevent other classes from incrementing the counter, which could compromise its integrity:

```

private static void incrementCount() {
    count = count + 1;
}

```

It is possible to access a class method or variable from the instance side of a class without specifying the class name. As an example, you could change the constructor code to the following:

```

public CourseSession(String department, String number, Date startDate) {
    this.department = department;
    this.number = number;
    this.startDate = startDate;
    incrementCount(); // don't do this!
}

```

Even though it will work, avoid doing this. Accessing class methods without using the class name introduces unnecessary confusion in your code and is considered bad form. Is `incrementCount` a class method or an instance method?

**Operating on
Class Variables
with Class
Methods**

Since it's not possible to tell from looking at the code in the `CourseSession` constructor alone, the intent is not clear. The expectation that a method is an instance method when it is in reality a class method can lead to some interesting problems.



Scope a class method call with the class name when invoking the class method from anywhere but another class method on the same class.

Static Import

I just told you to not call class methods from the instance side unless you supply the class name. Doing so obscures where the class method is defined. The same applies for accessing class variables (not including class constants).

Java permits you to muddle things even further. Including a *static import* in a class allows you to use class methods or variables defined in a different class as if they were defined locally. In other words, a static import allows you to omit the class name when referring to static members defined in another class.

There are appropriate uses for static import and inappropriate uses. I'll demonstrate an inappropriate use first. Modify `CourseSessionTest`:

```
// avoid doing this
package sis.studentinfo;

import junit.framework.TestCase;
import java.util.*;
import static sis.studentinfo.DateUtil.*; // poor use of static import

public class CourseSessionTest extends TestCase {
    private CourseSession session;
    private Date startDate;

    public void setUp() {
        startDate = createDate(2003, 1, 6); // poor use of static import
        session = CourseSession.create("ENGL", "101", startDate);
    }
    ...
    public void testCourseDates() {
        // poor use of static import:
        Date sixteenWeeksOut = createDate(2003, 4, 25);
        assertEquals(sixteenWeeksOut, session.getEndDate());
    }
    ...
}
```

Static Import

A static import statement looks similar to a regular import statement. However, a regular import statement imports one or all classes from a package, while a static import statement imports one or all class members (variables or methods) from a class. The above example imports all class members from the class DateUtil. Since there is only one class method in DateUtil, you could have explicitly imported just that method:

```
import static sis.studentinfo.DateUtil.createDate;
```

If DateUtil were to contain more than one class method with the name createDate (but with different argument lists), or if it also were to contain a class variable named createDate, they would each be statically imported.

Statically importing methods to avoid having to provide a class name in a few spots is lazy and introduces unnecessary confusion. Just where is createDate defined? If you are coding a class that requires quite a few external class method calls (perhaps a couple dozen or more), you might have an excuse to use static import. But a better approach would be to question why you need to make so many static calls in the first place and perhaps revisit the design of the other class.

A similar, potentially legitimate use of static import is to simplify the use of several related class constants that are gathered in a single place. Suppose you've created several report classes. Each report class will need to append new line characters to the output, so each report class will need a NEWLINE constant such as the one currently defined in RosterReporter:

```
static final String NEWLINE = System.getProperty("line.separator");
```

You don't want the duplication of defining this constant in each and every report class. You might create a new class solely for the purpose of holding this constant. Later it might hold other constants such as the page width for any report.

```
package sis.report;

public class ReportConstant {
    public static final String NEWLINE =
        System.getProperty("line.separator");
}
```

Since the NEWLINE constant will be used in a lot of places in a typical report class, you can add a static import to clean up your code a little:³

³You can eliminate the need for the NEWLINE constant entirely in a few other ways. You'll learn about one such technique, using the Java Formatter class, in Lesson 8.

Static Import

```

package sis.report;

import junit.framework.TestCase;
import sis.studentinfo.*;
import static sis.report.ReportConstant.NEWLINE;

public class RosterReporterTest extends TestCase {
    public void testRosterReport() {
        CourseSession session =
            CourseSession.create(
                "ENGL", "101", DateUtil.createDate(2003, 1, 6));

        session.enroll(new Student("A"));
        session.enroll(new Student("B"));

        String rosterReport = new RosterReporter(session).getReport();
        assertEquals(
            RosterReporter.ROSTER_REPORT_HEADER +
            "A" + NEWLINE +
            "B" + NEWLINE +
            RosterReporter.ROSTER_REPORT_FOOTER + "2" +
            NEWLINE, rosterReport);
    }
}

```

You can make similar changes to the RosterReporter class.

Putting a bunch of constants in a class with no behavior (methods) represents questionable OO design. Classes don't exist in a vacuum; the constants in ReportConstants class may be better off as part of another "normal" Java class, such as a class named Report.

Additional notes on static import:

Static Import

- It is not possible to statically import all members from all classes in a given package in a single statement. That is, you cannot code:

```
import static java.lang.*; // this does not compile!
```

- If a local method has the same signature as a statically imported method, the local method is called.

Use static imports with prudence. They make it more difficult to understand your classes by obscuring where members are defined. The rule of thumb is to limit use of static imports to things that are both universal and pervasive in your application.

Incrementing

In the `incrementCount` method, you coded:

```
count = count + 1;
```

The right-hand side of the statement is an expression whose value is one plus whatever value that `count` references. On execution, Java stores this new value back into the `count` variable.

Adding a value to a variable is a common operation, so common that Java supplies a shortcut. The following two statements are equivalent:

```
count = count + 1;  
count += 1;
```

The second statement demonstrates *compound assignment*. The second statement adds the value (1) on the right hand side of the compound assignment operator (`+=`) to the value referenced by the variable on the left hand side (`count`); it then assigns this new sum back to the variable on the left hand side (`count`). Compound assignment works for any of the arithmetic operators. For example,

```
rate *= 2;
```

is analogous to:

```
rate = rate * 2;
```

Adding the value 1 to an integer variable, or *incrementing* it, is so common that there is an even shorter cut. The following line uses the *increment operator* to increase the value of `count` by one:

```
++count;
```

The following line uses the *decrement operator* to decrease the value of `count` by one:

```
--count;
```

You could code either of these examples with plus or minus signs after the variable to increment:

```
count++;
```

```
count--;
```

The results would be the same. When they are used as part of a larger expression, however, there is an important distinction between the *prefix* oper-

Incrementing

ator (when the plus or minus signs appear before the variable) and the *postfix* operator (when the plus or minus signs appear after the variable).

When the Java VM encounters a prefix operator, it increments the variable *before* it is used as part of a larger expression.

```
int i = 5;
assertEquals(12, ++i * 2);
assertEquals(6, i);
```

When the Java VM encounters a postfix operator, it increments the variable *after* it is used as part of a larger expression.

```
int j = 5;
assertEquals(10, j++ * 2);
assertEquals(6, j);
```

Modify the code in CourseSession to use an increment operator. Since you're incrementing count all by itself, and not as part of a larger expression, it doesn't matter whether you use a pre-increment operator or a post-increment operator.

```
private static void incrementCount() {
    ++count;
}
```

Recompile and retest (something you should have been doing all along).

Factory
Methods

Factory Methods

You can modify CourseSession so that it supplies a static-side factory method to create CourseSession objects. By doing so, you will have control over what happens when new instances of CourseSession are created.

Modify the CourseSessionTest method `createCourseSession` to demonstrate how you will use this new factory method.

```
private CourseSession createCourseSession() {
    return CourseSession.create("ENGL", "101", startDate);
}
```

In CourseSession, add a static factory method that creates and returns a new CourseSession object:

```
public static CourseSession create(
    String department,
    String number,
    Date startDate) {
    return new CourseSession(department, number, startDate);
}
```

Find all other code that creates a `CourseSession` via `new CourseSession()`. Use the compiler to your advantage by first making the `CourseSession` constructor private:

```
private CourseSession(
    String department, String number, Date startDate) {
    // ...
```

Replace the `CourseSession` constructions you find (there should be one in `RosterReporterTest` and one in `CourseSessionTest`) with message sends to the static factory method. With the `CourseSession` constructor declared as `private`, no client code (which includes test code) will be able to create an instance of `CourseSession` directly using a constructor. Clients must instead use the static factory method.

Joshua Kerievsky names the above refactoring “Replace [Multiple⁴] Constructors with Creation Methods.”⁵ The class methods are the creation methods. The significant benefit is that you can provide descriptive names for creation methods. Since you must name a constructor the same as the class, it cannot always impart enough information for a developer to know its use.

Now that you have a factory method to create `CourseSession` objects, tracking the total count can be done on the static side, where it belongs. Much of good object-oriented design is about putting code where it belongs. This doesn’t mean that you must always start with code in the “right place,” but you should move it there as soon as you recognize that there is a better place for it to go. It makes more sense for `incrementCount` message sends to be made from the static side:

```
private CourseSession(
    String department, String number, Date startDate) {
    this.department = department;
    this.number = number;
    this.startDate = startDate;
}

public static CourseSession create(
    String department,
    String number,
    Date startDate) {
    incrementCount();
    return new CourseSession(department, number, startDate);
}
```

⁴Java allows you to code multiple constructors in a class. The descriptive names of creation methods are far more valuable in this circumstance, since they help a client developer determine which to choose.

⁵[Kerievsky2004].

Simple Design

Software development purists will tell you that you could have saved a lot of time by thinking through a complete design in the first place. With enough foresight, you might have figured out that static creation methods were a good idea and you would have put them in the code in the first place. Yes, after considerable experience with object-oriented development, you will learn how to start with a better design.

However, more often than not, the impact of design is not felt until you actually begin coding. Designers who don't validate their design in code frequently produce an overblown system by doing things such as adding static creation methods where they aren't warranted. They also often miss important aspects of design.

The best tactic to take is to keep your code as clean as possible at all times. The rules to keep the design clean are, in order of importance:

- Make sure your tests are complete and always running 100 percent green.
- Eliminate duplication.
- Ensure that the code is clean and expressive.
- Minimize the number of classes and methods.

The code should also have no more design in it than is necessary to support the current functionality. These rules are known as *simple design*.⁶

Static Dangers

Simple design will give you the flexibility you need to update the design as requirements change and as you require design improvements. Creating a static factory method from a constructor wasn't all that difficult, as you saw; it is easily and safely done when you follow simple design.

Static Dangers

Using statics inappropriately can create significant defects that can be difficult to resolve. A classic novice mistake is to declare attributes as class variables instead of instance variables.

The class `Student` defines an instance variable named `name`. Each `Student` object should have its own copy of `name`. By declaring `name` as static, every `Student` object will use the same copy of `name`:

⁶[Wiki2004b].

```
package sis.studentinfo;

public class Student {
    private static String name;

    public Student(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

A test can demonstrate the devastating effect this will have on your code:

```
package sis.studentinfo;

import junit.framework.*;

public class StudentTest extends TestCase {
    ...
    public void testBadStatic() {
        Student studentA = new Student("a");
        assertEquals("a", studentA.getName());
        Student studentB = new Student("b");
        assertEquals("b", studentB.getName());
        assertEquals("a", studentA.getName());
    }
}
```

**Using Statics:
Various Notes**

The last `assertEquals` statement will fail, since both `studentA` and `studentB` share the class variable `name`. In all likelihood, other test methods will fail as well.

A mistake like this can waste a lot of your time, especially if you don't have good unit tests. Developers often figure that something as simple as a variable declaration cannot be broken, so the variable declarations are often the last place they look when there are problems.

Revert the `Student` class to eliminate the keyword `static`, remove `testBadStatic`, recompile, and retest.

Using Statics: Various Notes

- Avoid turning an instance method into a class method for the sole sake of making it static. Ensure that either it semantically makes sense or at least one additional class needs access to the class method before doing so.

Garbage Collection

Java tries its best to manage your application's use of memory as it executes. Memory is a precious, limited commodity in most computer systems. Every time your code creates an object, Java must find memory space in which to store the object. If Java did nothing to manage memory, objects put in memory would stay there forever, and you would very quickly use up all available memory.

Java uses a technique known as *garbage collection* to manage your application's memory use. The Java VM tracks your use of all objects; from time to time it runs something known as a *garbage collector* in the background. The garbage collector reclaims objects that it knows you no longer need.

You no longer need an object when no other objects refer to that object. Suppose you create an object within a method and assign it to a local variable (but not to anything else). When the VM completes execution of the method, the object remains in memory, but nothing points to it—the local variable reference is only valid for the scope of the method. At this point, the object is eligible for garbage collection and should disappear the next time the garbage collector runs (if ever—you can never guarantee that the garbage collector will run).

If an instance variable refers to an object, you can give the object up for potential garbage collection by setting the value of the instance variable to `null`. Or you can wait until the object containing the instance variable is no longer referred to. Once nothing refers to an object, any objects it in turn refers to are also eligible for garbage collection.

If you store an object in a standard collection, such as an `ArrayList`, the collection holds a reference to the object. The object cannot be garbage collected as long the collection contains it.

**Jeff's Rule of
Statics**

- Static-side collections (for example, storing an `ArrayList` object in a class variable) are usually a bad idea. A collection holds a reference to any object added to it. Any object added to a class collection stays there until it is either removed from the collection or until the application terminates. An instance-side collection doesn't have this problem; see the sidebar for a brief overview of how garbage collection works.

Jeff's Rule of Statics

Finally, there is what I immodestly call Jeff's Rule of Statics:



Don't use statics until you know you *need* to use statics.

The simple rule comes about from observing first Java development efforts. A little knowledge goes a long way. A little knowledge about statics often leads developers to use them rampantly.

My philosophical opposition to overuse of statics is that they are not object-oriented. The more static methods you have in your system, the more procedural it is—it becomes a bunch of essentially global functions operating on global data. My practical opposition is that improper and careless use of statics can cause all sorts of problems, including design limitations, insidious and bizarre defects, and memory leaks.

You will learn when it is appropriate to use statics and when it is not. Make sure that you do not use a static unless you understand why you are doing so.

Booleans

 The next small portion of the student information system that you need to build is related to billing students for a semester. For now, the amount that students are billed is based upon three things: whether or not they are in-state students, whether or not they are full-time students, and how many credit hours the students are taking. In order to support billing, you will have to update the Student class to accommodate this information.

Students are either full-time or they are part-time. Put another way, students are either full-time or they are not full-time. Any time you need to represent something in Java that can be only in one of two states—on or off—you can use a variable of the type `boolean`. For a boolean variable, there are two possible boolean values, represented by the literals `true` (on) and `false` (off). The type `boolean` is a primitive type, like `int`; you cannot send messages to `boolean` values or variables.

Create a test method in the `StudentTest` class named `testFullTime`. It should instantiate a `Student`, then test that the student is not full time. Full-time students must have at least twelve credit hours; a newly created student has no credit hours.

```
public void testFullTime() {  
    Student student = new Student("a");  
    assertFalse(student.isFullTime());  
}
```

The `assertFalse` method is another method `StudentTest` inherits from `junit.framework.TestCase`. It takes a single `boolean` expression as an argument. If the expression represents the value `false`, then the test passes; otherwise, the test fails. In `testFullTime`, the test passes if the student is not full-time; that is, if `isFullTime` returns `false`.

Add a method named `isFullTime` to the `Student` class:

```
boolean isFullTime() {
    return true;
}
```

The return type of the method is `boolean`. By having this method return `true`, you should expect that the test fails—the test asserts that `isFullTime` should return `false`. Observe the test fail; modify the method to return `false`; observe the test pass.

The full-time/part-time status of a student is determined by how many credits worth of courses that the student takes. To be considered full-time, a student must have at least a dozen credits. How does a student get credits? By enrolling in a course session.

The requirement now is that when a student is enrolled in a course session, the student's number of credits must be bumped up. Simplest things first: Students need to be able to track credits, and a newly created student has no credits.

```
public void testCredits() {
    Student student = new Student("a");
    assertEquals(0, student.getCredits());
    student.addCredits(3);
    assertEquals(3, student.getCredits());
    student.addCredits(4);
    assertEquals(7, student.getCredits());
}
```

In `Student`:

```
package sis.studentinfo;

public class Student {
    private String name;
    private int credits;

    public Student(String name) {
        this.name = name;
        credits = 0;
    }

    public String getName() {
        return name;
    }

    boolean isFullTime() {
        return false;
    }

    int getCredits() {
        return credits;
    }
}
```

Booleans

```
void addCredits(int credits) {
    this.credits += credits;
}
```

The Student constructor initializes the value of the credits field to 0, to meet the requirement that newly created students have no credits. As you learned in Lesson 2, you could have chosen to use field initialization, or to have not bothered, since Java initializes int variables to 0 by default.

Up to this point, the student should still be considered part-time. Since the number of credits is directly linked to the student's status, perhaps you should combine the test methods (but this is a debatable choice). Instead of having two test methods, `testCredits` and `testFullTime`, combine them into a single method named `testStudentStatus`.

```
public void testStudentStatus() {
    Student student = new Student("a");
    assertEquals(0, student.getCredits());
    assertFalse(student.isFullTime());

    student.addCredits(3);
    assertEquals(3, student.getCredits());
    assertFalse(student.isFullTime());

    student.addCredits(4);
    assertEquals(7, student.getCredits());
    assertFalse(student.isFullTime());
}
```

Booleans

Assertion Failure Messages

You can code any JUnit assertion to provide a specialized error message. JUnit displays this message if the assertion fails. While the test itself should be coded well enough to help another developer understand the implication of an assertion failure, adding a message can help expedite understanding. Here's an example:

```
assertTrue(
    "not enough credits for FT status",
    student.isFullTime());
```

In the case of `assertEquals`, often the default message is sufficient. In any case, try reading the message generated and see if it imparts enough information to another developer.

This test should pass. Now modify the test to enroll the student in a back-breaking five-credit course in order to put them at twelve credits. You can use the `assertTrue` method to test that the student is now full-time. A test passes if the parameter to `assertTrue` is true, otherwise the test fails.

```

public void testStudentStatus() {
    Student student = new Student("a");
    assertEquals(0, student.getCredits());
    assertFalse(student.isFullTime());

    student.addCredits(3);
    assertEquals(3, student.getCredits());
    assertFalse(student.isFullTime());

    student.addCredits(4);
    assertEquals(7, student.getCredits());
    assertFalse(student.isFullTime());

    student.addCredits(5);
    assertEquals(12, student.getCredits());
    assertTrue(student.isFullTime());
}

```

The test fails. To make it pass, you must modify the `isFullTime` method to return `true` if the number of credits is 12 or more. This requires you to write a conditional. A conditional in Java is an expression that returns a *boolean* value. Change the method `isFullTime` in `Student` to include an appropriate expression:

```

boolean isFullTime() {
    return credits >= 12;
}

```

You can read this code as “return `true` if the number of credits is greater than or equal to 12, otherwise return `false`.”

Refactor `isFullTime` to introduce a `Student` class constant to explain what the number 12 means.

```

static final int CREDITS_REQUIRED_FOR_FULL_TIME = 12;
...
boolean isFullTime() {
    return credits >= CREDITS_REQUIRED_FOR_FULL_TIME;
}

```

Now that students support adding credits, you can modify `CourseSession` to ensure that credits are added to `Student` objects as they are enrolled. Start with the test:

```

public class CourseSessionTest extends TestCase {
    // ...
    private static final int CREDITS = 3;

    public void setUp() {
        startDate = createDate(2003, 1, 6);
        session = createCourseSession();
    }
}

```

Booleans

```
// ...
public void testEnrollStudents() {
    Student student1 = new Student("Cain DiVoe");
    session.enroll(student1);
    assertEquals(CREDITS, student1.getCredits());
    assertEquals(1, session.getNumberOfStudents());
    assertEquals(student1, session.get(0));

    Student student2 = new Student("Coralee DeVaughn");
    session.enroll(student2);
    assertEquals(CREDITS, student2.getCredits());
    assertEquals(2, session.getNumberOfStudents());
    assertEquals(student1, session.get(0));
    assertEquals(student2, session.get(1));
}
// ...
private CourseSession createCourseSession() {
    CourseSession session =
        CourseSession.create("ENGL", "101", startDate);
    session.setNumberOfCredits(CourseSessionTest.CREDITS);
    return session;
}
}
```

A few quick changes to CourseSession make this failing test pass:

```
public class CourseSession {
    ...
    private int numberOfCredits;
    ...
    void setNumberOfCredits(int numberOfCredits) {
        this.numberOfCredits = numberOfCredits;
    }

    public void enroll(Student student) {
        student.addCredits(numberOfCredits);
        students.add(student) ;
    }
    ...
}
```

Tests as Documentation

Tests as Documentation

The test method `testStudentStatus` ensures that students report the appropriate full-time or part-time status. It also ensures that the `Student` class correctly adds credits.

What the test does not do is exhaustively test every possibility. The general strategy for testing is to test against 0, 1, many, and any boundary conditions and any exceptional cases. With respect to student credits, the test would en-

sure that a student with 0, 1, or 11 credits reported as part-time and that a student with 12 or 13 credits reported full-time. It would also test the results of unexpected operations, such as adding negative credits, or adding very high numbers of credits.

Test-driven development takes a slightly different approach. The strategy is similar, but the goals are not quite the same. The tests are not just a means of ensuring that the code is correct. In addition, test-driven design provides a technique for consistently paced development. You learn how to incrementally develop code, getting feedback every few seconds or minutes that you are progressing in a valid direction. Tests are about confidence.

Test-driven development also begins to affect the design of the system you are building. This is deliberate: Test-driven development teaches you how to build systems that can be easily tested. Far too few production systems have testability as a characteristic. With test-driven development, you will learn how to test classes in isolation from other classes in the system. This leads to a system where the objects are not tightly coupled to one another, the leading indicator of a well-designed object-oriented system.

Finally, tests document the functionality that your classes provide. When you finish coding, you should be able to review your tests to understand what a class does and how it does it. First, you should be able to view the names of your tests to understand all the functionality that a given class supports. Second, each test should also be readable as documentation on how to use that functionality.



Code tests as comprehensive specifications that others can understand.

Tests as Documentation

In the case of `testStudentStatus`, you as the developer have a high level of confidence that the production code for `isFullTime` is valid. It's only a single line, and you know exactly what that line of code states:

```
return credits >= Student.CREDITS_REQUIRED_FOR_FULL_TIME
```

You might consider the test sufficient and choose to move on, and in doing so you wouldn't be entirely out of line. Again, tests are largely about confidence. The less confident you are and the more complex the code, the more tests you should write.

What about unexpected operations? Won't it destroy the integrity of a `Student` object if someone sends a negative value for number of credits? Remember, you are the developer building this system. You are the one who will control access to the `Student` class. You have two choices: you can test for and guard against every possible exceptional condition, or you can use the design of your system to make some assumptions.

In the student information system, the CourseSession class will be the only place where the Student number of credits can be incremented; this is by design. If CourseSession is coded properly, then it will have stored a reasonable number of credits. Since it will have stored a reasonable number of credits, there is theoretically no way for a negative number to be passed to Student. You know that you have coded CourseSession properly because you did it using TDD!

Of course, somewhere some human will have to enter that number of credits for a course session into the system. It is at that point—at code that represents the user interface level—that you must guard against all possibilities. A human might enter nothing, a letter, a negative number, or a \$ character. The tests for ensuring that a reasonable positive integer is passed into the system will have to be made at this point.

Once you have this sort of barrier against invalid data, you can consider that the rest of the system is under your control—theoretically, of course. With this assumption, you can remove much of the need to guard the rest of your classes against bad data.

In reality, there are always “holes” that you unwittingly use to drop defects in your code. But a large key to success with test-driven development is to understand its heavy emphasis on feedback. A defect is an indication that your unit tests were not complete enough: You missed a test. Go back and write the missing test. Ensure that it fails, then fix it. Over time you will learn what is important to test and what is not. You will learn where you are investing too much time or too little time on the tests.

I have confidence in testStudentStatus and the corresponding implementation. Where the test is lacking is in its ability to act as documentation. Part of the problem is that you as a developer have too much knowledge about how you have coded the functionality. It helps to find another developer to read the test to see if it describes all of the business rules and boundaries properly. If another developer is unavailable, take a step back and try to look at the test as if you had never seen the underlying code. Does it tell you how to use the class? Does it demonstrate the different scenarios? Does it indicate the constraints or limitations of the code being tested—perhaps by omission?

In this case, perhaps all that is needed is to let the test use the same constant that Student uses. The test becomes a good deal more expressive by virtue of doing so. The boundaries of full-time are now implicit and reasonably well understood.

```
public void testStudentStatus() {  
    Student student = new Student("a");  
    assertEquals(0, student.getCredits());  
    assertFalse(student.isFullTime());
```

Tests as Documentation

```

student.addCredits(3);
assertEquals(3, student.getCredits());
assertFalse(student.isFullTime());

student.addCredits(4);
assertEquals(7, student.getCredits());
assertFalse(student.isFullTime());

student.addCredits(5);
assertEquals(Student.CREDITS_REQUIRED_FOR_FULL_TIME,
    student.getCredits());
assertTrue(student.isFullTime());
}

```

More on Initialization

 To support the notion of in-state versus out-of-state students, the Student object needs to store a state that represents where the Student resides. The school happens to be located in the state of Colorado (abbreviation: CO). If the student resides in any other state, or if the student has no state specified (either the student is international or didn't complete the forms yet), then the student is an out-of-state student.

Here is the test:

```

public void testInState() {
    Student student = new Student("a");
    assertFalse(student isInState());
    student.setState(Student.IN_STATE);
    assertTrue(student isInState());
    student.setState("MD");
    assertFalse(student isInState());
}

```

More on
Initialization

The logic for determining whether a student is in-state will have to compare the String representing the student's state to the String "CO". This of course means you will need to create a field named state.

```

boolean isInState() {
    return state.equals(Student.IN_STATE);
}

```

To compare two strings, you use the `equals` method. You send the `equals` message to a `String` object, passing another `String` as an argument. The `equals` method will return `true` if both strings have the same length and if each string matches character for character. Thus `"CO".equals("CO")` would return `true`; `"Aa".equals("AA")` would return `false`.

In order for `testInState` to pass, it is important that the state field you create has an appropriate initial value. Anything but "CO" will work, but the empty String ("") will do just fine.

```
package sis.studentinfo;

public class Student {
    static final String IN_STATE = "CO";
    ...
    private String state = "";
    ...
    void setState(String state) {
        this.state = state;
    }
    boolean isInState() {
        return state.equals(Student.IN_STATE);
    }
}
```

You might also consider writing a test to demonstrate what should happen if someone passes a lowercase state abbreviation. Right now, if client code passes in "Co", it will not set the student's status to in-state, since "Co" is not the same as "CO". While that may be acceptable behavior, a better solution would be to always translate a state abbreviation to its uppercase equivalent prior to comparing against "CO". You could accomplish this by using the `String` method `toUpperCase`.

Exceptions

Exceptions

What if you do not supply an initial value for the state field? In the test, you create a new `Student` object and immediately send it the message `isInState`. The `isInState` message results in the `equals` message being sent to the state field. Find out what happens if you send a message to an uninitialized object. Change the declaration of the state field to comment out the initialization:

```
private String state; // = "";
```

Then rerun the tests. JUnit will report an error, not a test failure. An error occurs when neither your production code nor your test accounts for a problem. In this case, the problem is that you are sending a message to an uninitialized field reference. The second panel in JUnit shows the problem.

```
java.lang.NullPointerException
at studentinfo.Student.isInState(Student.java:37)
at studentinfo.StudentTest.testInState(StudentTest.java:39)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
...
...
```

This is known as a *stack walkback*, or *stack trace*. It provides you with the information on what went wrong, but figuring out the stack trace can take a little bit of detective work. The first line in a stack trace tells you what the problem is. In this case, you got something known as a `NullPointerException`. A `NullPointerException` is actually an error object, or *exception*, that is “thrown” by some underlying, problematic code.

The rest of the lines in a stack trace “walk back” through the message sends leading up to the error. Some of the lines refer to classes and methods that you have coded; other lines refer to Java system library code and third-party library code. The easiest way to decipher stack traces is to read down and find the first line of code that you recognize as being “your” code. Then keep reading lines up to the last line that you recognize. This last recognized line is the entry point into your code; you will probably want to dig down from there.

In the above example, the last line to execute was line 37 in `Student.java` (the line numbers in your code will likely differ). That code was invoked by the message send in `StudentTest` line 39. So start with `StudentTest` line 39 in order to follow the trail leading up to the trouble. That should take you to the following line of code:

```
assertFalse(student.isInState());
```

Taking a look at line 37 in `Student.java` reveals the line of your code that generated the `NullPointerException`:

```
return state.equals(Student.IN_STATE);
```

A reference that you do not explicitly initialize has a value of `null`. The value `null` represents the unique instance of an object known as the null object. If you send a message to the null object, you receive a `NullPointerException`. In this line of code, you sent the message `equals` to the uninitialized `state` reference, hence the `NullPointerException`.

In Lesson 6, you will see how to determine if a field is `null` before attempting to send a message to it. For now, ensure that you initialize `String` fields to the empty String ("").

Revert your code so that the `state` field is initialized properly. Rerun your tests.

**Revisiting
Primitive-Type
Field
Initialization**

Revisiting Primitive-Type Field Initialization

Only reference-type fields—fields that can point to objects in memory—can be initialized to `null` or have a value of `null`. You cannot initialize a variable of the primitive type to `null`, nor can such a variable ever have a value of `null`.

This includes boolean variables and variables of the numeric type (char and int, plus the rest that you'll learn about in Lesson 10: byte, short, long, float, and double). Fields of the boolean type have an initial value of false. Each of the numeric types has an initial value of 0.

Even though 0 is often a useful initial value for a numeric type, you should explicitly initialize the field to 0 if that represents a meaningful value for the field. For example, explicitly initialize a field to 0 if it represents a counter that will track values starting at 0 when the Java VM instantiates the object. If you are going to explicitly assign a more meaningful value to the field in later execution of the code, then you need not explicitly initialize the field to 0.

Provide explicit initializations only when necessary. This will help clarify what you, the developer of the code, intend.

Exercises

1. Concatenating a new line character to the end of a string has created repetitive code. Extract this functionality to a new utility method on the class util.StringUtil. Mark StringUtil as a utility class by changing its constructor's access privilege to private. You will need to move the class constant NEWLINE to this class. Use your compiler to help you determine impacts on other code. Ensure that you have tests for the utility method.
2. Transform your Pawn class into a more generic class named Piece. A Piece is a color plus a name (pawn, knight, rook, bishop, queen, or king). A Piece should be a *value object*: It should have a private constructor and no way to change anything on the object after construction. Create factory methods to return Piece objects based on the color and name. Eliminate the ability to create a default piece.
3. Change BoardTest to reflect the complete board:

```
package chess;

import junit.framework.TestCase;
import util.StringUtil;

public class BoardTest extends TestCase {
    private Board board;

    protected void setUp() {
        board = new Board();
    }
}
```

```
public void testCreate() {  
    board.initialize();  
    assertEquals(32, board.pieceCount());  
    String blankRank = StringUtil.appendNewLine(".....");  
    assertEquals(  
        StringUtil.appendNewLine("RNBQKBNR") +  
        StringUtil.appendNewLine("PPPPPPP") +  
        blankRank + blankRank + blankRank + blankRank +  
        StringUtil.appendNewLine("ppppppp") +  
        StringUtil.appendNewLine("rn b q k b n r"),  
        board.print());  
}
```

4. Ensure that a new Board creates sixteen black pieces and sixteen white pieces. Use a class counter on the Piece object to track the counts. Make sure you can run your tests twice without problems (unchecked the “Reload Classes Every Run” box on JUnit and click Run a second time).
5. Create methods `isBlack` and `isWhite` on the `Piece` class (test first, of course).
6. Gather the names of each of your test methods. Put the class name in front of each test name. Show the list to a friend and ask what the methods say about each class.
7. Read over the section on Simple Design again. Does your current chess design follow the patterns of simple design?

Exercises

This page intentionally left blank

Lesson 5

Interfaces and Polymorphism

In this lesson you will learn about:

- sorting
- interfaces
- the Comparable interface
- if statements
- enumerated types using enum
- polymorphism

Sorting: Preparation



The school needs a report of all course sessions. You must sort the report first by department, then by course number. This implies that all courses for a given department will be listed together. The groups of departments will be ordered in ascending alphabetical order. Within a department, the sessions will be ordered by course number.

Sorting:
Preparation

To get started, get a simple report working of all course sessions. Don't worry yet about its order.

```
package sis.report;

import junit.framework.*;
import java.util.*;
import sis.studentinfo.*;
import static sis.report.ReportConstant.NEWLINE;

public class CourseReportTest extends TestCase {
    public void testReport() {
        final Date date = new Date();
```

```

CourseReport report = new CourseReport();
report.add(CourseSession.create("ENGL", "101", date));
report.add(CourseSession.create("CZEC", "200", date));
report.add(CourseSession.create("ITAL", "410", date));

assertEquals(
    "ENGL 101" + NEWLINE +
    "CZEC 200" + NEWLINE +
    "ITAL 410" + NEWLINE,
    report.text());
}
}

```

The report is bare boned, showing simply a course department and number on each line. Currently the report lists sessions in the order in which you added them to the CourseReport object.

The production class, CourseReport, looks similar to RosterReporter:

```

package sis.report;

import java.util.*;
import sis.studentinfo.*;
import static sis.report.ReportConstant.NEWLINE;

public class CourseReport {
    private ArrayList<CourseSession> sessions =
        new ArrayList<CourseSession>();

    public void add(CourseSession session) {
        sessions.add(session);
    }

    public String text() {
        StringBuilder builder = new StringBuilder();
        for (CourseSession session: sessions)
            builder.append(
                session.getDepartment() + " " +
                session.getNumber() + NEWLINE);
        return builder.toString();
    }
}

```

**Sorting:
Collections.sort**

In order to get CourseReport to compile, you'll have to designate the CourseSession methods `getDepartment` and `getNumber` as `public`.

Sorting: Collections.sort

You can sort a list of String objects quite simply, as this language test demonstrates:

```
public void testSortStringsInPlace() {
    ArrayList<String> list = new ArrayList<String>();
    list.add("Heller");
    list.add("Kafka");
    list.add("Camus");
    list.add("Boyle");
    java.util.Collections.sort(list);
    assertEquals("Boyle", list.get(0));
    assertEquals("Camus", list.get(1));
    assertEquals("Heller", list.get(2));
    assertEquals("Kafka", list.get(3));
}
```

The static `sort` method in the `java.util.Collections` class takes a list as a parameter and sorts the list *in place*.¹ If you do not want to sort the list in place—if you do not want to modify the original list—you can create a new list and send that list as a parameter to the `sort` message.

```
public void testSortStringsInNewList() {
    ArrayList<String> list = new ArrayList<String>();
    list.add("Heller");
    list.add("Kafka");
    list.add("Camus");
    list.add("Boyle");
    ArrayList<String> sortedList = new ArrayList<String>(list);
    java.util.Collections.sort(sortedList);
    assertEquals("Boyle", sortedList.get(0));
    assertEquals("Camus", sortedList.get(1));
    assertEquals("Heller", sortedList.get(2));
    assertEquals("Kafka", sortedList.get(3));

    assertEquals("Heller", list.get(0));
    assertEquals("Kafka", list.get(1));
    assertEquals("Camus", list.get(2));
    assertEquals("Boyle", list.get(3));
}
```

CourseReport
Test

The last four assertions verify that the original list remains unmodified.

CourseReportTest

In `CourseReportTest`, modify the assertion in `testReport` to ensure that you produce the report in sorted order. Your test data currently takes only the department into account. That's fine for now:

¹The `sort` method uses a merge sort algorithm that splits the list of elements into two groups, recursively sorts each group, and merges all sorted subgroups into a complete list.

```
assertEquals(
    "CZEC 200" + NEWLINE +
    "ENGL 101" + NEWLINE +
    "ITAL 410" + NEWLINE,
    report.text());
```

The test fails as expected. You should be able to see in JUnit that the sessions are in the wrong order.

To solve the problem, use `Collections.sort` to order the sessions list.

```
public String text() {
    Collections.sort(sessions);
    StringBuilder builder = new StringBuilder();
    for (CourseSession session: sessions)
        builder.append(
            session.getDepartment() + " " +
            session.getNumber() + NEWLINE);
    return builder.toString();
}
```

Unfortunately, this will not compile:

```
cannot find symbol
symbol : method sort(java.util.List<sis.studentinfo.CourseSession>)
location: class java.util.Collections
    Collections.sort(sessions);
           ^
```

The source of the error can be difficult to decipher, even if you are relatively experienced in Java. The problem is that the method declaration for `sort` requires the objects it sorts to be of the type `java.lang.Comparable`. (How it does that involves advanced syntax that I will discuss in Lesson 14 on generics.)

`Comparable` is a type that allows you to compare objects to one another. But you want to compare `CourseSession` objects, since that's what you bound the variable `sessions` to. The secret to getting this to work takes advantage of a feature in Java known as *interfaces*. Interfaces allow an object to act as more than one type. You will modify `CourseSession` to act as a `java.lang.Comparable` type in addition to being a `CourseSession`.

Interfaces

Interfaces

`Comparable` is an *interface* type, not a class type. An interface contains any number of method *declarations*. A method declaration is the *signature* for a method followed by a semicolon. There are no braces and there is no method code. Java defines the type `Comparable` as follows:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

The letter T is a placeholder for the type of objects to be compared. Lesson 14 on generics will explain this concept in further detail. For now, understand that you declare a Comparable reference along with a type. The Java compiler substitutes this type for all occurrences of T in the interface declaration.

Contrast an interface declaration to a class declaration. You declare an interface using the `interface` keyword, whereas you declare a class using the `class` keyword. An interface can contain no implementations for any of the methods that it declares.

In UML, you can represent an interface in a number of ways. Figure 5.1 shows one such way.

The goal of the `compareTo` method is to return a value that indicates whether an object should come before or after another object with respect to ordering. All sort techniques involve comparing only two objects at a time and swapping those two objects if necessary.

For example, a list contains two String objects, "E" and "D", in that order. You want the list sorted in alphabetical order. The Collections method `sort` would send the `compareTo` message to the String "E", along with the parameter "D". The `compareTo` method would return a value that indicates that "D" should appear first in the sort order, before "E". The `sort` method would then know that it would have to swap the two objects for them to be sorted correctly.

A class can declare that it *implements* an interface. By making this declaration, a class promises to provide implementations for each method declared in the interface. Implementing an interface allows classes to act as more than one type.

The earlier sorting example, the one that actually worked, sorted a collection of String objects. The J2SE API documentation for the String class lists the Comparable interface as one of three implemented interfaces. Implementing the Comparable interface allows String objects to be treated not just as objects of String type but also as objects of Comparable type.

Interfaces

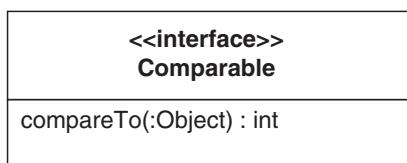


Figure 5.1 Comparable

The syntax for implementing interfaces is demonstrated in this bit of code from the actual Java String class:

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence
{
    ...
}
```

If you look further into the source for the String class, you will find the implementation for the `compareTo` method that Comparable declares.

For the sort invoked by CourseReport to work, it must be able to send the message `compareTo` to the objects contained within sessions. The sort method is unaware of the class of objects in the collection, however. It takes each object in turn and tries to assign the object to a variable of the Comparable interface type. When this assignment is successful, the sort can safely send the `compareTo` message to the object. When the assignment is not successful, Java generates an error.

Why Interfaces

Interfaces are a very powerful and very important feature in Java. Part of the key to solid design in Java is knowing when (and when not) to use interfaces. Used properly, interfaces help you compartmentalize your software to help minimize impacts on other parts of the code.

Interfaces provide higher levels of abstraction. The sort code doesn't need to know any details about the objects it is sorting. It need not know whether it is sorting Student objects, Customer objects, or Strings. All the sort code needs to know is that the objects it is sorting support being compared to other objects. The sort code knows only about this more abstract quality, not any other specifics of the objects it is sorting.

You can view this concept of abstracting sortability as a way to eliminate duplication. Interfaces allow a sort algorithm to operate on objects of different base types. The sort algorithm needs no repetitive code to determine the types of objects it compares.



Use interfaces to provide abstraction layers in your system and to help remove duplication.

The abstraction layers in your system isolate your code from negative effects. Ideally, you write the sort class once, get it to work perfectly, and then close it off to any future changes. The only detail the sort class has to know about the objects it sorts is that they respond to the `compareTo` method by

returning an `int`. The classes of those objects can change in other myriad ways, but none of these changes will ever impact your sort code.

You can use interfaces to break dependencies on code that doesn't work or doesn't even exist by writing stub code to meet the specifications of the interface. In Lesson 12, you will learn how to accomplish this using a technique known as mocking. Interfaces are an essential tool for effective testing.

By implementing the `Comparable` interface, the `String` class publicizes the fact that `Strings` can be compared for sort purposes. The `String` supplies code in the `compareTo` method to determine how `String` objects are compared.

In order to get the sort to work for the list of sessions, you must modify `CourseSession` to implement the `Comparable` interface.

Implementing Comparable

The job of the `compareTo` method in the `Comparable` interface is to indicate which of two objects—the message receiver and the message parameter—should appear first. The code you will supply in `compareTo` should result in what is known as the *natural*, or default, sort order for `CourseSessions`. In other words, how do you normally want to sort a collection of `CourseSession` objects?

The `compareTo` method must return an `int` value. If this return value is `0`, then the two objects are equal for purposes of sorting. If the return value is negative, then the receiver (the object to which code sends the `compareTo` message) should come before the parameter in the sort order. If the value is positive, then the parameter should come before the receiver in the sort order.

The `String` class implements the `compareTo` method in order to sort strings in alphabetical order. The following language test demonstrates the values returned from the `compareTo` method in the three possible scenarios.

```
public void testStringCompareTo() {
    assertTrue("A".compareTo("B") < 0);
    assertEquals(0, "A".compareTo("A"));
    assertTrue("B".compareTo("A") > 0);
}
```

Implementing Comparable

You will need to modify the `CourseSession` class to implement the `Comparable` interface and supply a definition for `compareTo`. An appropriate test to add to `CourseSessionTest`:

```
public void testComparable() {
    final Date date = new Date();
    CourseSession sessionA = CourseSession.create("CMSC", "101", date);
```

```

CourseSession sessionB = CourseSession.create("ENGL", "101", date);
assertTrue(sessionA.compareTo(sessionB) < 0);
assertTrue(sessionB.compareTo(sessionA) > 0);

CourseSession sessionC = CourseSession.create("CMSC", "101", date);
assertEquals(0, sessionA.compareTo(sessionC));
}

```

CourseSession must then declare that it implements the Comparable interface. It must also specify a bind type of CourseSession—you want to be able to compare CourseSession objects to CourseSession objects.

```

public class CourseSession implements Comparable<CourseSession> {
    ...

```

More on this

In the Student compareTo method, the return statement includes the expression `this.getDepartment()`. As you learned in Lesson 1, the `this` reference is to the current object. Scoping method calls with `this` is usually unnecessary, but in this case it helps differentiate the current object and the parameter object.

Another use of the `this` keyword that you haven't seen is for *constructor chaining*. You can invoke another constructor defined on the same class by using a slightly different form of this:

```

class Name {
    ...
    public Name(String first, String mid, String last) {
        this.first = first;
        this.mid = mid;
        this.last = last;
    }

    public Name(String first, String last) {
        this(first, "", last);
    }
    ...
}

```

The single statement in the second constructor of Name calls the first constructor. The goal in this example is to pass a default value of the empty string as the middle name.

Such a call to another constructor must appear as the first line of a constructor.

Constructor chaining can be a valuable tool for helping you eliminate duplication. Without chaining, you would need to write a separate method to do common initialization.

Since you bound the implementation of Comparable to the CourseSession, you must define the `compareTo` method so that it takes a CourseSession as parameter:

```
public int compareTo(CourseSession that) {  
    return this.getDepartment().compareTo(that.getDepartment());  
}
```

For the `compareTo` method implementation, you are returning the result of comparing the current `CourseSession`'s department (`this.getDepartment()`) to the parameter `CourseSession` object's department (`that.getDepartment()`).

At this point, both `testComparable` in `CourseSessionTest` and `testReport` in `CouseReportTest` should pass.

Sorting on Department and Number

You are now able to produce a report that sorts course sessions by their department. You will now enhance the report code to sort sessions by both department and number. You will need to first modify the test to incorporate appropriate data. The following test (in CourseReportTest) adds two new sessions. The test data now includes two sessions that share the same department.

Sorting on Department and Number

```

        "ITAL 410" + NEWLINE,
        report.text());
    }
}

```

In order to determine the proper ordering of these students, your `compareTo` method will need to first look at department. If the departments are different, that comparison will be sufficient to provide a return value. If the departments are the same, you will have to compare the course numbers.

The if Statement

You use the `if` statement for conditional branching: If a condition holds true, execute one branch (section) of code, otherwise execute another branch of code. A branch can be a single statement or a block of statements.

Flesh out the `CourseSessionTest` method `testComparable` by adding tests to represent more complex comparisons:

```

public void testComparable() {
    final Date date = new Date();
    CourseSession sessionA = CourseSession.create("CMSC", "101", date);
    CourseSession sessionB = CourseSession.create("ENGL", "101", date);
    assertTrue(sessionA.compareTo(sessionB) < 0);
    assertTrue(sessionB.compareTo(sessionA) > 0);

    CourseSession sessionC = CourseSession.create("CMSC", "101", date);
    assertEquals(0, sessionA.compareTo(sessionC));

    CourseSession sessionD = CourseSession.create("CMSC", "210", date);
    assertTrue(sessionC.compareTo(sessionD) < 0);
    assertTrue(sessionD.compareTo(sessionC) > 0);
}

```

The if Statement

Then update the `compareTo` method in `CourseSession`:

```

public int compareTo(CourseSession that) {
    int compare =
        this.getDepartment().compareTo(that.getDepartment());
    if (compare == 0)
        compare = this.getNumber().compareTo(that.getNumber());
    return compare;
}

```

Paraphrased, this code says: Compare this department to the department of the parameter; store the result of the comparison in the `compare` local variable. *If* the value stored in `compare` is 0 (if the departments are the same), com-

pare this course number to the parameter's course number and assign the result to `compare`. Regardless, return the value stored in the `compare` local variable.

You could have also written the code as follows:

```
public int compareTo(CourseSession that) {
    int compare =
        this.getDepartment().compareTo(that.getDepartment());
    if (compare != 0)
        return compare;
    return this.getNumber().compareTo(that.getNumber());
}
```

If, on the first comparison, the departments are unequal (the result of the comparison is not 0), then that's all that needs to happen. You can stop there and return the result of the comparison. The rest of the code is not executed. This style of coding can be easier to follow. Be cautious, however: In longer methods, multiple returns can make the method more difficult to understand and follow. For longer methods, then, I do not recommend multiple return statements. But the real solution is to have as few long methods as possible.

All tests should pass.

Grading Students

 You need to be able to produce a report card for all students. Before you can do that, you will need the capability to calculate the GPA (grade point average) for a given student. A student receives a number of grades. You store a student's grade by sending the message `addGrade` to a `Student` object, passing along the letter grade. When asked for the GPA, the student uses its stored grades to calculate a GPA.

You will supply grades as decimal numbers, known in Java as floating-point numbers.

Floating-Point
Numbers

Floating-Point Numbers

You are already familiar with the primitive-type `int`, representing integer values. In addition to integer numerics, Java supports IEEE 754 32-bit single precision binary floating point format numbers, also known as float points, or floats. Java also supports 64-bit double precision floats, also known as doubles.

Table 5.1 Floating-Point Precision Types

Type	Range	Example Literals
float	1.40239846e -45f to 3.40282347e + 38f	6.07f 6F 7.0f 7.0f 1.8e5f
double	4.94065645841246544e -324 to 1.79769313486231570e +308	1440.0 6 3.2545e7 32D 0d

Table 5.1 lists the types, the range of values that they support and some examples.

Any numeric literal with a decimal point or any numeric literal with a suffix of f, F, d, or D, is by definition a floating-point number.

Float and double literals may also use scientific notation. Any number with an e or E in the middle is by definition a floating-point number. The double numeric literal 1.2e6 represents 1.2 times 10 to the 6th power, or 1.2×10^6 in standard mathematical notation.

If you do not supply a suffix for a floating-point number, it is by default a double precision float. In other words, you must supply an F or f suffix to produce a single precision float literal. In Agile Java, I will prefer the use of double over float due to its higher precision and the fact that I need not supply a suffix.

Realize, though, that floating-point numbers are approximations of real numbers based on bit patterns. It is not possible to represent all real numbers, since there are an infinite amount of real numbers and only a finite set of possible representations in either 32- or 64-bit precision floats. This means that most real numbers have only an *approximate* floating-point representation.

For example, if you execute the following line of code:

```
System.out.println("value = " + (3 * 0.3));
```

the output will be:

```
value = 0.8999999999999999
```

and not the expected value 0.9.

If you use floating-point numbers, you must be prepared to deal with these inaccuracies by using rounding techniques. Another solution is to use the Java class BigDecimal to represent numbers with fractional parts. Agile Java presents BigDecimal in Lesson 10.

Testing Grades

The following test, which you should add to StudentTest, demonstrates a range of possibilities. It starts with the simplest case: The student has received no grades, in which case the student's GPA should be 0. It also tests grade combinations based on applying each of the possible letter grades from A through F.

```
private static final double GRADE_TOLERANCE = 0.05;
...
public void testCalculateGpa() {
    Student student = new Student("a");
    assertEquals(0.0, student.getGpa(), GRADE_TOLERANCE);
    student.addGrade("A");
    assertEquals(4.0, student.getGpa(), GRADE_TOLERANCE);
    student.addGrade("B");
    assertEquals(3.5, student.getGpa(), GRADE_TOLERANCE);
    student.addGrade("C");
    assertEquals(3.0, student.getGpa(), GRADE_TOLERANCE);
    student.addGrade("D");
    assertEquals(2.5, student.getGpa(), GRADE_TOLERANCE);
    student.addGrade("F");
    assertEquals(2.0, student.getGpa(), GRADE_TOLERANCE);
}
```

The `assertEquals` method calls in this test show three parameters instead of two. Since floating-point numbers are not precise representations of real numbers, there is a possibility that a calculated value may be off from an expected value by a certain amount. For example, were it possible to code:

```
assertEquals(0.9, 3 * 0.3);
```

the test would fail:

```
AssertionFailedError: expected:<0.9> but was:<0.8999999999999999>
```

Testing Grades

JUnit provides a third parameter when you need to compare two floating-point values for equality. This parameter represents a *tolerance*: How much can the two floating-point values be off by before JUnit reports an error?

A general rule of thumb is that values should be off by no more than half of the smallest precision you are interested in representing accurately. For example, if you are working with cents (hundredths of a dollar), then you want to ensure that compared amounts are off by no more than half a cent.

You want to accurately express GPAs to tenths of a point; therefore your tolerance should be 5/100 of a point. The above code defines a static constant `GRADE_TOLERANCE` local to the `StudentTest` class and uses it as the third parameter to each `assertEquals` method:

```
assertEquals(2.0, student.getGpa(), GRADE_TOLERANCE);
```

In order to make the test pass, you will need to change the Student class to store each added grade in an ArrayList.² You can then code the `getGpa` method to calculate the result GPA. To do so, you must first iterate through the list of grades and obtain a grade point total. You then divide this grade point total by the total number of grades to get the GPA.

```
import java.util.*;
...
class Student {
    private ArrayList<String> grades = new ArrayList<String>();
    ...
    void addGrade(String grade) {
        grades.add(grade);
    }
    ...
    double getGpa() {
        if (grades.isEmpty())
            return 0.0;
        double total = 0.0;
        for (String grade: grades) {
            if (grade.equals("A")) {
                total += 4;
            }
            else {
                if (grade.equals("B")) {
                    total += 3;
                }
                else
                {
                    if (grade.equals("C")) {
                        total += 2;
                    }
                    else {
                        if (grade.equals("D")) {
                            total += 1;
                        }
                    }
                }
            }
        }
        return total / grades.size();
    }
}
```

Testing Grades

The method `getGpa` starts with an `if` statement to determine whether or not the list of grades is empty. If there are no grades, the method immediately returns a GPA of `0.0`. An `if` statement that appears at the beginning of a method

²You could also calculate the GPA as each grade is added.

and returns upon a special condition is known as a *guard clause*. The guard clause in `getGpa` guards the remainder of the method from the special case where there are no grades. Since you calculate a GPA by dividing by the number of grades, the guard clause eliminates any possibility of dividing by zero.

The method code then uses a for-each loop to extract each grade from the `grades` collection. The body of the for-each loop compares each grade against the possible letter grades. The body code uses an extension of the if statement known as the if-else statement.

A paraphrasing of the body of the for-each loop: *If* the grade is an A, add 4 to the total,³ *otherwise*, execute the block of code (or single statement) appearing after the `else` keyword. In this case, the block of code is itself comprised of another if-else statement: If the grade is a B, add 3 to the total, otherwise execute yet another block of code. This pattern continues until all possibilities are exhausted (an F is ignored, since it adds nothing to the grade total).

Complex if-else statements can be difficult to read because of the many curly braces and repeated indenting. In the circumstance where you have repetitive *nested* if-else statements, you can use a tighter formatting style. Each else-if statement combination goes on the same line. Also, you can eliminate the braces in this example, since the body of each if statement is a single line.

```
double getGpa() {
    if (grades.isEmpty())
        return 0.0;
    double total = 0.0;
    for (String grade: grades) {
        if (grade.equals("A"))
            total += 4;
        else if (grade.equals("B"))
            total += 3;
        else if (grade.equals("C"))
            total += 2;
        else if (grade.equals("D"))
            total += 1;
    }
    return total / grades.size();
}
```

Testing Grades

This alternative formatting style makes the `getGpa` method far easier to read but produces the same results.

³You may add `int` values to doubles with no ill effects. However, mixing `int` and `double` values in an expression can cause unintended effects. See Lesson 10 for a discussion of numerics in Java.

Refactoring

The method `getGpa` is still a bit long and difficult to follow. Extract from it a new method that returns a double value for a given letter grade:

```
double getGpa() {
    if (grades.isEmpty())
        return 0.0;
    double total = 0.0;
    for (String grade: grades)
        total += gradePointsFor(grade);
    return total / grades.size();
}

int gradePointsFor(String grade) {
    if (grade.equals("A"))
        return 4;
    else if (grade.equals("B"))
        return 3;
    else if (grade.equals("C"))
        return 2;
    else if (grade.equals("D"))
        return 1;
    return 0;
}
```

Make sure this compiles and passes the tests. One further change that you can make is to remove the `else` clauses. A `return` statement is a flow of control. Once the Java VM encounters a `return` statement in a method, it executes no more statements within the method.

Refactoring

```
int gradePointsFor(String grade) {
    if (grade.equals("A")) return 4;
    if (grade.equals("B")) return 3;
    if (grade.equals("C")) return 2;
    if (grade.equals("D")) return 1;
    return 0;
}
```

In the `gradePointsFor` method, as soon as there is a match on `grade`, the VM executes a `return` statement and makes no more comparisons.

You'll also note that I put each `return` on the same line as the corresponding `if` statement conditional. Normally, you will want to keep the conditional and the body of an `if` statement on separate lines. In this example, the multiple `if` statements are repetitive, terse, and visually consistent. The code is easy to read and digest. In other circumstances, jumbling the conditional and body together can result in an unreadable mess.

You can refactor the test itself:

```

public void testCalculateGpa() {
    Student student = new Student("a");
    assertGpa(student, 0.0);
    student.addGrade("A");
    assertGpa(student, 4.0);
    student.addGrade("B");
    assertGpa(student, 3.5);
    student.addGrade("C");
    assertGpa(student, 3.0);
    student.addGrade("D");
    assertGpa(student, 2.5);
    student.addGrade("F");
    assertGpa(student, 2.0) ;
}

private void assertGpa(Student student, double expectedGpa) {
    assertEquals(expectedGpa, student.getGpa(), GRADE_TOLERANCE);
}

```

Enums

J2SE 5.0 introduces the notion of an enumerated type—a type that constrains all possible values to a discrete list. For example, if you implement a class that represents a deck of cards, you know that the only possible suit values are clubs, diamonds, spades, and hearts. There are no other suits. For grades, you know there are only five possible letter grades.

You can define grades as strings, as in the above GPA code, and use String values to represent possible letter grades. While this will work, it has some problems. First, it is easy to make a mistake when typing all the various strings. You can create class constants to represent each letter grade (which we probably should have done in the above code), and as long as all code uses the constants, things are fine.

Enums

Even if you have supplied class constants, client code can still pass an invalid value into your code. There is nothing that prevents a user of the Student class from executing this line of code:

```
student.addGrade("a");
```

The results are probably not what you expect—you wrote the code to handle only uppercase grade letters.

You can instead define an enum named Grade. The best place for this, for now, is within the Student class:

```

public class Student {
    enum Grade { A, B, C, D, F };
    ...
}
```

Using `enum` results in the declaration of a new type. In this case, you have just created a new type named `Student.Grade`, since you defined the `Grade` enum within the `Student` class. The new enum has five possible values, each representing a named object instance.

Change the test to use the `enum` instances:

```
public void testCalculateGpa() {
    Student student = new Student("a");
    assertGpa(student, 0.0);
    student.addGrade(Student.Grade.A);
    assertGpa(student, 4.0);
    student.addGrade(Student.Grade.B);
    assertGpa(student, 3.5);
    student.addGrade(Student.Grade.C);
    assertGpa(student, 3.0);
    student.addGrade(Student.Grade.D);
    assertGpa(student, 2.5);
    student.addGrade(Student.Grade.F);
    assertGpa(student, 2.0);
}
```

In the `Student` class, you will need to change any code that declared a grade to be of the type `String` to now be of the type `Grade`. Since the `Grade` enum is defined within `Student`, you can refer to it directly (i.e., as `Grade` instead of `Student.Grade`).

```
public class Student implements Comparable<Student> {
    enum Grade { A, B, C, D, F };
    ...
    private ArrayList<Grade> grades = new ArrayList<Grade>();
    ...

    void addGrade(Grade grade) {
        grades.add(grade);
    }
    ...

    double getGpa() {
        if (grades.isEmpty())
            return 0.0;
        double total = 0.0;
        for (Grade grade: grades)
            total += gradePointsFor(grade);
        return total / grades.size();
    }
    ...

    int gradePointsFor(Grade grade) {
        if (grade == Grade.A) return 4;
        if (grade == Grade.B) return 3;
        if (grade == Grade.C) return 2;
        if (grade == Grade.D) return 1;
    }
}
```

Enums

```

        return 0;
    }
    ...
}

```

The method `gradePointsFor` compares the value of the parameter `grade` to each of the `enum` values in turn. Each of the `enum` values represents a unique instance in memory. This allows you to make the comparison using the `==` operator instead of the `equals` method. Remember that the `==` operator is used for comparing two object references: Do the two references point to the same object in memory?

It is no longer possible for a client to pass in an invalid value to the method `addGrade`. Client code cannot create a new instance of `Grade`. The five instances specified in the `enum` declaration within `Student` are all that exist.

Polymorphism

Overuse of the `if` statement can quickly turn your code into a high-maintenance legacy that is difficult to follow. A concept known as polymorphism can help structure your code to minimize the need for `if` statements.



Student grading is a bit more involved than the above example. You need to support grading for honors students. Honors students are graded on a higher scale. They can earn five grade points for an A, 4 points for a B, 3 points for a C, and 2 points for a D.

A highly refactored test:

```

public void testCalculateHonorsStudentGpa() {
    assertGpa(createHonorsStudent(), 0.0);
    assertGpa(createHonorsStudent(Student.Grade.A), 5.0);
    assertGpa(createHonorsStudent(Student.Grade.B), 4.0);
    assertGpa(createHonorsStudent(Student.Grade.C), 3.0);
    assertGpa(createHonorsStudent(Student.Grade.D), 2.0);
    assertGpa(createHonorsStudent(Student.Grade.F), 0.0);
}

private Student createHonorsStudent(Student.Grade grade) {
    Student student = createHonorsStudent();
    student.addGrade(grade);
    return student;
}

private Student createHonorsStudent() {
    Student student = new Student("a");
    student.setHonors();
    return student;
}

```

Polymorphism

The code to resolve this is trivial.

```
private boolean isHonors = false;
// ...
void setHonors() {
    isHonors = true;
}
// ...
int gradePointsFor(Grade grade) {
    int points = basicGradePointsFor(grade);
    if (isHonors)
        if (points > 0)
            points += 1;
    return points;
}

private int basicGradePointsFor(Grade grade) {
    if (grade == Grade.A) return 4;
    if (grade == Grade.B) return 3;
    if (grade == Grade.C) return 2;
    if (grade == Grade.D) return 1;
    return 0;
}
```

... but things are becoming unwieldy. Next, suppose you must support a different grading scheme:

```
double gradePointsFor(Grade grade) {
    if (isSenatorsSon) {
        if (grade == Grade.A) return 4;
        if (grade == Grade.B) return 4;
        if (grade == Grade.C) return 4;
        if (grade == Grade.D) return 4;
        return 3;
    }
    else {
        double points = basicGradePointsFor(grade);
        if (isHonors)
            if (points > 0)
                points += 1;
        return points;
    }
}
```

Polymorphism

Now the code is getting messy. And the dean has indicated that there are more schemes on the way, in this politically correct age of trying to be everything to everyone. Every time the dean adds a new scheme, you must change the code in the Student class. In changing Student, it is easy to break the class and other classes that depend on it.

You would like to *close* the Student class to any further changes. You know that the rest of the code in the class works just fine. Instead of chang-

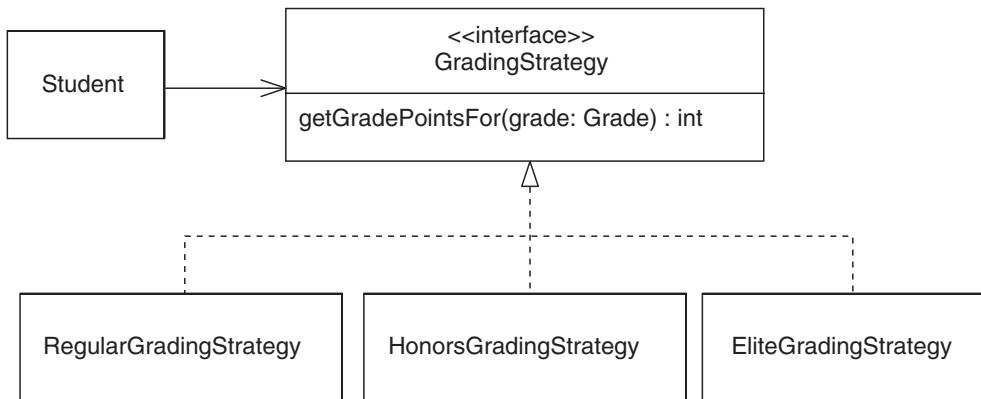


Figure 5.2 *Strategy*

ing the Student class each time the dean adds a new scheme, you want to support the new requirement by extending the system.⁴



Design your system to accommodate changes through extension, not modification.

You can consider the grading scheme to be a *strategy* that varies based on the type of student. You could create specialized student classes, such as HonorsStudent, RegularStudent, and PrivilegedStudent, but students can change status. Changing an object from being one type to being a different type is difficult.

Instead, you can create a class to represent each grading scheme. You can then assign the appropriate scheme to each student.

You will use an interface, GradingStrategy, to represent the abstract concept of a grading strategy. The Student class will store a reference of the interface type GradingStrategy, as shown in the UML diagram in Figure 5.2. In the diagram, there are three classes that implement the interface: RegularGradingStrategy, HonorsGradingStrategy, and EliteGradingStrategy. The closed-arrow relationship shown with a dashed line is known as the *realizes* association in UML. In Java terms, RegularGradingStrategy implements the GradingStrategy interface. In UML terms, RegularGradingStrategy realizes the GradingStrategy interface.

The GradingStrategy interface declares that any class implementing the interface will provide the ability to return the grade points for a given grade. You define GradingStrategy in a separate source file, GradingStrategy.java, as follows:

Polymorphism

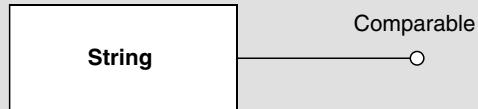
⁴[Martin2003], p. 99.

UML and Interfaces

More often than not, UML class diagrams are intended to show the structure of the system—the relations and thus the dependencies between its classes. As such, UML diagrams are best left uncluttered with detail. You should rarely show non-public methods, and often you might show no methods at all in a particular class. Diagram only interesting and useful things.

UML is best used as a simple semipictorial language for expressing some high-level concepts of a system. If you clutter your diagram with every possible detail, you will obscure the important things that you are trying to express.

Representing interfaces in UML is no different. Often it is valuable to show only that a class implements a particular interface. The methods defined in the interface might be generally understood and are also available in the code. UML gives you a visually terse way of showing that a class implements an interface. As an example, you can express the fact that the `String` method implements the `Comparable` interface with this UML diagram:



```

package sis.studentinfo;

public interface GradingStrategy {
    public int getGradePointsFor(Student.Grade grade);
}
  
```

Polymorphism

You represent each strategy with a separate class. Each strategy class implements the `GradingStrategy` interface and thus provides appropriate code for the `getGradePointsFor` method.

An interface defines methods that must be part of the public interface of the implementing class. All interface methods are thus `public` by definition. As such, you need not specify the `public` keyword for method declarations in an interface:

```

package sis.studentinfo;

public interface GradingStrategy {
    int getGradePointsFor(Student.Grade grade);
}

HonorsGradingStrategy:

package sis.studentinfo;

public class HonorsGradingStrategy implements GradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        ...
    }
}
  
```

```

        int points = basicGradePointsFor(grade);
        if (points > 0)
            points += 1;
        return points;
    }

    int basicGradePointsFor(Student.Grade grade) {
        if (grade == Student.Grade.A) return 4;
        if (grade == Student.Grade.B) return 3;
        if (grade == Student.Grade.C) return 2;
        if (grade == Student.Grade.D) return 1;
        return 0;
    }
}

```

RegularGradingStrategy:

```

package sis.studentinfo;

public class RegularGradingStrategy implements GradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        if (grade == Student.Grade.A) return 4;
        if (grade == Student.Grade.B) return 3;
        if (grade == Student.Grade.C) return 2;
        if (grade == Student.Grade.D) return 1;
        return 0;
    }
}

```

If you're savvy enough to spot the duplication in this code, take the time to refactor it away. You could introduce a BasicGradingStrategy and have it supply a static method with the common code. Or you can wait (only because I said so—otherwise never put off eliminating duplication!): The next lesson introduces another way of eliminating this duplication.

In StudentTest, instead of using setHonors to create an honors student, you send the message setGradingStrategy to the Student object. You pass an HonorsGradingStrategy instance as the parameter.

```

private Student createHonorsStudent() {
    Student student = new Student("a");
    student.setGradingStrategy(new HonorsGradingStrategy());
    return student;
}

```

Note how you were able to make this change in one place by having eliminated all duplication in the test code!

You then need to update the Student class to allow the strategy to be set and stored. Initialize the gradingStrategy instance variable to a RegularGradingStrategy object to represent the default strategy.

Polymorphism

```
public class Student {
    ...
    private GradingStrategy gradingStrategy =
        new RegularGradingStrategy();
    ...
    void setGradingStrategy(GradingStrategy gradingStrategy) {
        this.gradingStrategy = gradingStrategy;
    }
    ...
}
```

The next section, *Using Interface References*, explains why you can declare both the `gradingStrategy` instance variable and the parameter to `setGradingStrategy` as the type `GradingStrategy`.

Modify the `Student` code to obtain grade points by sending the `gradePointsFor` message to the object stored in the `gradingStrategy` reference.

```
int gradePointsFor(Grade grade) {
    return gradingStrategy.getGradePointsFor(grade);
}
```

Eliminate the instance variable `isHonors`, its associated setter method, and the method `basicGradePointsFor`.

Now that the `gradePointsFor` method does nothing but a simple delegation, you can *inline* its code to the `getGpa` method. In some cases, single-line delegation methods such as `gradePointsFor` can provide cleaner, clearer code, but in this case, the `gradePointsFor` method adds neither. Its body and name say the same thing, and no other code uses the method.

Inline the `gradePointsFor` method by replacing the call to it (from `getGPA`) with the single line of code in its body. You then can eliminate the `gradePointsFor` method entirely.

```
double getGpa() {
    if (grades.isEmpty())
        return 0.0;
    double total = 0.0;
    for (Grade grade: grades)
        total += gradingStrategy.getGradePointsFor(grade);
    return total / grades.size();
}
```

You now have an extensible, closed solution. Adding a third implementation of `GradingStrategy`, such as `EliteGradingStrategy`, is a simple exercise of creating a new class and implementing a single method. The code in `Student` can remain unchanged—you have closed it to any modifications in grading strategy. Also, you can make future changes to a particular grading strategy exclusive of all other grading strategies.

Polymorphism

Using Interface References

The code you implemented in `Student` uses `GradingStrategy` as the type for defining both instance variables and parameters:

```
public class Student implements Comparable<Student> {
    ...
    private GradingStrategy gradingStrategy =
        new RegularGradingStrategy();
    ...
    void setGradingStrategy(GradingStrategy gradingStrategy) {
        this.gradingStrategy = gradingStrategy;
    }
}
```

Even though you create a `RegularGradingStrategy` object and assign it to the `gradingStrategy` instance variable, you defined `gradingStrategy` as being of the type `GradingStrategy` and not of the type `RegularGradingStrategy`. Java allows this: A class that implements an interface is of that interface type. A `RegularGradingStrategy` implements the `GradingStrategy` interface, so it *is* a `GradingStrategy`.

When you assign an object to a reference variable of the `GradingStrategy` interface type, it is still a `RegularGradingStrategy` object in memory. The client working with a `GradingStrategy` reference, however, can send only `GradingStrategy` messages through that reference.

The code in `Student` doesn't care whether it is passed a `RegularGradingStrategy` or an `HonorsGradingStrategy`. The code in the `getGpa` method can send the `getGradePointsFor` message to the `gradingStrategy` reference without having to know the actual type of the instance stored at that reference.



Always prefer use of an interface type for variables and parameters.

Using Interface References

The use of interface types is a cornerstone to implementing systems with good object-oriented design. Interfaces are “walls of abstraction.” They allow you to define a means of interacting with an object through an abstraction—something that is unlikely to change. The alternative is to be dependent on a class with concrete details—something that is likely to change.

Why is this important? Because any time you introduce a change to class X, you must recompile and retest all classes that are dependent upon class X. You must also recompile and retest any classes dependent upon *those* classes. Changes to class X in Figure 5.3 could adversely impact both dependent classes A and B.

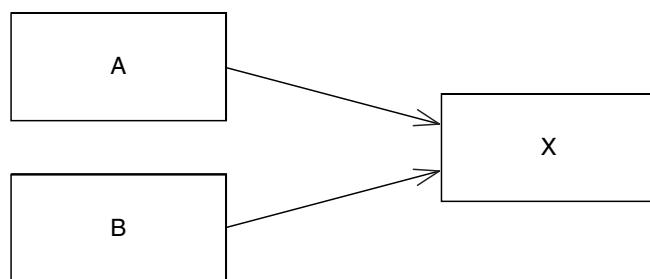


Figure 5.3 Changes to X Impact Both A and B

Interfaces *invert* the dependencies so that client classes are dependent upon abstractions.⁵ The class that implements the interface is also dependent upon this abstraction. But the client class is no longer dependent upon the concrete, changing class. In Figure 5.4, the interface XAbstraction represents abstract concepts from XImplementation. Classes A and B now communicate to XImplementation objects via this interface. Classes A and B now are dependent upon abstractions only. The XImplementation class depends upon the abstraction as well. No class in Figure 5.4 depends upon implementation details that are likely to change!

When designing object-oriented systems, you want to strive for this dependency arrangement. The more you depend upon concrete types, the more difficult your system is to change. The more you depend upon abstract types (interfaces), the easier your system is to change. By introducing interfaces, you build an abstraction barrier between the client and a concrete server class.

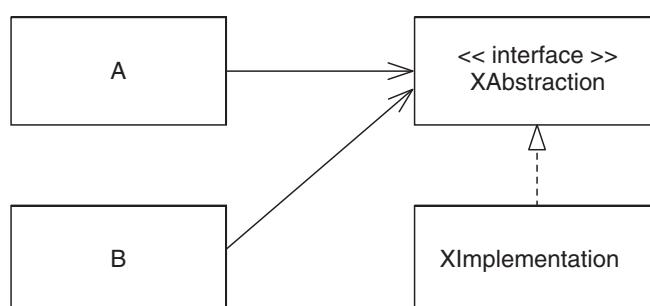


Figure 5.4 Inverted Dependencies

⁵[Martin2003], p. 127.

ArrayList and the List Interface

In earlier lessons, you created `ArrayList` instances and assigned them to references of the `ArrayList` type:

```
ArrayList<Student> students = new ArrayList<Student>();
```

The Java API documentation shows that `ArrayList` implements the `List` interface. If you browse the documentation for the `List` interface, you'll see that virtually all messages you had sent to `ArrayList` instances are defined in `List`. Thus, you should prefer defining `students` (in this example) as a `List` type:

```
List<Student> students = new ArrayList<Student>();
```

The `ArrayList` class defines a fixed-length array in memory. Its performance characteristics are good unless you are constantly inserting or deleting elements that appear before the end of the list. If your list needs these characteristics, a linked list is the better data structure, since it uses more dynamic memory management. The downside is that accessing elements in a linked list is done by serially traversing the list from the beginning, whereas accessing an element in an array is immediate, since it only needs to calculate an offset from the beginning of the memory block.

Java's implementation of a linked list is provided as `java.util.LinkedList`. If you use the `List` reference type throughout your code, you could change the performance characteristics of your application by making a change in one place:

```
List<Student> students = new LinkedList<Student>();
```

Exercises

For now, go through your source code and change as many references as possible to be an interface type.

Exercises

1. Create an enumeration for the two colors. Encapsulate the enumeration so that only the `Piece` class is aware of its existence. Depending on your implementation, this may require significant refactoring. Note each place you have to change in order to introduce the change—How could you have removed the need for any changes you felt were done twice?

You might consider also creating an enumeration for the piece values. If your code allows you to do so easily, go ahead. You may want to wait until Lesson 6, where you will learn how to associate data with enumeration values.

2. Introduce an enum for each piece. You will want to divorce the piece representation from the piece enum.
3. Create a separate factory method on Piece for each color/piece combination (e.g., `createWhitePawn`, `createBlackRook`).

```
package pieces;

import junit.framework.TestCase;

public class PieceTest extends TestCase {
    public void testCreate() {
        verifyCreation(
            Piece.createWhitePawn(), Piece.createBlackPawn(),
            Piece.Type.PAWN, Piece.PAWN_REPRESENTATION);
        verifyCreation(
            Piece.createWhiteRook(), Piece.createBlackRook(),
            Piece.Type.ROOK, Piece.ROOK_REPRESENTATION);
        verifyCreation(
            Piece.createWhiteKnight(), Piece.createBlackKnight(),
            Piece.Type.KNIGHT, Piece.KNIGHT_REPRESENTATION);
        verifyCreation(
            Piece.createWhiteBishop(), Piece.createBlackBishop(),
            Piece.Type.BISHOP, Piece.BISHOP_REPRESENTATION);
        verifyCreation(Piece.createWhiteQueen(), Piece.createBlackQueen(),
            Piece.Type.QUEEN, Piece.QUEEN_REPRESENTATION);
        verifyCreation(Piece.createWhiteKing(), Piece.createBlackKing(),
            Piece.Type.KING, Piece.KING_REPRESENTATION);
        Piece blank = Piece.noPiece();
        assertEquals('.', blank.getRepresentation());
        assertEquals(Piece.Type.NO_PIECE, blank.getType());
    }

    private void verifyCreation(Piece whitePiece, Piece blackPiece,
        Piece.Type type, char representation) {
        assertTrue(whitePiece.isWhite());
        assertEquals(type, whitePiece.getType());
        assertEquals(representation, whitePiece.getRepresentation());

        assertTrue(blackPiece.isBlack());
        assertEquals(type, blackPiece.getType());
        assertEquals(Character.toUpperCase(representation),
            blackPiece.getRepresentation());
    }
}
```

Exercises

Reduce the `createWhite` and `createBlack` methods to a single private factory method that uses an if clause.

4. Write code in Board to return the number of pieces, given the color and piece representation. For example, the below board should return 3 if you ask it for the number of black pawns. Calculate the count on demand (in other words, do not track the counts as you create pieces).

```
. K R . . . .
P . P B . . .
. P . . Q . .
. . . . .
. . . . n q .
. . . . p .
. . . . p .
. . . r k . .
```

5. Create a method to retrieve the piece at a given location. Given an initial board setup, asking for the piece at "a8" returns the black rook. The white king is at "e1".

Use utility methods defined on the Character class to help convert the individual characters of the location string to numeric indexes. Note: Your board may be flipped; in other words, rank 1 may be at the top and rank 8 at the bottom. Your code will have to invert the rank passed in.

```
R N B Q K B N R 8 (rank 8)
P P P P P P P P 7
. . . . . . . 6
. . . . . . . 5
. . . . . . . 4
. . . . . . . 3
p p p p p p p 2
r n b q k b n r 1 (rank 1)
a b c d e f g h
files
```

Exercises

In the next exercise, you will create the ability to place pieces on an empty board. Start by adding a test that verifies that a newly instantiated Board object contains no pieces. This will lead you to a bit of refactoring. You may need to replace use of the add method on ArrayList with the set method.

6. Create the code necessary to place pieces at arbitrary positions on the board. Make sure you refactor your solution with earlier code you've written to manage squares.

```
. . . . . . 8 (rank 8)
. . . . . . 7
. K . . . . 6
. R . . . . 5
. . k . . . . 4
. . . . . . 3
. . . . . . 2
```

```
..... 1 (rank 1)
a b c d e f g h
files
```

7. In chess programs, determining the relative strength of board positions is pivotal in determining which move to make. Using a very primitive board evaluation function, you will add up the values of the pieces on the board. Calculate this sum by adding 9 points for a queen, 5 points for a rook, 3 points for a bishop, and 2.5 points for a knight. Add 0.5 points for a pawn that is on the same file as a pawn of the same color and 1 point for a pawn otherwise. Remember to count the points for only one side at a time.

```
. K R ..... 8
P . P B ..... 7
. P . . Q ..... 6
..... 5
..... n q . 4
..... p . p 3
..... p p . 2
..... r k . . 1
a b c d e f g h
```

For the example shown, black would have a strength of 20, and white would have a strength of 19.5.

Develop your solution incrementally. Start with a single piece on the board. Add pieces to the board one by one, altering your assertions each time. Finally, add the complex scenario of determining if another pawn is in the same file.

Exercises

8. As you loop through all pieces on the board, assign the strength of each piece to the Piece object itself. For each side (black and white), gather the list of pieces in a collection. Ensure that the collection is sorted in order from the piece with the highest value to the piece with the lowest value.
9. Examine your code for an opportunity to create an interface. Does the system feel cleaner or more cluttered? Be wary of introducing an interface without need—remember that simplicity requires “fewest classes, fewest methods” shortly after “no duplication.”
10. Go through the code you have produced so far and look for opportunities to use early returns and guard clauses to simplify your code.
11. Go through the code you have produced so far and ensure that you are using interfaces rather than concrete implementations wherever possible. In particular, examine your usages of collections and ensure you are programming to the interfaces.

Also: While the static import facility can make code more difficult to follow, one appropriate place you can use it is in test code. Test code generally interacts with a single target class. If the target class contains class constants, you can use static import in the test class for these constants. The context should make it clear as to where the constants are defined.

Alter your code to make judicious use of `import static`.

Exercises

This page intentionally left blank

Lesson 6

Inheritance

In this lesson you will learn about:

- switch statements
- maps
- lazy initialization
- inheritance
- extending methods
- calling superclass constructors
- the principle of subcontracting

The switch Statement

In the last chapter, you coded methods in HonorsGradingStrategy and RegularGradingStrategy to return the proper GPA for a given letter grade. To do so, you used a succession of if statements. As a reminder, here is what some of the relevant code looks like (taken from HonorsGradingStrategy):

```
int basicGradePointsFor(Student.Grade grade) {  
    if (grade == Student.Grade.A) return 4;  
    if (grade == Student.Grade.B) return 3;  
    if (grade == Student.Grade.C) return 2;  
    if (grade == Student.Grade.D) return 1;  
    return 0;  
}
```

The switch Statement

Each of the conditionals in basicGradePointsFor compares a value to a single variable, grade. Another construct in Java that allows you to represent related comparisons is the switch statement:

```
int basicGradePointsFor(Student.Grade grade) {
    switch (grade) {
        case A: return 4;
        case B: return 3;
        case C: return 2;
        case D: return 1;
        default: return 0;
    }
}
```

As the target of the `switch` statement, you specify a variable or expression to compare against. In this example, the target is the `grade` parameter:

```
switch (grade) {
```

You then specify any number of `case` labels. Each `case` label specifies a single `enum` value. When the Java VM executes the `switch` statement, it determines the value of the target expression. It compares this value to each `case` label in turn.

If the target value matches a `case` label, the Java VM transfers control to the statement immediately following that `case` label. It skips statements between the `switch` target and the `case` label. If the target value matches no `case` label, the Java VM transfers control to the `default` case label, if one exists. The `default` case label is optional. If no `default` case label exists, the VM transfers control to the statement immediately following the `switch` statement.

Thus, if the value of the `grade` variable is `Student.Grade.B`, Java transfers control to the line that reads:

```
case B: return 3;
```

The Java VM executes this `return` statement, which immediately transfers control out of the method.

Case Label Are Just Labels

Case Labels Are Just Labels

A `case` label is just a label. The Java VM uses the `case` labels only when the Java VM initially encounters the `switch` statement. Once Java has located the matching `case` label and transferred control to it, it ignores the remainder of the `case` labels. Java executes the remainder of the code in the `switch` statement in top-to-bottom order. Java blissfully ignores the `case` labels and the `default` label, through to the end of the `switch` statement (or until another statement such as `return` transfers control).¹

¹You'll learn more about statements to transfer control in Lesson 7.

The following example demonstrates how the switch statement flows:

```
public void testSwitchResults() {
    enum Score
        { fieldGoal, touchdown, extraPoint,
          twoPointConversion, safety };

    int totalPoints = 0;

    Score score = Score.touchdown;

    switch (score) {
        case fieldGoal:
            totalPoints += 3;
        case touchdown:
            totalPoints += 6;
        case extraPoint:
            totalPoints += 1;
        case twoPointConversion:
            totalPoints += 2;
        case safety:
            totalPoints += 2;
    }
    assertEquals(6, totalPoints);
}
```

The test fails:

```
junit.framework.AssertionFailedError: expected<6> but was<11>
```

Since score was set to Score.touchdown, the Java VM transferred control to the line following the corresponding case label:

```
case touchdown:
    totalPoints += 6;
```

Once Java executed that line, it then executed the next three statements (and ignored the case labels):

```
case extraPoint: // ignored
    totalPoints += 1;
case twoPointConversion: // ignored
    totalPoints += 2;
case safety: // ignored
    totalPoints += 2;
```

Case Labels Are Just Labels

Total points thus is 6 plus 1 plus 2 plus 2, or 11. Test failure!

You obtain the desired behavior by inserting a break statement at the end of each section of code following a case label:

```
public void testSwitchResults() {
    enum Score
        { fieldGoal, touchdown, extraPoint,
```

```

        twoPointConversion, safety };

int totalPoints = 0;

Score score = Score.touchdown;

switch (score) {
    case fieldGoal:
        totalPoints += 3;
        break;
    case touchdown:
        totalPoints += 6;
        break;
    case extraPoint:
        totalPoints += 1;
        break;
    case twoPointConversion:
        totalPoints += 2;
        break;
    case safety:
        totalPoints += 2;
        break;
}
assertEquals(6, totalPoints);
}
}

```

The `break` statement is another statement that transfers control flow. It transfers control out of the `switch` statement to the next statement following. In the example, executing any `break` statement results in control being transferred to the `assertEquals` statement.

Multiple case labels can precede a statement appearing within the `switch` statement:

```

switch (score) {
    case fieldGoal:
        totalPoints += 3;
        break;
    case touchdown:
        totalPoints += 6;
        break;
    case extraPoint:
        totalPoints += 1;
        break;
    case twoPointConversion:
    case safety:
        totalPoints += 2;
        break;
}

```

Case Labels Are Just Labels

If `score` matches either `Score.twoPointConversion` or `Score.safety`, the code adds two points to the total.

In addition to switching on `enum` values, you can switch on `char`, `byte`, `short`, or `int` values. You cannot, unfortunately, switch on `String` values.

Your code should not contain a lot of `switch` statements. Also, it should not contain lots of multiple `if` statements that accomplish the same goal. Often, you can eliminate `switch` statements in favor of polymorphic solutions, as in the previous lesson. Your main guides as to whether you should replace `switch` statements with polymorphism are duplication and frequency/ease of maintenance.

Before continuing, refactor `RegularGradingStrategy` to use a `switch` statement instead of multiple `if` statements.

Maps

Yet another alternative to using a `switch` statement is to use a map. A map is a collection that provides fast insertion and retrieval of values associated with specific keys. An example is an online dictionary that stores a definition (a value) for each word (key) that appears in it.

Java supplies the interface `java.util.Map` to define the common behavior for all map implementations. For the report card messages, you will use the `EnumMap` implementation. An `EnumMap` is a Map with the additional constraint that all keys must be `enum` objects.

To add a key-value pair to a Map, you use the `put` method, which takes the key and value as parameters. To retrieve a value stored at a specific key, you use the `get` method, which takes the key as a parameter and returns the associated value.

Suppose you need to print an appropriate message on a report card for each student, based on their grade. Add a new class named `ReportCardTest` to the `sis.reports` package.

```
package sis.report;

import junit.framework.*;
import sis.studentinfo.*;

public class ReportCardTest extends TestCase {
    public void testMessage() {
        ReportCard card = new ReportCard();
        assertEquals(ReportCard.A_MESSAGE,
                    card.getMessage(Student.Grade.A));
        assertEquals(ReportCard.B_MESSAGE,
                    card.getMessage(Student.Grade.B));
        assertEquals(ReportCard.C_MESSAGE,
```

Maps

```

        card.getMessage(Student.Grade.C));
assertEquals(ReportCard.D_MESSAGE,
            card.getMessage(Student.Grade.D));
assertEquals(ReportCard.F_MESSAGE,
            card.getMessage(Student.Grade.F));
    }
}

```

Change the Grade enum defined in Student to public to make this code compile.

The ReportCard class:

```

package sis.report;

import java.util.*;
import sis.studentinfo.*;

public class ReportCard {
    static final String A_MESSAGE = "Excellent";
    static final String B_MESSAGE = "Very good";
    static final String C_MESSAGE = "Hmmm...";
    static final String D_MESSAGE = "You're not trying";
    static final String F_MESSAGE = "Loser";

    private Map<Student.Grade, String> messages = null;

    public String getMessage(Student.Grade grade) {
        return getMessages().get(grade);
    }

    private Map<Student.Grade, String> getMessages() {
        if (messages == null)
            loadMessages();
        return messages;
    }

    private void loadMessages() {
        messages =
            new EnumMap<Student.Grade, String>(Student.Grade.class);
        messages.put(Student.Grade.A, A_MESSAGE);
        messages.put(Student.Grade.B, B_MESSAGE);
        messages.put(Student.Grade.C, C_MESSAGE);
        messages.put(Student.Grade.D, D_MESSAGE);
        messages.put(Student.Grade.F, F_MESSAGE);
    }
}

```

Maps

The ReportCard class defines an instance variable messages as a parameterized Map type. The type parameters are Student.Grade for the key and String for the value. When you construct an EnumMap, you must also pass in the class that represents the enum type of its keys.

The `getMessages` method uses lazy initialization (see the sidebar) to load the appropriate message strings, which you define as static constants, into a new instance of the `EnumMap`.

Lazy Initialization

You have learned to initialize fields either where you declared them or within a constructor. Another option is to wait until you first need to use a field and then initialize it. This is a technique known as *lazy initialization*.

Within a getter method, such as `getMessages`, you first test to see whether or not a field has already been initialized. For a reference instance variable, you test whether or not it is `null`. If it is `null`, you do whatever initialization is necessary and assign the new value to the field. Regardless, you return the field as the result of the getter, like you normally would.

The primary use of lazy initialization is to defer the potentially costly operation of loading a field until it is necessary. If the field is never needed, you never expend the cycles required to load it. A minor amount of overhead is required to test the field each time it is accessed. This overhead is negligible for occasional access.

Here, lazy initialization is used as a way of keeping more-complex initialization logic close to the field access logic. Doing so can improve understanding of how the field is initialized and used.

The `getMessage` method is a single line of code that uses the `Map` method `get` to retrieve the appropriate string value for the grade key.

Maps are extremely powerful, fast data structures that you might use frequently in your applications. Refer to Lesson 9 for a detailed discussion of their use.

Inheritance

Inheritance

The code for `RegularGradingStrategy` and `HonorsGradingStrategy` contains duplicate logic. The code that you used to derive a basic grade is the same in both classes—both classes contain `switch` statements with the exact same logic:

```
switch (grade) {  
    case A:  return 4;  
    case B:  return 3;  
    case C:  return 2;  
    case D:  return 1;  
    default: return 0;  
}
```

One solution for eliminating this duplication is to use inheritance. I discussed inheritance briefly in the Agile Java Overview chapter. You can factor the commonality between RegularGradingStrategy and HonorsGradingStrategy into a common class called BasicGradingStrategy. You can then declare the classes RegularGradingStrategy and HonorsGradingStrategy as *subclasses* of BasicGradingStrategy. As a subclass, each of RegularGradingStrategy and HonorsGradingStrategy obtain all of the behaviors of BasicGradingStrategy.

You will refactor to an inheritance-based solution in very small increments. The first step is to create a BasicGradingStrategy class:

```
package sis.studentinfo;

public class BasicGradingStrategy {
```

Then declare both RegularGradingStrategy and HonorsGradingStrategy as subclasses of BasicGradingStrategy by using the `extends` keyword. The `extends` clause must appear before any `implements` clause.

```
// RegularGradingStrategy.java
package sis.studentinfo;

public class RegularGradingStrategy
    extends BasicGradingStrategy
    implements GradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        switch (grade) {
            case A: return 4;
            case B: return 3;
            case C: return 2;
            case D: return 1;
            default: return 0;
        }
    }
}

// HonorsGradingStrategy.java
package sis.studentinfo;

public class HonorsGradingStrategy
    extends BasicGradingStrategy
    implements GradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        int points = basicGradePointsFor(grade);
        if (points > 0)
            points += 1;
        return points;
    }

    int basicGradePointsFor(Student.Grade grade) {
        switch (grade) {
```

Inheritance

```
    case A: return 4;  
    case B: return 3;  
    case C: return 2;  
    case D: return 1;  
    default: return 0;  
}  
}  
}
```

You can now move the common code to a method on the `BasicGradingStrategy` *superclass* (also known as the *base class*). First, move the method `basicGradePointsFor` from `HonorsGradingStrategy` to `BasicGradingStrategy`:

```
// HonorsGradingStrategy.java
package sis.studentinfo;

public class HonorsGradingStrategy
    extends BasicGradingStrategy
    implements GradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        int points = basicGradePointsFor(grade);
        if (points > 0)
            points += 1;
        return points;
    }
}

// BasicGradingStrategy.java
package sis.studentinfo;

public class BasicGradingStrategy {
    int basicGradePointsFor(Student.Grade grade) {
        switch (grade) {
            case A: return 4;
            case B: return 3;
            case C: return 2;
            case D: return 1;
            default: return 0;
        }
    }
}
```

Inheritance

Recompile and test.

Even though HonorsGradingStrategy no longer contains the definition for basicGradePointsFor, it can call it just as if it were defined in the same class. This is akin to your ability to call assertion methods in your test classes, because they extend from junit.framework.TestCase. Conceptually, HonorsGradingStrategy is a BasicGradingStrategy, much as a class that implements an interface is of that interface type. Thus HonorsGradingStrategy can use any (nonprivate) method defined in BasicGradingStrategy. (We'll discuss the non-private distinction shortly.)

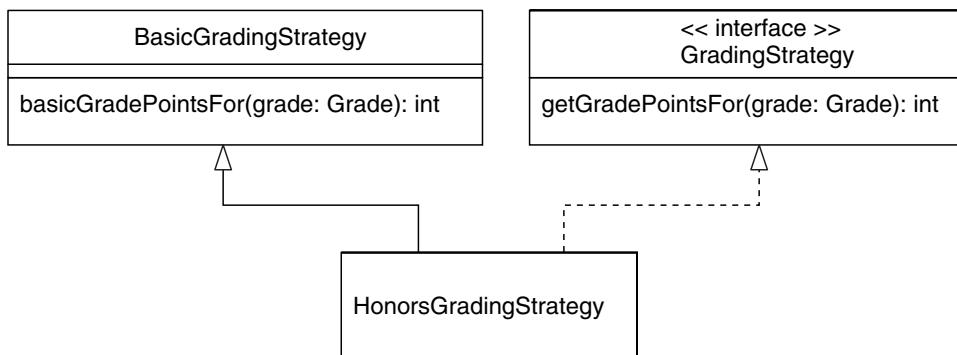


Figure 6.1 Inheriting from `BasicGradingStrategy`

The UML representation of this relationship appears in Figure 6.1.

Next, modify `RegularGradingStrategy` to reuse the method `basicGradePointsFor`.

```

package sis.studentinfo;

public class RegularGradingStrategy
    extends BasicGradingStrategy
    implements GradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        return basicGradePointsFor(grade);
    }
}
  
```

Abstract Classes

Abstract Classes

You could stop here and have a sufficient solution. You could also go one step further and eliminate a bit more duplication by having the superclass `BasicGradingStrategy` implement the `GradingStrategy` interface directly. The only problem is, you don't necessarily have an implementation for `getGradePointsFor`.

Java allows you to define a method as abstract, meaning that you cannot or do not wish to supply an implementation for it. Once a class contains at least one abstract method, the class itself must similarly be defined as abstract. You cannot create new instances of an abstract class, much as you cannot directly instantiate an interface.

Any class extending from an abstract class must either implement all inherited abstract methods, otherwise it too must be declared as abstract.

Change the declaration of BasicGradingStrategy to implement the GradingStrategy interface. Then supply an abstract declaration for the getGradePointsFor method:

```
package sis.studentinfo;

abstract public class BasicGradingStrategy implements GradingStrategy {
    abstract public int getGradePointsFor(Student.Grade grade);

    int basicGradePointsFor(Student.Grade grade) {
        switch (grade) {
            case A: return 4;
            case B: return 3;
            case C: return 2;
            case D: return 1;
            default: return 0;
        }
    }
}
```

The subclasses no longer need to declare that they implement the GradingStrategy interface:

```
// HonorsGradingStrategy.java
package sis.studentinfo;

public class HonorsGradingStrategy extends BasicGradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        int points = basicGradePointsFor(grade);
        if (points > 0)
            points += 1;
        return points;
    }
}

// RegularGradingStrategy.java
package sis.studentinfo;

public class RegularGradingStrategy extends BasicGradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        return basicGradePointsFor(grade);
    }
}
```

**Abstract
Classes**

Extending Methods

A third solution² is to recognize that the `RegularGradingStrategy` is virtually the same as `BasicGradingStrategy`. The method `getGradePointsFor` as defined in `HonorsGradingStrategy` is basically an extension of the method in `RegularGradingStrategy`—it does what the base class method does, plus a little more. In other words, you can supply a default definition for `getGradePointsFor` in `BasicGradingStrategy` that will be extended in `HonorsGradingStrategy`.

Move the definition for `getGradePointsFor` from `RegularGradingStrategy` to `BasicGradingStrategy`. You should no longer declare `BasicGradingStrategy` as abstract, since all of its methods supply definitions.

```
// BasicGradingStrategy.java
package sis.studentinfo;

public class BasicGradingStrategy implements GradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        return basicGradePointsFor(grade);
    }

    int basicGradePointsFor(Student.Grade grade) {
        switch (grade) {
            case A: return 4;
            case B: return 3;
            case C: return 2;
            case D: return 1;
            default: return 0;
        }
    }
}

// RegularGradingStrategy.java
package sis.studentinfo;

public class RegularGradingStrategy extends BasicGradingStrategy {
```

**Extending
Methods**

`RegularGradingStrategy` no longer defines any methods!

(Compile, test.)

Next, change the code in `HonorsGradingStrategy` to extend the superclass method `getGradePointsFor`. To extend a method, you define a method with the same name in a subclass. In that subclass method, you call the superclass method to effect its behavior. You supply additional, specialized behavior in the remainder of the subclass method.

²Are you getting the picture that there are many ways to skin a cat in Java? All these options allow you to make your code as expressive as possible. It also means that there is no one “perfect” way to implement anything of complexity.

In comparison to extending, Java will let you fully supplant a method's definition with a completely new one that has nothing to do with the original. This is known as *overriding* a method. This is a semantic definition: The Java compiler knows no distinction between overriding and extending; it's all overriding to the compiler. You, on the other hand, recognize an extending method by its call to the superclass method of the same name.

Prefer extending to overriding.



Your subclass definition should specialize the behavior of the superclass method in some manner, not change it completely.

To explicitly call a method in a superclass, use the `super` keyword to scope the message send, as shown in bold:

```
package sis.studentinfo;

public class HonorsGradingStrategy extends BasicGradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        int points = super.getGradePointsFor(grade);
        if (points > 0)
            points += 1;
        return points;
    }
}
```

When you use the `super` keyword, the Java VM looks in the superclass to find the corresponding method definition.

Refactoring

The method `basicGradePointsFor` in `BasicGradingStrategy` is now superfluous. You can inline it and eliminate the method `basicGradePointsFor`.

```
package sis.studentinfo;

public class BasicGradingStrategy implements GradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        switch (grade) {
            case A: return 4;
            case B: return 3;
            case C: return 2;
            case D: return 1;
            default: return 0;
        }
    }
}
```

Refactoring

Is there any more need for the separate subclass RegularGradingStrategy? There's not much left to it:

```
package studentinfo;

public class RegularGradingStrategy
    extends BasicGradingStrategy {
}
```

You can eliminate this class by changing the gradeStrategy reference in Student to use BasicGradingStrategy as the default.

```
public class Student {
    ...
    private GradingStrategy gradingStrategy =
        new BasicGradingStrategy();
    ...
}
```

All the work you have done since learning about inheritance has been on the code side only. You have not changed any tests. (I hope you have been running your tests with each incremental change.) You have not changed any behavior; you have only changed the way in which the existing behavior is implemented. The tests in Student for GPA adequately cover the behavior.

However, the tests in Student cover behavior within the context of managing students. You want to additionally provide tests at the unit level. The general rule is to have at least one test class for each production class. You should test each of HonorsGradingStrategy and BasicGradingStrategy individually.

Here is a test for BasicGradingStrategy:

Refactoring

```
package sis.studentinfo;

import junit.framework.*;

public class BasicGradingStrategyTest extends TestCase {
    public void testGetGradePoints() {
        BasicGradingStrategy strategy = new BasicGradingStrategy();
        assertEquals(4, strategy.getGradePointsFor(Student.Grade.A));
        assertEquals(3, strategy.getGradePointsFor(Student.Grade.B));
        assertEquals(2, strategy.getGradePointsFor(Student.Grade.C));
        assertEquals(1, strategy.getGradePointsFor(Student.Grade.D));
        assertEquals(0, strategy.getGradePointsFor(Student.Grade.F));
    }
}
```

A corresponding test for HonorsGradingStrategy:

```
package sis.studentinfo;

import junit.framework.*;

public class HonorsGradingStrategyTest extends TestCase {
    public void testGetGradePoints() {
        GradingStrategy strategy = new HonorsGradingStrategy();
        assertEquals(5, strategy.getGradePointsFor(Student.Grade.A));
        assertEquals(4, strategy.getGradePointsFor(Student.Grade.B));
        assertEquals(3, strategy.getGradePointsFor(Student.Grade.C));
        assertEquals(2, strategy.getGradePointsFor(Student.Grade.D));
        assertEquals(0, strategy.getGradePointsFor(Student.Grade.F));
    }
}
```

Don't forget to add these tests to the suite defined in AllTests.

Enhancing the Grade Enum

The number of basic grade points is information that can be associated directly with the Grade enum. It is possible to enhance the definition of an enum to include instance variables, constructors, and methods, just like any other class type. The main restriction is that you cannot extend an enum from another enum.

Modify the declaration of the Grade enum in the Student class:

```
public class Student {
    public enum Grade {
        A(4),
        B(3),
        C(2),
        D(1),
        F(0);

        private int points;

        Grade(int points) {
            this.points = points;
        }

        int getPoints() {
            return points;
        }
    }
    ...
}
```

Enhancing the Grade Enum

You have now associated each named instance of the enum with a parameter. Further, you terminated the list of enum instances with a semicolon. After that semicolon, code in the Grade enum declaration looks just like that in any class type. The parameter associated with each named enum instance is passed to the constructor of Grade. This `points` parameter is stored in an instance variable named `points`. You retrieve the points via the `getPoints` method.

You can now simplify the code in `BasicGradingStrategy` to:

```
package sis.studentinfo;

public class BasicGradingStrategy implements GradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        return grade.getPoints();
    }
}
```

No more switch statement!

Summer Course Sessions

The `CourseSession` class currently supports only 15-week sessions (with a one-week break) that occur in the spring and fall. The university also needs support for summer sessions. Summer sessions will begin in early June. They last eight weeks with no break.

One solution would be to pass the session length into the `CourseSession` class, store it, and use it as a multiplier when calculating the session end date.

However, there are many other differences between summer sessions and spring/fall sessions, including differences in maximum class session, credit values, and so on. For this reason, you have chosen to create a new class named `SummerCourseSession`.

The simplest approach for implementing `SummerCourseSession`, given what you know, is to have it extend from `CourseSession`. A `SummerCourseSession` is a `CourseSession` with some refinement on the details (see Figure 6.2).

You want to create the `SummerCourseSession` classes (test and production) in a new package, `summer`, for organizational purposes.

The test you initially code verifies that `SummerCourseSession` calculates the course end date correctly:

```
package sis.summer;

import junit.framework.*;
import java.util.*;
import sis.studentinfo.*;
```

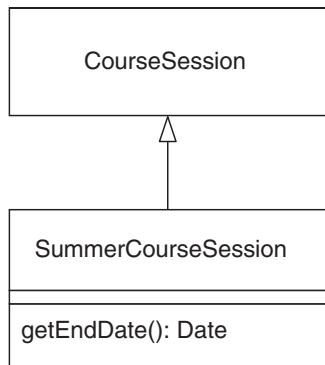


Figure 6.2 *SummerCourseSession Specialization*

```

public class SummerCourseSessionTest extends TestCase {
    public void testEndDate() {
        Date startDate = DateUtil.createDate(2003, 6, 9);
        CourseSession session =
            SummerCourseSession.create("ENGL", "200", startDate);
        Date eightWeeksOut = DateUtil.createDate(2003, 8, 1);
        assertEquals(eightWeeksOut, session.getEndDate());
    }
}
  
```

Calling Superclass Constructors

An initial noncompiling attempt at an implementation for SummerCourseSession follows.

```

// this code does not compile!
package sis.summer;

import java.util.*;
import sis.studentinfo.*;

public class SummerCourseSession extends CourseSession {
    public static SummerCourseSession create(
        String department,
        String number,
        Date startDate) {
        return new SummerCourseSession(department, number, startDate);
    }

    private SummerCourseSession(
        String department,
        String number,
        Date startDate)
  
```

Calling
Superclass
Constructors

```

        Date startDate) {
    super(department, number, startDate);
}

Date getEndDate() {
    GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(startDate);
    int sessionLength = 8;
    int daysInWeek = 7;
    int daysFromFridayToMonday = 3;
    int numberOfDays =
        sessionLength * daysInWeek - daysFromFridayToMonday;
    calendar.add(Calendar.DAY_OF_YEAR, numberOfDays);
    return calendar.getTime();
}
}

```

The key things to note are in bold.

First, you want to declare SummerCourseSession as a subclass of CourseSession, so you use the `extends` keyword in the class declaration.

Second, the constructor for SummerCourseSession makes a call to the superclass constructor that takes the department, course number, and start date as parameters. It does this by using the `super` keyword. This is similar to a `super` method invocation.

Third, in order to quickly get things working, you copied the `getEndDate` method from CourseSession and altered the value of the `sessionLength` temporary to be 8 instead of 16.

This code will not compile. You should see three compiler errors. The first one you know how to fix:

```
getEndDate() is not public in sis.studentinfo.CourseSession; cannot be accessed from outside
package
    assertEquals(eightWeeksOut, session.getEndDate());
    ^

```

**Calling
Superclass
Constructors**

Change the definition of `getEndDate` in CourseSession to `public` and recompile:

```
public Date getEndDate() {
```

You now receive a different error related to `getEndDate`:

```
getEndDate() in sis.summer.SummerCourseSession cannot override getEndDate() in
sis.studentinfo.CourseSession; attempting to assign weaker access privileges; was public
    Date getEndDate() {
    ^

```

A subclass method that overrides a superclass method must have equivalent or looser access privileges than a superclass method. In other words, CourseSession's control over the `getEndDate` method must be tighter than the control of

any of its subclasses. If a superclass marks a method as package, a subclass can mark the overridden method as package or public. If a superclass marks a method as public, a subclass must mark the overridden method as public.

The solution is to change `getEndDate` in `SummerCourseSession` to be public:

```
public Date getEndDate() {
```

After compiling, you should have two remaining errors that appear similar:

```
CourseSession(java.util.Date) has private access in sis.studentinfo.CourseSession
    super(department, number, startDate);
    ^
startDate has private access in sis.studentinfo.CourseSession
    calendar.setTime(startDate);
    ^
```

Sure enough, the `CourseSession` constructor that takes a `Date` as a parameter is marked `private`:

```
private CourseSession(
    String department, String number, Date startDate) {
    // ...
```

You made the constructor `private` in order to force clients to construct `CourseSession` instances using the class creation method. The problem is, the `private` access modifier restricts access to the constructor. Only code defined within `CourseSession` itself can invoke its constructor.

Your intent is to expose the `CourseSession` constructor to code only in `CourseSession` itself or code in any of its subclasses. Even if you define a `CourseSession` subclass in a different package, it should have access to the `CourseSession` constructor. You don't want to expose the constructor to other classes.

Specifying `public` access would allow too much access. Non-subclasses in different packages would be able to directly construct `CourseSession` objects. Package access would not allow subclasses in different packages to access the `CourseSession` constructor.

Java provides a fourth (and final) access modifier, `protected`. The `protected` keyword indicates that a method or field may be accessed by the class in which it is defined, by any other class in the same package, or by any of its subclasses. A subclass has access to anything in its superclass marked as `protected`, even if you define the subclass in a package other than the superclass.

Change the `CourseSession` constructor from `private` to `protected`:

```
protected CourseSession(
    String department, String number, Date startDate) {
    // ...
```

Calling
Superclass
Constructors

The compilation error regarding the constructor goes away. You have one remaining error: `getEndDate` references the private superclass field `startDate` directly.

```
startDate has private access in sis.studentinfo.CourseSession
    calendar.setTime(startDate);
    ^
```

You could solve the problem by changing `startDate` to be a protected field. A better solution is to require that subclasses, like any other class, access the field by using a method.

Change the `getStartDate` method from package to protected access.

```
public class CourseSession implements Comparable<CourseSession> {
    ...
    protected Date getStartDate() {
        return startDate;
    }
    ...
}
```

You must change the code in both `CourseSession` and `SummerCourseSession` to use this getter:

```
// CourseSession.java
public Date getEndDate() {
    GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(getStartDate());
    final int sessionLength = 16;
    final int daysInWeek = 7;
    final int daysFromFridayToMonday = 3;
    int numberOfDays =
        sessionLength * daysInWeek - daysFromFridayToMonday;
    calendar.add(Calendar.DAY_OF_YEAR, numberOfDays);
    return calendar.getTime();
}

// SummerCourseSession.java:
public Date getEndDate() {
    GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(getStartDate());
    int sessionLength = 8;
    int daysInWeek = 7;
    int daysFromFridayToMonday = 3;
    int numberOfDays =
        sessionLength * daysInWeek - daysFromFridayToMonday;
    calendar.add(Calendar.DAY_OF_YEAR, numberOfDays);
    return calendar.getTime();
}
```

**Calling
Superclass
Constructors**

Since I told you to be lazy and copy the code for `getEndDate`, note that you had the additional effort of making the change in two places.

In `SummerCourseSession`, `getEndDate` makes a call to `getStartDate`, but `getStartDate` is not defined in `SummerCourseSession`. Since the `super.` scope modifier was not used, Java first looks to see if `getStartDate` is defined in `SummerCourseSession`. It is not, so Java starts searching up the inheritance chain until it finds an accessible definition for `getStartDate`.

Protected-level access is one step “looser” than package. One minor downside is that it is possible for other classes in the same package to access a protected element (method or field). Usually this is not what you want. There are few good design reasons to make a field accessible to other classes in the same package *and* to subclasses in different packages but not to other classes.

In the case of `CourseSession`, you want the constructor to be accessible *only* to subclasses, regardless of package, and not to any other classes. There is no way in Java to enforce this restriction.

Refactoring

2800703

Time to eliminate the duplicate code you hastily introduced.

The only change between `getEndDate` as defined in `CourseSession` and `SummerCourseSession` is the session length. Define a protected method in `CourseSession` to return this value:

```
public class CourseSession implements Comparable<CourseSession> {
    ...
    protected int getSessionLength() {
        return 16;
    }
    ...
}
```

Refactoring

Change `getEndDate` to call `getSessionLength`:

```
public Date getEndDate() {
    GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(getStartDate());
    final int daysInWeek = 7;
    final int daysFromFridayToMonday = 3;
    int numberofDays =
        getSessionLength() * daysInWeek - daysFromFridayToMonday;
    calendar.add(Calendar.DAY_OF_YEAR, numberofDays);
    return calendar.getTime();
}
```

In `SummerCourseSession`, override the method `getSessionLength`. Summer sessions are 8 weeks, not 16. In this case, overriding is a fine alternative to extending, since you are not changing behavior, only data-related details of the behavior.

```
public class SummerCourseSession extends CourseSession {
    ...
    @Override
    protected int getSessionLength() {
        return 8;
    }
    ...
}
```

When you override a method, you should precede it with an `@Override` annotation. You use annotations to mark, or annotate, specific portions of code. Other tools, such as compilers, IDEs, and testing tools, can read and interpret these annotations. The Java compiler is the tool that reads the `@Override` annotation.

The compiler ensures that for a method marked as `@Override`, a method with the same name and arguments exists in a superclass. If you screwed up somehow—perhaps you mistyped the method as `getSessionLenth` instead of `getSessionLength`—the compile fails. You see the message:

```
method does not override a method from its superclass
```

Java does not require you to add `@Override` annotations to your code. However, they provide valuable documentation for your code as well as protections from simple mistakes. You can read more about annotations in Lesson 15.

You may now remove `getEndDate` from `SummerCourseSession`. In Figure 6.3, the `getSessionLength` operation appears twice, an indication that the subclass (`SummerCourseSession`) overrides the superclass (`CourseSession`) definition. Note that I have preceded each method with access privilege information, something I rarely do. Since normally I *only* display public information in a UML sketch, I prefer to omit the + (plus sign) that indicates a publicly accessible operation.

In this circumstance, however, `getSessionLength` is `protected`, not `public`. Adding the access privilege indicators to the diagram in Figure 6.3 differentiates access between the methods and highlights the relevancy of `getSessionLength`. The UML indicator for `protected` is the pound sign (#).³

Refactoring

³The UML access indicator for `private` is (-). The indicator for package-level access is (~).

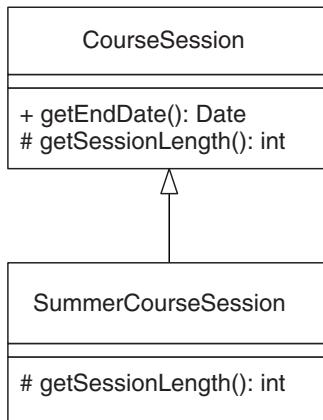


Figure 6.3 Overriding a Protected Method

Idiosyncrasies

Since Java does not require you to add `@Override` annotations to methods, it's easy to forget. (That's one thing a pair developer can be good for.) The `@Override` annotation is a new J2SE 5.0 feature, so I have my excuse if I've forgotten to use it elsewhere in Agile Java.

You'll note that I do not use `@Override` on my `setUp` and `tearDown` methods, even though the annotation is applicable for these `TestCase` method overrides. Using `@Override` would be a good idea (particularly since it's easy enough to spell `setUp` as `setup`). But old habits are hard to break. I justify it by saying that it's only test code and the tests will help correct me anyway. Nonetheless, I recommend that you avoid my bad habits.

Most of the logic for determining the end date of a session is fixed. The only variance in the logic between `CourseSession` and `SummerCourseSession` is the session length. This variance is represented by the abstraction of a separate method, `getSessionLength`. The `CourseSession` class can supply one implementation for `getSessionLength`, the `SummerCourseSession` another.

The method `getEndDate` in `CourseSession` acts as a template. It supplies the bulk of the algorithm for calculating a session end date. Certain pieces of the algorithm are “pluggable,” meaning that the details are provided elsewhere. Subclasses can vary these details. The template algorithm in `getEndDate` is an example of a design pattern known as Template Method.⁴

Refactoring

⁴[Gamma1995].

More on Constructors

You call a superclass constructor from a subclass constructor by using the `super` keyword, as demonstrated earlier. A call to a superclass constructor must appear as the first line in a subclass constructor.

A subclass extends, or builds upon, a superclass. You can think of a subclass object as an outer layer, or shell, around an object of the base class. Before the subclass object can exist, Java must build and properly initialize an object of the superclass. The following language test demonstrates how classes are constructed, using a demo class called `SuperClass` and its subclass `SubClass`.

```
// SuperClassTest.java
import junit.framework.TestCase;

public class SuperClassTest extends TestCase {
    public void testConstructorCalls() {
        SuperClass superClass = new SubClass();
        assertTrue(SuperClass.constructorWasCalled);
    }
}

// SuperClass.java
class SuperClass {
    static boolean constructorWasCalled = false;

    SuperClass() {
        constructorWasCalled = true;
    }
}

// SubClass.java
class SubClass extends SuperClass {
    SubClass() {
    }
}
```

More on Constructors

The Java virtual machine requires every class to have at least one constructor. However, you can code a class without explicitly defining a constructor. If you don't define a constructor Java will automatically generate a default, no-argument constructor.

If you do not provide a `super` call in a subclass constructor, it is as if Java inserted a call to the no-argument superclass constructor. Even constructors that take parameters call the no-argument superclass constructor by default.

```
// SuperClassTest.java
import junit.framework.TestCase;
```

```

public class SuperClassTest extends TestCase {
    public void testConstructorCalls() {
        SuperClass superClass = new SubClass("parm");
        assertTrue(SuperClass.constructorWasCalled);
    }
}

// SubClass.java
class SubClass extends SuperClass {
    SubClass(String parm) {
    }
}

```

The test constructs a new SubClass instance, passing in a String as a parameter. The parameter information appears in bold. The test then demonstrates that creating a SubClass instance results the execution of the SuperClass no-argument constructor.

Remember, if you *do* supply a constructor with arguments in a class, Java *does not* automatically supply a no-argument constructor. In this situation, any subclass constructors *must* explicitly call a superclass constructor, otherwise you receive a compilation error.

```

// This code will not compile
// SuperClass.java
class SuperClass {
    static boolean constructorWasCalled = false;

    SuperClass(String parm) {
        constructorWasCalled = true;
    }
}

// SubClass.java
class SubClass extends SuperClass {
    SubClass(String parm) {
    }
}

```

The above code generates the following compiler error:

```

SubClass.java: cannot find symbol
symbol  : constructor SuperClass()
location: class studentinfo.SuperClass
    SubClass(String parm) {
                           ^

```

More on Constructors

The subclass constructor should probably be:

```

class SubClass extends SuperClass {
    SubClass(String parm) {
        super(parm);
    }
}

```

Inheritance and Polymorphism

The class `SummerCourseSession` inherits from `CourseSession`. Just as a class that implements an interface is of that interface type, a class that inherits from another is of that superclass type. Therefore, `SummerCourseSession` is a `CourseSession`.

Just as you were taught to prefer interface reference types, you should also prefer base class for the reference type. Assign a new `SummerCourseSession` to a `CourseSession` reference:

```
CourseSession session = new SummerCourseSession();
```

Even though the `session` variable is of the type `CourseSession`, it still refers to a `SummerCourseSession` in memory.

You defined the method `getEndDate` in both `CourseSession` and `SummerCourseSession`. When you send the `getEndDate` message using the `session` variable, the message is received by a `SummerCourseSession` object. Java executes the version of `getEndDate` defined in `SummerCourseSession`.

The client thinks it is sending a message to a `CourseSession`, but an object of a class *derived from* `CourseSession` receives and interprets the message. This is another example of polymorphism.

The Principle of Subcontracting

The Principle of
Subcontracting

The client that sends a message to an object using a variable of `CourseSession` type expects it to be interpreted and acted upon in a certain manner. In test-driven development, unit tests define this “manner.” A unit test describes the behavior supported by interacting with a class through its interface. It describes the behavior by first actually effecting that behavior, then by ensuring that any number of *postconditions* hold true upon completion.

For a class to properly derive from `CourseSession`, it must not change the expectations of behavior. Any postcondition that held true for a test of `CourseSession` should also hold true for a test executing against a subclass of `CourseSession`. In general, subclasses should keep or strengthen postconditions. Subclasses should *extend* the behavior of their superclasses, meaning that postconditions should be *added* to tests for the subclass.

What this means for TDD is that you should specify contracts (in the form of unit tests) at the level of the superclass. All subclasses will need to conform

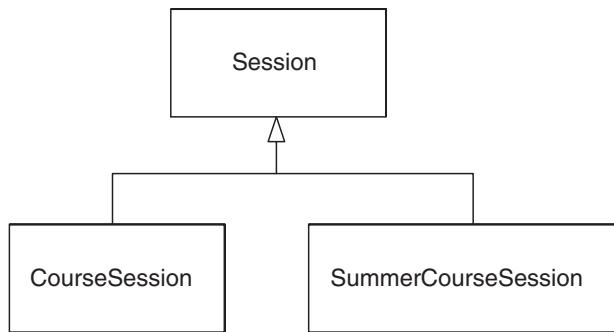


Figure 6.4 *The Session Hierarchy*

to these unit tests. You will build these contracts using a pattern known as Abstract Test.⁵

Technically, there is no compelling reason to make this refactoring. The tests for CourseSession class could retain the contract for both CourseSession and SummerCourseSession. Minor differences do exist, though. The CourseSession class explicitly tracks the number of instances created; this is something the SummerCourseSession class need not do.

Building the Session superclass results in a slightly cleaner and easier to understand hierarchy. Conceptually, a CourseSession seems to be an object at the same hierarchical “level” as a SummerCourseSession.

It’s up to you to decide if such a refactoring is worthwhile. If it gives you a simpler solution, go for it. If it creates artificial classes in the hierarchy without any benefit, it’s a waste of time.

For the CourseSession example, you will refactor such that both CourseSession and SummerCourseSession inherit from a common abstract superclass, instead of SummerCourseSession inheriting from CourseSession. See Figure 6.4.

Create an abstract test class, SessionTest, that specifies the base unit tests for Session and its subclasses. Move testCreate, testComparable, and testEnrollStudents from CourseSessionTest into SessionTest. These tests represent the contracts that should apply to all Session subclasses.

The Principle of Subcontracting

SessionTest contains an abstract factory method, createSession. Subclasses of SessionTest will provide definitions for createSession that return an object of the appropriate type (CourseSession or SummerCourseSession).

```

package sis.studentinfo;

import junit.framework.TestCase;
import java.util.*;
import static sis.studentinfo.DateUtil.createDate;
  
```

⁵[George2002].

```

abstract public class SessionTest extends TestCase {
    private Session session;
    private Date startDate;
    public static final int CREDITS = 3;

    public void setUp() {
        startDate = createDate(2003, 1, 6);
        session = createSession("ENGL", "101", startDate);
        session.setNumberOfCredits(CREDITS);
    }

    abstract protected Session createSession(
        String department, String number, Date startDate);

    public void testCreate() {
        assertEquals("ENGL", session.getDepartment());
        assertEquals("101", session.getNumber());
        assertEquals(0, session.getNumberOfStudents());
        assertEquals(startDate, session.getStartDate());
    }

    public void testEnrollStudents() {
        Student student1 = new Student("Cain DiVoe");
        session.enroll(student1);
        assertEquals(CREDITS, student1.getCredits());
        assertEquals(1, session.getNumberOfStudents());
        assertEquals(student1, session.get(0));

        Student student2 = new Student("Coralee DeVaughn");
        session.enroll(student2);
        assertEquals(CREDITS, student2.getCredits());
        assertEquals(2, session.getNumberOfStudents());
        assertEquals(student1, session.get(0));
        assertEquals(student2, session.get(1));
    }

    public void testComparable() {
        final Date date = new Date();
        Session sessionA = createSession("CMSC", "101", date);
        Session sessionB = createSession("ENGL", "101", date);
        assertTrue(sessionA.compareTo(sessionB) < 0);
        assertTrue(sessionB.compareTo(sessionA) > 0);

        Session sessionC = createSession("CMSC", "101", date);
        assertEquals(0, sessionA.compareTo(sessionC));

        Session sessionD = createSession("CMSC", "210", date);
        assertTrue(sessionC.compareTo(sessionD) < 0);
        assertTrue(sessionD.compareTo(sessionC) > 0);
    }
}

```

The Principle of Subcontracting

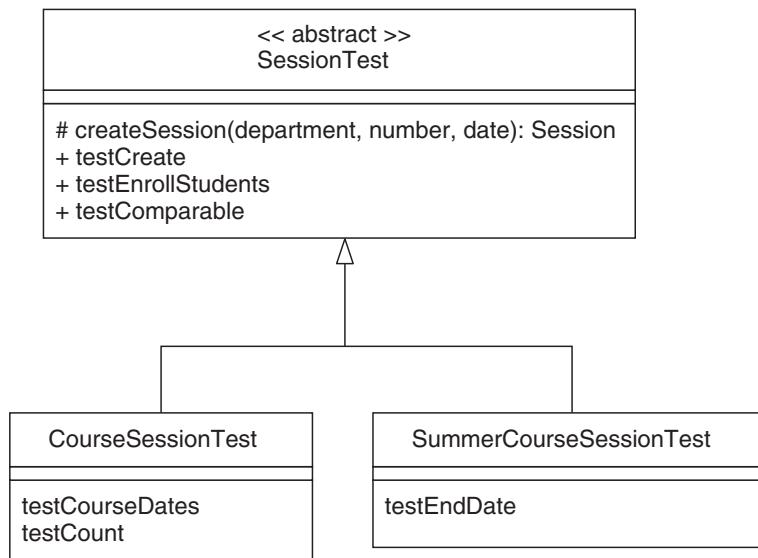


Figure 6.5 Abstract Test

You must change the test subclasses, `CourseSessionTest` and `SummerCourseSessionTest`, so they extend from `SessionTest`. As you work through the example, you may refer to Figure 6.5 as the UML basis for the test reorganization.

Each test subclass will also need to provide the implementation for `createSession`.

```

// SummerCourseSessionTest.java
package sis.summer;

import junit.framework.*;
import java.util.*;
import sis.studentinfo.*;

public class SummerCourseSessionTest extends SessionTest {
    public void testEndDate() {
        Date startDate = DateUtil.createDate(2003, 6, 9);
        Session session = createSession("ENGL", "200", startDate);
        Date eightWeeksOut = DateUtil.createDate(2003, 8, 1);
        assertEquals(eightWeeksOut, session.getEndDate());
    }

    protected Session createSession(
        String department,
        String number,
        Date date) {
  
```

The Principle of
Subcontracting

```
        return SummerCourseSession.create(department, number, date);
    }
}
```

In CourseSessionTest, you'll need to change any usages of the `createCourseSession` method to `createSession`.

```
// CourseSessionTest.java
package sis.studentinfo;

import junit.framework.TestCase;
import java.util.*;
import static sis.studentinfo.DateUtil.createDate;

public class CourseSessionTest extends SessionTest {
    public void testCourseDates() {
        Date startDate = DateUtil.createDate(2003, 1, 6);
        Session session = createSession("ENGL", "200", startDate);
        Date sixteenWeeksOut = createDate(2003, 4, 25);
        assertEquals(sixteenWeeksOut, session.getEndDate());
    }

    public void testCount() {
        CourseSession.resetCount();
        createSession("", "", new Date());
        assertEquals(1, CourseSession.getCount());
        createSession("", "", new Date());
        assertEquals(2, CourseSession.getCount());
    }

    protected Session createSession(
        String department,
        String number,
        Date date) {
        return CourseSession.create(department, number, date);
    }
}
```

The Principle of Subcontracting

The method `testCourseDates` creates its own local instance of a `CourseSession`. The alternative would be to use the `CourseSession` object created by the `setUp` method in `SessionTest`. Creating the `CourseSession` directly in the subclass `test` method makes it easier to read the test, however. It would be difficult to understand the meaning of the test were the instantiation in another class. The assertion is closely related to the setup of the `CourseSession`.

Both CourseSessionTest and SummerCourseSessionTest now provide an implementation for the abstract createSession method.

The most important thing to note is that CourseSessionTest and SummerCourseSessionTest share all tests defined in SessionTest, since both classes extend from SessionTest. JUnit executes each test defined in SessionTest

twice—once for a CourseSessionTest instance and once for a SummerCourseSessionTest instance. This ensures that both subclasses of Session behave appropriately.

When JUnit executes a method defined in SessionTest, the code results in a call to `createSession`. Since you defined `createSession` as abstract, there is no definition for the method in SessionTest. The Java VM calls the `createSession` method in the appropriate test subclass instead. Code in `createSession` creates an appropriate subtype of Session, either CourseSession or SummerCourseSession.

The refactoring of the production classes (I've removed the Java API documentation):

```
package sis.studentinfo;

import java.util.*;

abstract public class Session implements Comparable<Session> {
    private static int count;
    private String department;
    private String number;
    private List<Student> students = new ArrayList<Student>();
    private Date startDate;
    private int numberOfCredits;

    protected Session(
        String department, String number, Date startDate) {
        this.department = department;
        this.number = number;
        this.startDate = startDate;
    }

    public int compareTo(Session that) {
        int compare =
            this.getDepartment().compareTo(that.getDepartment());
        if (compare != 0)
            return compare;
        return this.getNumber().compareTo(that.getNumber());
    }

    void setNumberOfCredits(int numberOfCredits) {
        this.numberOfCredits = numberOfCredits;
    }

    public String getDepartment() {
        return department;
    }

    public String getNumber() {
        return number;
    }
}
```

The Principle of
Subcontracting

```

int getNumberOfStudents() {
    return students.size();
}

public void enroll(Student student) {
    student.addCredits(numberOfCredits);
    students.add(student);
}

Student get(int index) {
    return students.get(index);
}

protected Date getStartDate() {
    return startDate;
}

public List<Student> getAllStudents() {
    return students;
}

abstract protected int getSessionLength();

public Date getEndDate() {
    GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(getStartDate());
    final int daysInWeek = 7;
    final int daysFromFridayToMonday = 3;
    int numberofDays =
        getSessionLength() * daysInWeek - daysFromFridayToMonday;
    calendar.add(Calendar.DAY_OF_YEAR, numberofDays);
    return calendar.getTime();
}
}

```

The bulk of code in Session comes directly from CourseSession. The method getSessionLength becomes an abstract method, forcing subclasses to each implement it to provide the session length in weeks. Here are CourseSession and SummerCourseSession, each having been changed to inherit from Session:

```

// CourseSession.java
package sis.studentinfo;

import java.util.*;

public class CourseSession extends Session {
    private static int count;

```

The Principle of Subcontracting

```

public static CourseSession create(
    String department,
    String number,
    Date startDate) {
    return new CourseSession(department, number, startDate);
}

protected CourseSession(
    String department, String number, Date startDate) {
    super(department, number, startDate);
    CourseSession.incrementCount();
}

static private void incrementCount() {
    ++count;
}

static void resetCount() {
    count = 0;
}

static int getCount() {
    return count;
}

protected int getSessionLength() {
    return 16;
}
}

// SummerCourseSession.java
package sis.summer;

import java.util.*;
import sis.studentinfo.*;

public class SummerCourseSession extends Session {
    public static SummerCourseSession create(
        String department,
        String number,
        Date startDate) {
        return new SummerCourseSession(department, number, startDate);
    }

    private SummerCourseSession(
        String department,
        String number,
        Date startDate) {
        super(department, number, startDate);
    }
}

```

The Principle of
Subcontracting

```

protected int getSessionLength() {
    return 8;
}
}

```

All that is left in the subclasses are constructors, static methods, and template method overrides. Constructors are not inherited. Common methods, such as `enroll` and `getAllStudents`, now appear only in the `Session` superclass. All of the fields are common to both subclasses and thus now appear in `Session`. Also important is that the return type for the `create` method in `SummerCourseSession` and `CourseSession` is the abstract supertype `Session`.

The test method `testCourseDates` appears in both `SessionTest` subclasses. You could define a common assertion for this test in `SessionTest`, but what would the assertion be? One simple assertion would be that the end date must be later than the start date. Subclasses would strengthen that assertion by verifying against a specific date.

Another solution would be to assert that the end date is exactly n weeks, minus the appropriate number of weekend days, after the start date. The resultant code would have you duplicate the logic from `getEndDate` directly in the test itself.

Of perhaps more value is adding assertions against the method `getSessionLength` in the abstract test:

```

public void testSessionLength() {
    Session session = createSession(new Date());
    assertTrue(session.getSessionLength() > 0);
}

```

By expressing postconditions clearly in the abstract class, you are establishing a subcontract that all subclasses must adhere to.

This refactoring was a considerable amount of work. I hope you took the time to incrementally apply the changes.

Exercises

Exercises

1. In the exercises for Lesson 5, you created a method to calculate the strength of a piece based on its type and board position. Modify this method to use a `switch` statement instead of an `if` statement.
2. Modify the strength calculation method to use a `Map` that associates piece types with their base strength value. In order to place `double` values in `Map`, you will need to declare the map as `Map<Piece.Type,Double>`.⁶

⁶Lesson 7 includes a section on wrapper types that explains why this is so.

3. Use lazy initialization to preload the Map of piece type to base-strength value. Note the effect on readability this change has on the code. What factors would drive you to either leave this change in the code or to remove it?
4. Move the point values for the pieces into the piece type enumeration and use a method on the enumeration to get the point value.
5. Move the character representation for a piece onto the Piece.Type enum. Code in Board will still need to manage uppercasing this character for a black piece.
6. You must implement the various moves that a king can make. The basic rule is that a king can move one square in any direction. (For now, we will ignore the fact that neighboring pieces can either restrict such a move or can turn it into a capture.)
7. The code in Board is getting complex. It now manages the mathematics of a grid, it stores pieces, it determines valid moves for pieces (the queen and king, so far), it gathers pieces and determines their strength. By the time you add move logic for the remaining pieces (queen, pawn, rook, bishop, and knight), the class will be very cluttered with conditional logic.

There are a few ways to break things up. From my viewpoint, there are two main functions of code in Board class. One goal is to implement the 8x8 board using a data structure—a list of lists of pieces. A lot of the code in Board is geared toward navigating this data structure, without respect to the rules of chess. The rest of the code in Board defines the rules of the chess game.

Break the Board class into two classes: Game, which represents the logic of a chess game, and Board, a simple data structure that stores pieces and understands the grid layout of the board.

This will be a significant refactoring. Move methods as incrementally as possible, making sure tests remain green with each move. If necessary, create parallel code as you refactor, then eliminate the unnecessary code once everything is working. Make sure you add tests as needed.

Take this opportunity to begin to encapsulate the implementation details of the data structure of Board. Instead of asking for ranks and adding pieces, have the Game code use the Board method `put`.

8. A queen can move any number of squares as long as it forms a straight line. Implement this capability on Piece. Note that your initial implementation will require an if statement to determine whether the piece is a queen or a king.

The solution may involve *recursion*—the ability of a method to call itself. A recursive call is no big deal; it is a method call like any other. But you must be careful to provide a way for the recursion to stop,

Exercises

otherwise you might get stuck in an infinite loop of the method calling itself.

9. Your code to manage moves for kings and queens includes an if statement. You can imagine that supporting further moves for pieces will involve a bunch of conditional statements. One way of cleaning things up is to create a Piece hierarchy, with separate subtypes for King, Queen, Bishop, and so on.

In preparation for creating Piece subclasses, move the method `getPossibleMoves` to the Piece class.

10. Now you are ready to create Piece subclasses. You will note that as you create subclasses for Queen and King, the `Piece.Type` enum seems redundant. In the next exercise, you'll eliminate this enum. Create Queen and King classes to extend Piece. Move the relevant tests from `PieceTest` into `QueenTest` and `KingTest`. This will force you to move code as well into the new subclasses. Also, create subclasses for the remaining pieces and alter the factory methods accordingly on Piece. Don't worry about coding up the move rules for the other pieces yet.

Refactor at will. Eliminate duplication. Add tests where tests were missing due to moving methods onto a new class.

11. The final step is to eliminate the need for the Type enum. Since you now have a subclass for each piece, you can move code depending on the type into the subclass.

You may need to test the type of the subclass created. Instead of asking each piece for its `Piece.Type` value, you can simply ask for its class:

```
piece.getClass()
```

You can then compare this to a Class literal:

```
Class expectedClass = Queen.class;
assertEquals(expectedClass, piece.getClass());
```

You also want to eliminate as much code as possible that asks for the type of an object—that's one of the main reasons for creating the subclasses in the first place. See if you can move any code that looks like `if (piece.getClass().equals(Pawn.class))` into the appropriate class.

Exercises

Lesson 7

Legacy Elements

In this lesson you will learn about legacy elements of Java. Java is a continually evolving language. The original release of Java was immature with respect to both its language features and its class library. Over the years, Sun has added to the language and class library in attempts to provide a more robust platform. Sun has removed little. The result is that Java contains many features that Sun recommends you no longer use.

J2SE 5.0 introduces many new language elements, some that aspire to replace existing Java elements. As the most relevant example, the designers of Java changed the means of iterating through a collection in 5.0. Previously, the language supported iteration through a combination of the class library and procedural looping constructs. Now, the Java language directly supports iteration with the `for-each` loop.

In this lesson, you will learn about many Java elements that come from its syntactical grandparent, the C language. These elements are largely procedural in nature but are still necessary in order to provide a fully functional programming language. Nonetheless, most of the time you should be able to use constructs that are more object-oriented instead of the legacy elements in this lesson.

While I downplay the use of these legacy elements, you *must* understand them in order to fully master Java. They remain the underpinnings of the Java language. In many circumstances, you will be forced to use them. And in some cases, they still provide the best solution to the problem at hand.

Some of the legacy elements you will learn about include:

- `for`, `while`, and `do` loops
- looping control statements
- legacy collections
- iterators
- casting

Legacy
Elements

- wrapper classes
- arrays
- varargs

Each of these elements, with the exception of varargs, has been around since the advent of Java. The concept of collections, Iterators, and wrapper classes are object-oriented constructs. Everything else derives from the C language. The switch statement that you learned in Lesson 6 is also a legacy element that was derived from C.

Looping Constructs

The `for` loop as you have learned it provides a means of iterating through a collection and operating on each element.

You will need more general-purpose looping. Perhaps you need to execute a body of code infinitely, loop until some condition is met, or send a message ten times. Java meets these needs with three looping constructs: `while`, `do`, and `for`.

Breaking Up a Student's Name

 Currently, the `Student` class supports constructing a student with a single string that represents the student's name. You need to modify the `Student` class so that it can break apart a single string into its constituent name parts. For example, the `Student` class should be able to break the name string "Robert Cecil Martin" into a first name "Robert," a middle name "Cecil," and a last name "Martin."

The class should also assume that a string with two name parts represents a first and last name and that a string with one name part represents a last name only.

First, modify the existing `testCreate` method defined in `StudentTest`:

```
public void testCreate() {
    final String firstStudentName = "Jane Doe";
    Student firstStudent = new Student(firstStudentName);
    assertEquals(firstStudentName, firstStudent.getName());
    assertEquals("Jane", firstStudent.getFirstName());
    assertEquals("Doe", firstStudent.getLastName());
    assertEquals("", firstStudent.getMiddleName());
```

Breaking Up a Student's Name

```

final String secondStudentName = "Blow";
Student secondStudent = new Student(secondStudentName);
assertEquals(secondStudentName, secondStudent.getName());
assertEquals("", secondStudent.getFirstName());
assertEquals("Blow", secondStudent.getLastName());
assertEquals("", secondStudent.getMiddleName());

final String thirdStudentName = "Raymond Douglas Davies";
Student thirdStudent = new Student(thirdStudentName);
assertEquals(thirdStudentName, thirdStudent.getName());
assertEquals("Raymond", thirdStudent.getFirstName());
assertEquals("Davies", thirdStudent.getLastName());
assertEquals("Douglas", thirdStudent.getMiddleName());
}

```

You'll need to add fields and create getter methods in Student:

```

private String firstName;
private String middleName;
private String lastName;

...
public String getFirstName() {
    return firstName;
}

public String getMiddleName() {
    return middleName;
}

public String getLastName() {
    return lastName;
}

```

The modified Student constructor code shows your *intent* of how to solve the problem:

```

public Student(String fullName) {
    this.name = fullName;
    credits = 0;
    List<String> nameParts = split(fullName);
    setName(nameParts);
}

```

Programming by intention is a useful technique that helps you figure out *what* you need to do before you figure out *how* you are going to do it.¹ It can be viewed as decomposing the problem into smaller abstractions. You can put these abstractions into code before actually implementing the abstractions. Here, you know that you need to split the full name into different parts, but you don't yet know how.

**Breaking Up a
Student's Name**

¹[Astels2003], p. 45.



Use programming by intention to break down a problem into a number of smaller goals.

Once you have figured out how to split the string into a list of up to three name parts, the implementation of `setName` is straightforward. It involves a few `if-else` statements. Each `if` conditional tests the number of name parts available in the list of tokens:

```
private void setName(List<String> nameParts) {
    if (nameParts.size() == 1)
        this.lastName = nameParts.get(0);
    else if (nameParts.size() == 2) {
        this.firstName = nameParts.get(0);
        this.lastName = nameParts.get(1);
    }
    else if (nameParts.size() == 3) {
        this.firstName = nameParts.get(0);
        this.middleName = nameParts.get(1);
        this.lastName = nameParts.get(2);
    }
}
```

The `setName` code always sets a value into `lastName`. However, it does not always set the value of `firstName` or `middleName`. You will need to ensure that these fields are properly initialized in `Student`:

```
private String firstName = "";
private String middleName = "";
private String lastName;
```

The `while` Loop

As usual, Java gives you several ways to solve the problem of splitting the name. You'll start with a more "classic," more tedious approach. You will loop through each character in the name, looking for space characters, using them to determine where one name part ends and the next name part begins. Later you'll learn a couple of simpler, more-object-oriented solutions.

The `split` method needs to take the full name as an argument and return a list of tokens as a result. You will use a `while` loop to derive each token in turn from the string and add it to the result list.

Breaking Up a Student's Name

```
private List<String> tokenize(String string) {
    List<String> results = new ArrayList<String>();

    StringBuffer word = new StringBuffer();
    int index = 0;
    while (index < string.length()) {
```

```

char ch = string.charAt(index);
if (ch != ' ') // prefer Character.isSpace. Defined yet?
    word.append(ch);
else
    if (word.length() > 0) {
        results.add(word.toString());
        word = new StringBuffer();
    }
    index++;
}
if (word.length() > 0)
    results.add(word.toString());
return results;
}

```

You use the `while` loop to execute a statement or block of code as long as a condition holds true.

In this example, after Java initializes an index counter to 0, it executes the `while` loop. As long as the index is less than the string length (`index < string.length()`), Java executes the body of the `while` loop (all code between the braces following the `while` conditional).

The body in this example extracts the character at the current index from the string. It tests whether the character is a space (' ') or not. If not, the character is appended to the current `StringBuffer` instance. If so, and if there are any characters in the current `StringBuffer`, the contents of the `StringBuffer` represent a complete word and are added to the results. A new `StringBuffer`, to represent a new word, is created and assigned to the `word` reference.

Each time the body of the `while` loop completes, control is transferred back up to the `while` loop conditional. Once the conditional returns a `false` result, the `while` loop terminates. Control is then transferred to the statement immediately following the `while` loop body. In the example, the statement immediately following is an `if` statement that ensures that the last word extracted becomes part of the results.

The modified test should pass.

Refactoring

The `setName` method is a bit repetitive. You can refactor it by taking advantage of having the name parts in a list that you can manipulate.

Breaking Up a Student's Name

```

private void setName(List<String> nameParts) {
    this.lastName = removeLast(nameParts);
    String name = removeLast(nameParts);
    if (nameParts.isEmpty())
        this.firstName = name;
    else {

```

```
        this.middleName = name;
        this.firstName = removeLast(nameParts);
    }

private String removeLast(List<String> list) {
    if (list.isEmpty())
        return "";
    return list.remove(list.size() - 1);
}
```

You actually create more lines of code overall with this solution. But you eliminate the duplication and hard-coded indexes rampant in `setName`. Further, you produce a generic method, `removeLast`, that later may be a useful abstraction.

The for Loop

Looping using a counter and a limit, as in the previous example, is a common operation. You can more succinctly express this operation by using a variant of the `for` loop. The `for` loop allows you to combine three common loop parts into one declaration. Each `for` loop declaration contains:

- initialization code
 - a conditional expression representing whether or not to continue execution of the loop
 - an update expression that the Java VM executes each time the loop terminates

You separate each of these three sections in the for loop declaration using a semicolon.

The classic use of a `for` loop is to execute a body of code a certain number of times. Typically, but not necessarily, the initialization code sets an index to 0, the conditional tests that the index is less than a limit, and the update expression increments the counter by one.

Breaking Up a Student's Name

```
private List<String> split(String name) {  
    List<String> results = new ArrayList<String>();  
  
    StringBuffer word = new StringBuffer();  
    for (int index = 0; index < name.length(); index++) {  
        char ch = name.charAt(index);  
        if (!Character.isWhitespace(ch))  
            word.append(ch);  
        else if (word.length() > 0)  
            results.add(word.toString());  
        word.setLength(0);  
    }  
    if (word.length() > 0)  
        results.add(word.toString());  
    return results;  
}
```

```

        else
            if (word.length() > 0) {
                results.add(word.toString());
                word = new StringBuffer();
            }
    }
    if (word.length() > 0)
        results.add(word.toString());
    return results;
}

```

In the `for` loop example, the initialization code declares `index` as an `int` and initializes it to 0. The conditional tests that `index` is less than the number of characters in the `String` argument. As long as the condition holds true, Java executes the body of the loop. Subsequent to each time Java executes the loop, Java increments `index` (`index++`); it then transfers control back to the conditional. Once the conditional returns `false`, Java transfers control to the statement following the `for` loop body.

In the example, Java executes the body of the `for` loop once for each character in the input string. The `index` value ranges from zero up to the number of characters in the input string minus one.

Multiple Initialization and Update Expressions in the `for` Loop

Both the initialization and update code sections allow for multiple expressions separated by commas. The `countChars` method,² which returns the number of occurrences of a specified character within a string, shows how you can initialize both the variables `i` and `count` to 0.

```

public static int countChars(String input, char ch) {
    int count;
    int i;
    for (i = 0, count = 0; i < input.length(); i++)
        if (input.charAt(i) == ch)
            count++;
    return count;
}

```

The use of the letter `i` as a `for` loop index variable is an extremely common idiom. It is one of the few commonly accepted standards that allows you to abbreviate a variable name. Most developers prefer to use `i` over a full word such as `index`.

Breaking Up a Student's Name

²This method and several other methods in this lesson such as the upcoming method `isPalindrome`, have nothing to do with the student information system. They are here to help demonstrate some of the lesser-seen nuances of Java syntax.

The following method, `isPalindrome`, returns true if a string reads backward the same as forward. It shows how you can declare and initialize more than one variable in the initialization code. The method also shows multiple expressions—incrementing forward and decrementing backward—in the update step.

```
public static boolean isPalindrome(String string) {
    for (int forward = 0, backward = string.length() - 1;
        forward < string.length();
        forward++, backward--)
        if (string.charAt(forward) != string.charAt(backward))
            return false;
    return true;
}

// tests
public void testPalindrome() {
    assertFalse(isPalindrome("abcdef"));
    assertFalse(isPalindrome("abccda"));
    assertTrue(isPalindrome("abccba"));
    assertFalse(isPalindrome("abcxba"));
    assertTrue(isPalindrome("a"));
    assertTrue(isPalindrome("aa"));
    assertFalse(isPalindrome("ab"));
    assertTrue(isPalindrome(""));
    assertTrue(isPalindrome("aaa"));
    assertTrue(isPalindrome("aba"));
    assertTrue(isPalindrome("abbba"));
    assertTrue(isPalindrome("abba"));
    assertFalse(isPalindrome("abbaa"));
    assertFalse(isPalindrome("abcd"));
}
```

A local variable that you declare in the initialization code of a `for` loop is valid only for the scope of the `for` loop. Thus, in the `countChars` method, you must declare `count` prior to the loop, since you return `count` *after* the `for` loop.

Java stipulates that “if you have a local variable declaration, each part of the expression after a comma is expected to be a part of that local variable declaration.”³ So while it would be nice to be able to recode `countChars` to declare `i` and only initialize `count`, Java expects the following initialization code to declare both `i` *and* `count`.

```
// this code will not compile!
public static int countChars(String input, char ch) {
    int count;
    for (int i = 0, count = 0; i < input.length(); i++)
        if (input.charAt(i) == ch)
            count++;
    return count;
}
```

³[Arnold2000].

Breaking Up a Student’s Name

But since you already declared `count` before the `for` loop, the Java compiler rejects the attempt:

```
count is already defined in countChars(java.lang.String,char)
```

All three parts of the `for` loop declaration—initialization, conditional, and update expression—are optional. If you omit the conditional, Java executes the loop infinitely. The following `for` statement is a common idiom for an infinite loop:

```
for (;;) {  
}
```

A more expressive way to create an infinite loop is to use a `while` loop:

```
while (true) {  
}
```

As mentioned, the most common use for `for` loops is to count from 0 up to $n - 1$, where n is the number of elements in a collection. Nothing prohibits you from creating `for` loops that work differently. The `isPalindrome` method shows how you can count upward at the same time as counting downward.

Another language test shows how you can have the update expression do something other than decrement or increment the index:

```
public void testForSkip() {  
    StringBuilder builder = new StringBuilder();  
    String string = "123456";  
    for (int i = 0; i < string.length(); i += 2)  
        builder.append(string.charAt(i));  
    assertEquals("135", builder.toString());  
}
```

The `do` Loop

The `do` loop works like a `while` loop, except that the conditional to be tested appears at the *end* of the loop body. You use a `do` loop to ensure that the you execute the body of a loop at least once.

In the thirteenth century, Leonardo Fibonacci came up with a numeric sequence to represent how rabbits multiply.⁴ The Fibonacci sequence starts with 0 and 1. Each subsequent number in the sequence is the sum of the two preceding numbers in the sequence. The test and code below demonstrate an implementation of the Fibonacci function using a `do` loop.

Breaking Up a Student's Name

⁴[Wikipedia2004]

```

public void testFibonacci() {
    assertEquals(0, fib(0));
    assertEquals(1, fib(1));
    assertEquals(1, fib(2));
    assertEquals(2, fib(3));
    assertEquals(3, fib(4));
    assertEquals(5, fib(5));
    assertEquals(8, fib(6));
    assertEquals(13, fib(7));
    assertEquals(21, fib(8));
    assertEquals(34, fib(9));
    assertEquals(55, fib(10));
}

private int fib(int x) {
    if (x == 0) return 0;
    if (x == 1) return 1;
    int fib = 0;
    int nextFib = 1;
    int index = 0;
    int temp;
    do {
        temp = fib + nextFib;
        fib = nextFib;
        nextFib = temp;
    } while (++index < x);
    return fib;
}

```

Comparing Java Loops

A way exists to code any looping need using any one of the three loop variants. The following test and three methods show three techniques for displaying a list of numbers separated by commas.

```

public void testCommas() {
    String sequence = "1,2,3,4,5";
    assertEquals(sequence, sequenceUsingDo(1, 5));
    assertEquals(sequence, sequenceUsingFor(1, 5));
    assertEquals(sequence, sequenceUsingWhile(1, 5));

    sequence = "8";
    assertEquals(sequence, sequenceUsingDo(8, 8));
    assertEquals(sequence, sequenceUsingFor(8, 8));
    assertEquals(sequence, sequenceUsingWhile(8, 8));
}

String sequenceUsingDo(int start, int stop) {
    StringBuilder builder = new StringBuilder();
    int i = start;

```

Comparing Java Loops

```

do {
    if (i > start)
        builder.append(',');
    builder.append(i);
} while (++i <= stop);
return builder.toString();
}

String sequenceUsingFor(int start, int stop) {
    StringBuilder builder = new StringBuilder();
    for (int i = start; i <= stop; i++) {
        if (i > start)
            builder.append(',');
        builder.append(i);
    }
    return builder.toString();
}

String sequenceUsingWhile(int start, int stop) {
    StringBuilder builder = new StringBuilder();
    int i = start;
    while (i <= stop) {
        if (i > start)
            builder.append(',');
        builder.append(i);
        i++;
    }
    return builder.toString();
}

```

As this example shows, you will find that one of the loop variants is usually more appropriate than the other two. Here, a `for` loop is best suited since the loop is centered around counting needs.

If you have no need to maintain a counter or other incrementing variable, the `while` loop will usually suffice. Most of the time, you want to test a condition each and every time upon entry to a loop. You will use the `do` loop far less frequently. The need to always execute the loop at least once before testing the condition is less common.

Refactoring

Refactoring

isPalindrome

The `isPalindrome` method expends almost twice as much effort as necessary to determine whether a string is a palindrome. While normally you shouldn't worry about performance until it is a problem, an algorithm should be clean.

Doing something unnecessarily in an algorithm demonstrates poor understanding of the problem. It can also confuse future developers.

You only need to traverse the string up to its middle. The current implementation traverses the string for its entire length, which means that all or almost all characters are compared twice. A better implementation:

```
public static boolean isPalindrome(String string) {
    if (string.length() == 0)
        return true;
    int limit = string.length() / 2;
    for
        (int forward = 0, backward = string.length() - 1;
         forward < limit;
         forward++, backward--)
        if (string.charAt(forward) != string.charAt(backward))
            return false;
    return true;
}
```

Fibonacci and Recursion

Java supports *recursive* method calls. A recursive method is one that calls itself. A more elegant solution for the Fibonacci sequence uses recursion:

```
private int fib(int x) {
    if (x == 0)
        return 0;
    if (x == 1)
        return 1;
    return fib(x - 1) + fib(x - 2);
}
```

Be cautious when using recursion! Make sure you have a way of “breaking” the recursion, otherwise your method will recurse infinitely. You can often use a guard clause for this purpose, as in the `fib` method (which has two guard clauses).

Looping Control Statements

Looping Control Statements

Java provides additional means for altering control flow within a loop: the `continue` statement, the `break` statement, the labeled `break` statement, and the labeled `continue` statement.

The **break** Statement

You may want to prematurely terminate the execution of a loop. If the Java VM encounters a `break` statement within the body of a loop, it immediately transfers control to the statement following the loop body.

The `String` class defines a `trim` method that removes blank characters⁵ from *both ends* of a `String`. You will define another `String` utility that trims blank characters from only the *end* of a `String`.

```
public void testEndTrim() {
    assertEquals("", endTrim(""));
    assertEquals(" x", endTrim(" x "));
    assertEquals("y", endTrim("y"));
    assertEquals("xaxa", endTrim("xaxa"));
    assertEquals("", endTrim(" "));
    assertEquals("xxx", endTrim("xxx      "));
}

public String endTrim(String source) {
    int i = source.length();
    while (-i >= 0)
        if (source.charAt(i) != ' ')
            break;
    return source.substring(0, i + 1);
}
```

The **continue** Statement

When it encounters a `continue` statement, Java immediately transfers control to the conditional expression of a loop.



The example calculates the average GPA for part-time students enrolled in a course session. The test, as implemented in `SessionTest`:

```
public void testAverageGpaForPartTimeStudents() {
    session.enroll(createFullTimeStudent());

    Student partTimer1 = new Student("1");
    partTimer1.addGrade(Student.Grade.A);
    session.enroll(partTimer1);

    session.enroll(createFullTimeStudent());

    Student partTimer2 = new Student("2");
    partTimer2.addGrade(Student.Grade.B);
    session.enroll(partTimer2);
```

Looping Control Statements

⁵It actually removes *whitespace*, which includes space characters as well as tab, new line, form feed, and carriage return characters.

```

        assertEquals(3.5, session.averageGpaForPartTimeStudents(), 0.05);
    }

private Student createFullTimeStudent() {
    Student student = new Student("a");
    student.addCredits(Student.CREDITS_REQUIRED_FOR_FULL_TIME);
    return student;
}

```

The implementation (in Session) demonstrates use of the `continue` statement to skip students that are not part-time.

```

double averageGpaForPartTimeStudents() {
    double total = 0.0;
    int count = 0;
    for (Student student: students) {
        if (student.isFullTime())
            continue;
        count++;
        total += student.getGpa();
    }
    if (count == 0) return 0.0;
    return total / count;
}

```

You can easily rewrite this (and most) uses of `continue` using `if-else` statements. Sometimes the use of `continue` provides the more-elegant, easy-to-read presentation.

Labeled `break` and `continue` Statements

The labeled `break` and `continue` statements allow you to transfer control when you have more complex nesting.

The first example tests a labeled `break`. Don't let the interesting declaration of `table` scare you. `List<List<Integer>>` declares a list into which you can put lists bound to the `Integer` type.

```

public void testLabeledBreak() {
    List<List<String>> table = new ArrayList<List<String>>();

    List<String> row1 = new ArrayList<String>();
    row1.add("5");
    row1.add("2");
    List<String> row2 = new ArrayList<String>();
    row2.add("3");
    row2.add("4");

    table.add(row1);
    table.add(row2);
    assertTrue(found(table, "3"));
}

```

```

        assertFalse(found(table, "8"));
    }

private boolean found(List<List<String>> table, String target) {
    boolean found = false;
    search:
    for (List<String> row: table) {
        for (String value: row) {
            if (value.equals(target)) {
                found = true;
                break search;
            }
        }
    }
    return found;
}

```

If you did not associate the `search` label with the `break` statement in this example, Java would break out of only the innermost `for` loop (the one with the expression `Integer num: row`). Instead, the `break` statement causes looping to terminate from the `search` label, representing the outermost loop in this example.

The labeled `continue` statement works similarly, except the Java VM transfers control to the loop with the label instead of breaking from the loop with the label.

There are few situations where your solution will require labeled `break` or `continue` statements. In most cases, you can easily restructure code to eliminate them, either by use of `if-else` statements or, even better, by decomposition of methods into smaller, more composed methods. Use the labeled statements only when the solution is easier to comprehend.

The Ternary Operator

Lesson 5 introduced the `if` statement. Often you want to assign a value to a variable or return a value from a method based on the results of an `if` conditional. For example:

```

String message =
    "the course has " + getText(sessions) + " sessions";
...
private String getText(int sessions) {
    if (sessions == 1)
        return "one";
    return "many";
}

```

The Ternary Operator

For simple conditional expressions that must return a new value, Java provides a shortcut form known as the *ternary operator*. The ternary operator compacts the if-else statement into a single expression. The general form of the ternary operator is:

```
conditional ? true-value : false-value
```

If conditional returns the value `true`, the result of the entire expression is `true-value`. Otherwise the result of the entire expression is `false-value`. Using the ternary operator, you can rewrite the previous code to produce a message as:

```
String message =
    "the course has " + (sessions == 1 ? "one" : "many") + " sessions";
```

If the value of `sessions` is 1, the parenthesized ternary expression returns the String `"one"`, otherwise it returns `"many"`. This result string is then used as part of the larger string concatenation.

The ternary operator is a holdover from the language C. Do not use the ternary operator as a general replacement for the `if` statement! The best use of the ternary operator is for simple, single-line expressions like the one shown here. If you have more complex needs, or if the code won't fit on a single line, you're better off using an `if` statement. Abuse of the ternary operator can result in cryptic code that is more costly to maintain.

Legacy Collections

Prior to the SDK version 1.2, there was no collections “framework” in Java. Java contained two main workhorse collections—`java.util.Vector` and `java.util.Hashtable`⁶—that still exist today. The analogous classes in modern Java are respectively `java.util.ArrayList` and `java.util.HashMap`. The class `java.util.HashMap` provides similar functionality as the class `java.util.EnumMap`, with which you are already familiar. Refer to Lesson 9 for more information on `HashMap`.

Legacy
Collections

Both `Vector` and `Hashtable` are concrete; they implement no common interfaces. They incorporate support for multithreaded programming⁷ by default, a choice that often results in unnecessary overhead. Unfortunately, today you will still encounter a large amount of code that uses or requires use of the `Vector` class.

⁶There were only two other collection classes: `BitSet` and `Stack` (a subclass of `Vector`). Both were minimally useful.

⁷See Lesson 13 for a discussion of multithreaded programming.

Since there were no corresponding interfaces for these classes in the initial versions of Java, you were forced to assign Vector or Hashtable instances to a reference to an implementation class:

```
Vector names = new Vector();
```

With the introduction of the collections framework, Sun retrofitted the Vector class to implement the List interface and the Hashtable class to implement the Map interface. You might for some reason be stuck with continuing to use a Vector or Hashtable. If so, you may be able to migrate to a modern approach by assigning the instance to an interface reference:

```
List names = new Vector();
Map dictionary = new Hashtable();
```

Sun also retrofitted Vector and Hashtable so that they are parameterized types. You can thus code:

```
List<String> names = new Vector<String>();
Map<Student.Grade, String> dictionary =
    new Hashtable<Student.Grade, String>();
```

Iterators

You have learned to use the `for-each` loop to iterate through each element in a collection. This form of the `for` loop is a new construct, introduced in J2SE 5.0.

Prior to the `for-each` loop, the preferred way to iterate through a collection was to ask it for a `java.util.Iterator` object. An Iterator object maintains an internal pointer to an element in the collection. You can ask an iterator to return the next available element in the collection. After returning an element from the collection, an Iterator advances its internal pointer to refer to the next available element. You can also ask an iterator whether or not there are more elements in a collection.

As an example, rewrite the `averageGpaForPartTimeStudents` method shown earlier in this lesson to use an Iterator object and a `for` loop instead of a `for-each` loop:

```
double averageGpaForPartTimeStudents() {
    double total = 0.0;
    int count = 0;

    for (Iterator<Student> it = students.iterator();
         it.hasNext(); ) {
        Student student = it.next();
```

Iterators

```

        if (student.isFullTime())
            continue;
        count++;
        total += student.getGpa();
    }
    if (count == 0) return 0.0;
    return total / count;
}

```

You usually obtain an Iterator by sending the message `iterator` to the collection. You assign the Iterator object to an Iterator reference that you bind to the type of object stored within the collection. Here, you bind the Iterator to the type `Student`, which is what the `students` collection stores.

The Iterator interface defines three methods: `hasNext`, `next`, and `remove` (which is optional and infrequently needed). The `hasNext` method returns `true` if there are any elements remaining in the collection that have not yet been returned. The `next` method returns the next available element from the collection and advances the internal pointer.

The legacy collections `Vector` and `Hashtable` use an analogous technique known as enumeration. The solution is virtually the same, only the names are different. Redefine the `students` instance variable in `Session` to be a `Vector`.

```

package sis.studentinfo;

import java.util.*;

abstract public class Session
    implements Comparable<Session> {
    ...
    private Vector<Student> students = new Vector<Student>();

    double averageGpaForPartTimeStudents() {
        double total = 0.0;
        int count = 0;

        for (Enumeration<Student> it = students.elements();
            it.hasMoreElements(); ) {
            Student student = it.nextElement();
            if (student.isFullTime())
                continue;
            count++;
            total += student.getGpa();
        }
        if (count == 0) return 0.0;
        return total / count;
    }
}

```

Iterators

Instead of asking for an Iterator using the iterator method, you ask for an Enumeration by using the elements method. You test whether there are more elements available using hasMoreElements instead of hasNext. You retrieve the next element using nextElement instead of next. Sun introduced the Iterator in response to many complaints about the unnecessary verbosity of the class name Enumeration and its method names.

Since Vector implements the List interface, you can send a Vector the iterator message to obtain an Iterator instead of an Enumeration.

Revert your Session code to use modern collections and the for-each loop.

Iterators and the for-each Loop

The for-each loop construct is built upon the Iterator construct. In order to be able to loop through a collection, the collection must be able to return an Iterator object when sent the iterator message. The for-each loop recognizes whether a collection is iterable or not by seeing if the collection implements the java.lang.Iterable interface.

In this upcoming example, you will modify the Session class to support iterating through its collection of students.

Code the following test in SessionTest:

```
public void testIterate() {
    enrollStudents(session);

    List<Student> results = new ArrayList<Student>();
    for (Student student: session)
        results.add(student);

    assertEquals(session.getAllStudents(), results);
}

private void enrollStudents(Session session) {
    session.enroll(new Student("1"));
    session.enroll(new Student("2"));
    session.enroll(new Student("3"));
}
```

**Iterators and the
for-each Loop**

When you compile, you will receive an error, since Session does not yet support the ability to be iterated over.

```
foreach not applicable to expression type
    for (Student student: session)
        ^
```

You must change the Session class to implement the Iterable interface. Since you want the for-each loop to recognize that it is iterating over a collection containing only Student types, you must bind the Iterable interface to the Student type.

```
abstract public class Session
    implements Comparable<Session>, Iterable<Student> {
// ...
```

Now you only need to implement the iterator method in Session. Since Session merely encapsulates an ArrayList of students, there is no reason that the iterator method can't simply return the ArrayList's Iterator object.

```
public Iterator<Student> iterator() {
    return students.iterator();
}
```

Casting

The need for *casting* of references occurs when *you*, the programmer, know that an object is of a certain type but the compiler couldn't possibly know what the type is. Casting allows you to tell the compiler that it can safely assign an object of one type to a reference of a different type.

Sun's introduction of parameterized types removed much of the need for casting in Java. Prior to J2SE 5.0, since the collection classes in Java did not support parameterized types, everything that went into a collection had to go in as an Object reference, as shown by the signature for the add method in the List interface:

```
public boolean add(Object o)
```

Since everything went *in* as an Object, you could only retrieve it as an Object:

```
public Object get(int index)
```

Casting

You, the developer, knew you were stuffing Student objects into a collection, but the collection could only store Object references to the Students. When you went to retrieve the objects from the collection, you knew that the collection stored Student objects. This meant that you typically cast the Object references back to Student references as you retrieved them from the collection.

To cast, precede a target expression with the reference type that you want to cast it to, in parentheses:

```
Student student = (Student)students.get(0);
```

In this example, the target is the result of the entire expression `students.get(0)`. You could have parenthesized the target to be more explicit:

```
(Student)(students.get(0));
```

Note the absence of any spaces between the cast and the target. While you may interject spaces between a cast and its target, the more commonly accepted style specifies no spaces.

It is important to understand that casting does nothing to change the target object. The cast to Student type only creates a new reference to the same memory location. Since this new reference is of type Student, the Java compiler will then allow you to send Student-specific messages to the object via the reference. Without casting, you would only be able to send messages defined in Object to the Student object.

Using parameterized types, there are few reasons to cast object references. If you find yourself casting, determine if there is a way to avoid casting by using parameterized types.

As an exercise, the following test demonstrates the “old way” of iterating through a collection—without a `for-each` loop and without parameterized types. Note that this code will still work under J2SE 5.0, although you may receive warnings because you are not using the parameterized collection types.

```
public void testCasting() {  
    List students = new ArrayList();  
    students.add(new Student("a"));  
    students.add(new Student("b"));  
  
    List names = new ArrayList();  
  
    Iterator it = students.iterator();  
    while (it.hasNext()) {  
        Student student = (Student)it.next();  
        names.add(student.getLastName());  
    }  
  
    assertEquals("a", names.get(0));  
    assertEquals("b", names.get(1));  
}
```

Casting

Casting Primitive Types

You cannot cast primitive types to reference types or vice versa.

There is another form of casting that you will still need to use in order to convert numeric types. I will discuss numeric conversion in Lesson 10 on mathematics.

Wrapper Classes



The student system must store student charges in a collection to be totaled later. Each charge is an `int` that represents the number of cents a student spent on an item.

Remember that primitive types are not objects—they do not inherit from the class `java.lang.Object`. The `java.util.List` interface only supplies an `add` method that takes a reference type as a argument. It does not supply overloaded `add` methods for each of the primitive types.

In order to store the `int` charge in a collection, then, you must convert it to an object. You accomplish this by storing the `int` in a *wrapper* object.

For each primitive type, `java.lang` defines a corresponding wrapper class.⁸

Type	Wrapper Class
<code>char</code>	<code>Character</code>
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>boolean</code>	<code>Boolean</code>

Each wrapper class provides a constructor that takes a primitive of the appropriate type as argument. The wrapper stores this primitive and allows for extracting it using a corresponding getter method. In the case of the `Integer` wrapper, you extract the original `int` by sending the message `intValue` to the wrapper.

A test (in `StudentTest`) for the example:

```
Wrapper  
Classes
public void testCharges() {
    Student student = new Student("a");
    student.addCharge(500);
    student.addCharge(200);
    student.addCharge(399);
    assertEquals(1099, student.totalCharges());
}
```

⁸The table lists all available primitive types and corresponding wrappers. You will learn more about the unfamiliar types in Lesson 10.

If you had to write the implementation using pre-J2SE 5.0 code, it would look something like this:

```
public class Student implements Comparable {
    ...
    private List charges = new ArrayList();
    ...
    public void addCharge(int charge) {
        charges.add(new Integer(charge));
    }

    public int totalCharges() {
        int total = 0;
        Iterator it = charges.iterator();
        while (it.hasNext()) {
            Integer charge = (Integer)it.next();
            total += charge.intValue();
        }
        return total;
    }
    ...
}
```

The `addCharge` method shows how you would wrap the `int` in an instance of the `Integer` wrapper class in order to pass it to the `add` method of the `charges` collection. When iterating through the collection (in `totalCharges`), you would have to cast each retrieved `Object` to the `Integer` wrapper class. Only then would you be able to extract the original `int` value through use of the `intValue` method.

J2SE 5.0 simplifies the code through the use of parameterized types and the for-each loop.

```
public class Student implements Comparable<Student> {
    ...
    private List<Integer> charges = new ArrayList<Integer>();
    ...
    public void addCharge(int charge) {
        charges.add(new Integer(charge));
    }

    public int totalCharges() {
        int total = 0;
        for (Integer charge: charges)
            total += charge.intValue();
        return total;
    }
    ...
}
```

Wrapper Classes

In addition to allowing you to wrap a primitive, the wrapper classes provide many class methods that operate on primitives. Without the wrapper

classes, these methods would have nowhere to go. You have already used the Character class method `isWhitespace`.

A further simplification of the code comes about from one of the new J2SE 5.0 features, autoboxing.

Autoboxing and Autounboxing

Autoboxing is the compiler's ability to automatically wrap, or "box" a primitive type in its corresponding wrapper class. Autoboxing only occurs when Java can find no method with a matching signature. A signature match occurs when the name and arguments of a message send match a method defined on the class of the object to which the message is being sent. An argument is considered to match if the types match exactly or if the type of the argument is a subtype of the argument type as declared in the method.

As an example, if `Box` declares a method as:

```
void add(List list) { ... }
```

then any of the following message sends will resolve to this `add` method:

```
Box b = new Box();
b.add(new List());
b.add(new ArrayList());
```

The second `add` message send works because `ArrayList` is a subclass of `List`.

If Java finds no direct signature match, it attempts to find a signature match based on wrapping. Any method with an `Object` argument in the appropriate place is a match.

In the `addCharge` method, the explicit wrapping of the `int` into an `Integer` is no longer necessary:

```
public void addCharge(int charge) {
    charges.add(charge);
}
```

It is important for you to understand that the wrapping does occur behind the scenes. Java creates a new `Integer` instance for each primitive value you wrap.

Autoboxing only occurs on arguments. It would be nice if autoboxing occurred anywhere you attempted to send a message to a primitive. This is not yet a capability, but it would allow expressions like:

```
6.toString() // this does not work
```

Autounboxing occurs when you attempt to use a wrapped primitive anywhere a primitive is expected. This is demonstrated in the following two tests:

```
public void testUnboxing() {
    int x = new Integer(5);
    assertEquals(5, x);
}

public void testUnboxingMath() {
    assertEquals(10, new Integer(2) * new Integer(5));
}
```

The more useful application of autounboxing is when you extract elements from a collection bound to a primitive wrapper type. With both auto-boxing and autounboxing, the code in Student becomes:

```
public void addCharge(int charge) {
    charges.add(charge);
}

public int totalCharges() {
    int total = 0;
    for (int charge: charges)
        total += charge;
    return total;
}
```

You will learn about addition capabilities of wrapper classes in Lesson 10 on mathematics.

Arrays

I have mentioned arrays several times in this book. An array is a fixed-size, contiguous set of slots in memory. Unlike an ArrayList or other collection object, an array can fill up, preventing you from adding more elements. The only way to add elements to a full array is to create a second larger array and copy all elements from the first array to the second array.

Arrays

Java provides special syntactical support for arrays. Indirectly, you have been using arrays all along, in the form of the class ArrayList. An ArrayList stores all elements you add to it in a Java array. The ArrayList class also encapsulates the tedium of having to grow the array when you add too many elements.



The example here creates a new class named `Performance`, which `CourseSession` might use to track test scores for each student. The `Performance` class allows calculating the average across all tests.

```
package sis.studentinfo;

import junit.framework.*;

public class PerformanceTest extends TestCase {
    private static final double tolerance = 0.005;
    public void testAverage() {
        Performance performance = new Performance();
        performance.setNumberOfTests(4);
        performance.set(0, 98);
        performance.set(1, 92);
        performance.set(2, 81);
        performance.set(3, 72);

        assertEquals(92, performance.get(1));

        assertEquals(85.75, performance.average(), tolerance);
    }
}
```

The `Performance` class:

```
package sis.studentinfo;

public class Performance {
    private int[] tests;
    public void setNumberOfTests(int numberOfWorks) {
        tests = new int[numberOfWorks];
    }

    public void set(int testNumber, int score) {
        tests[testNumber] = score;
    }

    public int get(int testNumber) {
        return tests[testNumber];
    }

    public double average() {
        double total = 0.0;
        for (int score: tests)
            total += score;
        return total / tests.length;
    }
}
```

Arrays

You store the scores for the tests in an `int` array named `tests`:

```
private int[] tests;
```

This declaration only says that the instance variable tests is of type int[] (you read this as “int array”). The braces are the indicator that this is an array. You use the braces to *subscript*, or index, the array. By subscripting the array, you tell Java the index of the element with which you wish to work.

Java allows you to declare an array with the braces following the variable.

```
private int tests[]; // don't declare arrays this way
```

Avoid this construct. Use of it will brand you as a C/C++ programmer (a fate worse than death in Java circles).

Neither of the two array declarations above allocate any memory. You will receive an error⁹ if you reference any of its (nonexistent) elements.

When a Performance client calls setNumberOfTests, the assignment statement initializes the test array.

```
public void setNumberOfTests(int numberOfTests) {
    tests = new int[numberOfTests];
}
```

You allocate an array using the new keyword. After the new keyword, you specify the type (int in this example) that the array will hold and the number of slots to allocate (numberOfTests).

Once you initialize an array, you can set a value into any of its slots. The type of the value must match the type of the array. For example, you can only set int values into an int[].

```
public void set(int testNumber, int score) {
    tests[testNumber] = score;
}
```

The line of code in set assigns the value of score to the slot in the tests array that corresponds to the testNumber. Indexing of arrays is 0-based. The first slot is slot #0, the second slot #1, and so on.

The get method demonstrates accessing an element in an array at a particular index:

```
public int get(int testNumber) {
    return tests[testNumber];
}
```

You can iterate an array like any other collection by using a for-each loop.

```
public double average() {
    double total = 0.0;
    for (int score: tests)
```

Arrays

⁹Specifically, a NullPointerException. See Lesson 8 for more information on exceptions.

```

        total += score;
    return total / tests.length;
}

```

The average method also shows how you may access the number of elements in an array by using a special variable named length.

You can iterate an array using a classic for loop:

```

public double average() {
    double total = 0.0;
    for (int i = 0; i < tests.length; i++)
        total += tests[i];
    return total / tests.length;
}

```

You may declare an array as any type, reference or primitive. You might have an instance variable representing an array of Students:

```
private Student[] students;
```

Array Initialization

When you allocate an array, Java initializes its elements to the default value for the type. Java initializes all slots in an array of numerics to 0, to false in an array of booleans, and to null in an array of references.

Java provides special array initialization syntax that you can use at the time of array instantiation and/or declaration.

```

public void testInitialization() {
    Performance performance = new Performance();
    performance.setScores(75, 72, 90, 60);
    assertEquals(74.25, performance.average(), tolerance);
}

```

The setScores method passes each of the four score arguments into an *array initializer*.

Arrays

```

public void setScores(int score1, int score2, int score3, int score4) {
    tests = new int[] { score1, score2, score3, score4 };
}

```

The array initializer in this example produces an array with four slots, each populated with an int value.

A shortcut that you can use as part of an array declaration assignment statement is:

```
int[] values = { 1, 2, 3 };
```

You may terminate the list of values in an array initializer with a comma. The Java compiler ignores the final comma (it does not create an additional slot in the array). For example, the preceding declaration and initialization is equivalent to:

```
int[] values = {  
    1,  
    2,  
    3,  
};  
assertEquals(3, values.length);
```

I coded the initialization on multiple lines to help demonstrate why this feature might be useful. It allows you to freely add to, remove from, and move elements around in the array initialization. You need not worry about leaving a comma after the last element; you can terminate each element with a comma.

When to Use Arrays

Prefer use of Java reference-type collections over native arrays. Your code will be simpler, closer to pure OO, and more flexible. It is easy to replace an `ArrayList` with a `LinkedList`, but it can be very costly to have to replace a native array with a `LinkedList`.

Some situations will force you into using arrays. Many existing APIs either return arrays or require arrays as arguments.

There are circumstances where arrays are more appropriate. Many mathematical algorithms and constructs (matrices, for example) are better implemented with arrays.

Accessing an element in an array is a matter of knowing which numbered slot it's in, multiplying by the slot size, and adding to some memory offset. The pseudo-memory diagram in Figure 7.1 shows a three-element `int` array. Java stores the starting location of the array in this example as `0x100`. To access the third element of the array (`x[2]`), Java calculates the new memory location as:

`0x100 starting address + 2nd slot * 4 bytes per slot`

Each slot requires four bytes, since it takes four bytes to represent an `int`.¹⁰ The result is the address `0x108`, where the element `55` can be found.

Arrays

¹⁰Java stores primitive types directly in the array. An array of objects would store a contiguous list of references to *other* locations in memory.

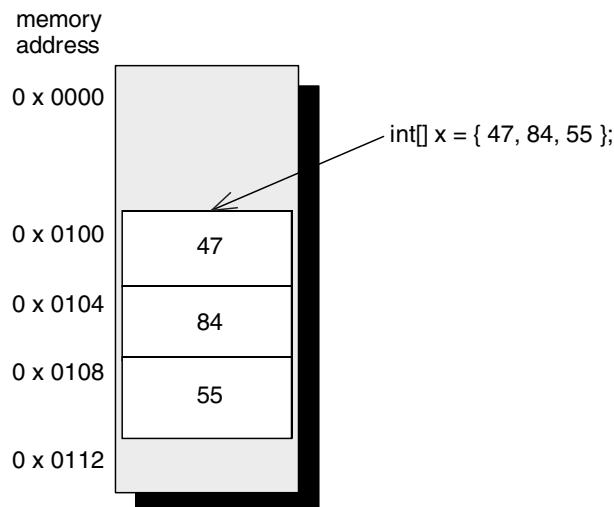


Figure 7.1 An Array in Memory

If you need the absolute fastest performance possible, an array provides a performance improvement over a collection class. However, prefer the use of collection classes over arrays. Always measure performance before arbitrarily converting use of a collection class to an array.



Prefer object-oriented collections over arrays.

Varargs

If you want to pass a variable number of similar-typed arguments to a method, you must first combine them into a collection. The `setScores` method above is restricted to four test scores; you would like to be able to pass any number of scores into a `Performance` object. Prior to J2SE 5.0, a standard idiom for doing so was to declare a new array on the fly:

Arrays

```
public void testArrayParm() {
    Performance performance = new Performance();
    performance.setScores(new int[] { 75, 72, 90, 60 });
    assertEquals(74.25, performance.average(), tolerance);
}
```

The `setScores` method can then store the array directly:

```
public void setScores(int[] tests) {
    this.tests = tests;
}
```

You might try to use the shorter array declaration/initialization syntax:

```
performance.setScores({ 75, 72, 90, 60 }); // this will not compile
```

but Java does not allow it.

J2SE 5.0 allows you to specify that a method takes a variable number of arguments. The arguments must appear at the end of the method argument list and must all be of the same type.

You declare the variability of a parameter with ellipses (...). Programmers familiar with C refer to this as a *varargs* declaration.

```
public void setScores(int... tests) {
    this.tests = tests;
}
```

Method calls to `setScores` can now have a variable number of test scores:

```
public void testVariableMethodParms() {
    Performance performance = new Performance();
    performance.setScores(75, 72, 90, 60);
    assertEquals(74.25, performance.average(), tolerance);

    performance.setScores(100, 90);
    assertEquals(95.0, performance.average(), tolerance);
}
```

When you execute a Java application, the VM passes command-line arguments to the `main` method as an array of String objects:

```
public static void main(String[] args)
```

You can declare the `main` method too as:

```
public static void main(String... args)
```

Multidimensional Arrays

Java also supports arrays of arrays, also known as multidimensional arrays. A classic use of multidimensional arrays is to represent matrices. You can create arrays that have up to 255 dimensions, but you will probably never need more than 3 dimensions. The following language test demonstrates how you allocate, populate, and access a two-dimensional array.

```
// 0 1 2 3
// 4 5 6 7
// 8 9 10 11
public void testTwoDimensionalArrays() {
    final int rows = 3;
    final int cols = 4;
    int count = 0;
```

Arrays

```

int[][] matrix = new int[rows][cols];
for (int x = 0; x < rows; x++)
    for (int y = 0; y < cols; y++)
        matrix[x][y] = count++;
assertEquals(11, matrix[2][3]);
assertEquals(6, matrix[1][2]);
}

```

You do not need to allocate all dimensions at once. You can allocate dimensions starting at the left, leaving the rightmost dimensions unspecified. Later, you can allocate these rightmost dimensions. In `testPartialDimensions`, the code initially allocates `matrix` as an `int` array of three rows. Subsequently, it allocates each slot in the `rows` portion of the array to differing-sized `int` arrays of columns.

```

// 0
// 1 2
// 3 4 5
public void testPartialDimensions() {
    final int rows = 3;
    int[][] matrix = new int[rows][];
    matrix[0] = new int[]{ 0 };
    matrix[1] = new int[]{ 1, 2 };
    matrix[2] = new int[]{ 3, 4, 5 };
    assertEquals(1, matrix[1][0]);
    assertEquals(5, matrix[2][2]);
}

```

As shown in `testPartialDimensions`, Java does not restrict you to rectangular multidimensional arrays; you can make them *jagged*.

You can initialize multidimensional arrays using array initialization syntax. The following initialization produces an array that is equivalent to the array initialized by `testPartialDimensions`:

```
int[][] matrix2 = { { 0 }, { 1, 2 }, { 3, 4, 5 } };
```

The Arrays Class

Arrays

Since arrays are not objects, you cannot send messages to them. You can access the variable `length` to determine the size of an array, but Java provides it as special syntax.

The class `java.util.Arrays` gives you a number of class methods to perform common operations on arrays. Using the `Arrays` class, you can do the following with a single-dimensional array:

- perform binary searches (which assumes the array is in sorted order) (`binarySearch`)

- sort it (`sort`)
- convert it to an object that implements the List interface (`asList`)
- compare it to another single-dimensional array of the same type (`equals`)
- obtain a hash code (see Lesson 9) (`hashCode`)
- fill each of its elements, or a subrange of its elements, with a specified value (`fill`)
- obtain a printable representation of the array (`toString`)

I'll discuss the `equals` method in more depth. For more information about the other methods, refer to the Java API documentation for more information on `java.util.Arrays`.

Arrays.equals

An array is a reference type, regardless of whether you declare it to contain primitives or references. If you create two separate arrays, the Java VM will store them in separate memory locations. This means that comparing the two arrays with `=` will result in `false`.

```
public void testArrayEquality() {
    int[] a = { 1, 2, 3 };
    int[] b = { 1, 2, 3 };
    assertFalse(a == b);
}
```

Since an array is a reference, you can compare two arrays using the `equals` method. But even if you allocate both arrays to exactly same dimensions and populate them with exactly the same contents, the comparison will return `false`.

```
public void testArrayEquals() {
    int[] a = { 1, 2, 3 };
    int[] b = { 1, 2, 3 };
    assertFalse(a.equals(b));
}
```

You can use the `Arrays.equals` method to compare the contents, not the memory location, of two arrays.

```
public void testArraysEquals() {
    int[] a = { 1, 2, 3 };
    int[] b = { 1, 2, 3 };
    assertTrue(Arrays.equals(a, b));
}
```

Arrays

Refactoring

Splitting a String

While either the hand-coded `while` loop or `for` loop in the `Student` method `split` will work, the resulting code is fairly complex. Java provides at least two more ways to break apart `String` objects into tokens. First, you will learn how to use the `StringTokenizer` class to break an input string into individual components, or *tokens*. The `StringTokenizer` class is considered a legacy class, meaning that Sun wants you to supplant its use with something better. That something better is the `split` method, defined on `String`. Once you have coded the solution using `StringTokenizer`, you will recode it to use `String.split`.

Up until Java 1.4, Sun recommended the `StringTokenizer` as the solution for tokenizing strings. You will still encounter a lot of code that uses `StringTokenizer`; that's why you'll learn it here. In Java 1.4, Sun introduced the regular expressions API. Regular expressions are specifications for pattern-matching against text. A robust, standardized language defines the patterns you can express with regular expressions. The newer `String.split` method uses a regular expression, making it far more effective than the `StringTokenizer` solution. For more information about the regular expressions API, see Additional Lesson III.

The constructor for the class `java.util.StringTokenizer` takes as parameters an input `String` and a `String` representing a list of character delimiters. `StringTokenizer` then breaks the input `String` into tokens, allowing you to iterate through each token in turn. The `while` loop provides the best mechanism for traversing these tokens:

```
private List<String> split(String name) {  
    List<String> results = new ArrayList<String>();  
    StringTokenizer tokenizer = new StringTokenizer(name, " ");  
    while (tokenizer.hasMoreTokens())  
        results.add(tokenizer.nextToken());  
    return results;  
}
```

Refactoring

Sending the message `hasMoreTokens` to a `StringTokenizer` returns `false` once there are no more tokens. The message `nextToken` returns the next token available. The `StringTokenizer` code uses the list of delimiters to determine what separates tokens. The example here uses only the space character to delineate tokens.

Splitting a String: `String.split`

The `String` class contains a `split` method that makes the job of tokenizing a full name even simpler than using `StringTokenizer`. Using `String.split` would allow you to eliminate your entire `Student.split` method with one line of code.

Here is a stab at the refactoring:

```
public Student(String fullName) {
    this.name = fullName;
    credits = 0;
    List<String> nameParts = split(fullName);
    setName(nameParts);
}

private List<String> split(String name) {
    return Arrays.asList(name.split(" ")); // this doesn't work!
}

private void setName(List<String> nameParts) {
    this.lastName = removeLast(nameParts);
    String name = removeLast(nameParts);
    if (nameParts.isEmpty())
        this.firstName = name;
    else {
        this.middleName = name;
        this.firstName = removeLast(nameParts);
    }
}

private String removeLast(List<String> list) {
    if (list.isEmpty())
        return "";
    return list.remove(list.size() - 1);
}
```

Part of your goal should be to make changes with as little impact to other code as possible. The single line of code that now appears in your `split` method seems like it should do the trick:

```
return Arrays.asList(name.split(" ")); // this doesn't work!
```

The `split` method takes a single parameter, the list of characters that represent word boundaries. It returns an array of words. The `Arrays` class method `asList` takes an array and returns a `List` representation of that array.

One would think that this line of code would be a suitable replacement for the previous code that constructed the list using `StringTokenizer`. But it doesn't work. You will receive many JUnit errors indicating that the `remove` call

Refactoring

in your `removeLast` method now generates something known as an `UnsupportedOperationException`.

The problem is in the `Arrays` method `asList`. According to the Java API documentation for `asList`, it returns a list *backed by* the array you pass as parameter. The list is a view onto the array, which does not go away. The list reflects any changes you make to the array.

Since the list is backed by an array, and since you cannot remove elements from an array, Java prohibits the `remove` call.

A solution that will work is to iterate through the results of `split` using a `for-each` loop. As with the `StringTokenizer` solution, you can add each word to a results array.

```
private List<String> split(String fullName) {
    List<String> results = new ArrayList<String>();
    for (String name: fullName.split(" "))
        results.add(name);
    return results;
}
```

The `split` method uses a Java feature known as *regular expressions*. A regular expression is a set of syntactical elements and symbols that is used to match text. You have likely used a simple form of regular expression when listing files in a directory. You use the asterisk (*) as a wildcard symbol to match any sequence of characters in a filename. See Additional Lesson III for an overview of the powerful regular expressions feature.

Exercises

- To see the different loop types in action, code the factorial algorithm (without using recursion). The definition of factorial:

```
if n is 0, n factorial = 1.  
if n is 1, n factorial = 1.  
if n is greater than one, n factorial is 1 * 2 * 3 * ... up to n
```

Exercises

Code the tests, then create a factorial method using a `while` loop. Next, modify the same method to pass the tests without using the `while` keyword, but instead using the C-style `for` loop. Finally, recode it again using the `do-while` loop. Notice the changes you must make in order to use the different loop structures. Also note that you only need one loop structure to solve any looping problem.

- Why would you prefer one loop structure over the others?
- Which is shortest?

- c) Which is the most appropriate to this problem? Why?
- d) Replace the loop with

```
while (true) {
```

and get the test to pass using the `break` keyword to end the loop.

2. Demonstrate your understanding of the `continue` keyword. Create a method to return a string containing the numbers from 1 to n . Separate each pair of numbers with a single space. Append an asterisk to the numbers divisible by five. For example, setting n to 12 would result in the string:

```
"1 2 3 4 5* 6 7 8 9 10* 11 12".
```

3. a) Break the results of the string from the previous exercise into a Vector of substrings. Split the string on the space character. The string "1 2 3 4 5* 6 7" would split into a Vector containing the strings "1", "2", "3", "4", "5*", "6", and "7".
- b) Iterate through the elements of the vector using an Enumeration and recreate the string from Exercise #2.
- c) Remove all parameterized type information, note the compiler errors, and correct the code where necessary.
4. In an earlier exercise, you wrote a test to ensure that the list of valid moves for a given piece contained a set of squares. Simplify this assertion to something like:

```
public void testKingMoveNotOnEdge() {
    Piece piece = Piece.createBlackKing();
    board.put("d3", piece);
    assertEquals(piece.getPossibleMoves("d3", board),
                "c4", "d4", "e4", "c3", "e3", "c2", "d2", "e2");
}
```

Refactor other tests to use this mechanism.

5. You may have noticed that the chess board you have spent so much time on is really a two-dimensional array of squares. Back up your Board class. Modify Board to use a two-dimensional array as its underlying data structure.

One bit of difficulty you may encounter is due to the fact that client code (such as Game and Pawn) uses the implementation specifics of the board. For this exercise, modify the client code to loop through the ranks and files of the two-dimensional array. This is a less-than-ideal situation: Normally you want to design a solid public interface to your class, one that does not change. When you change the implementation details of the Board class, the Game class should not be affected.

The alternative would be to dynamically convert the Piece two-dimensional array to a List of List of Piece objects each time a client

Exercises

wanted to iterate through it. But the best solution is to not require the client to do the iteration themselves. One sophisticated (and somewhat complex) solution to accomplish this involves the visitor design pattern. See the book *Design Patterns*¹¹ for further information. For now, exposing the board's underlying two-dimensional array will suffice.

Compare and contrast the two Board versions. Which is easier to read and understand? Which involves the least code?

6. Make the Board class iterable, so that you can access the pieces (not including “no piece” objects) using a for-each loop. A complex solution would involve creating a separate Iterator class that tracks both a rank and file index. A simpler solution would involve looping through the board’s matrix and adding each piece to a List object. You could then return the iterator from the List object.

Once you have made the class iterable, go back and change any code that loops through all pieces to use a for-each loop. Look for opportunities to refactor code so that you use the for-each loop as much as possible. For example, if you haven’t already, modify your code so that each Piece object stores its current rank and file.

7. Read the javadoc API for the ArrayList class and determine a way to simplify the last example in the chapter—the one that splits the student name and returns a List of name parts. It is possible to recode the split method without the use of a loop at all! Hint: Look at the constructors for ArrayList.

Exercises

¹¹[Gamma1995].

Lesson 8

Exceptions and Logging

In this lesson you will learn about exceptions and logging. Exceptions in Java are a transfer control mechanism. You generate exceptions to signal problematic conditions in your code. You write code to either acknowledge or handle exceptions that are generated.

Java provides a flexible logging facility that allows you to record information as your application executes. You might choose to log exceptions, interesting occurrences, or events that you want to track.

Things you will learn include:

- the try-catch block
- checked vs. unchecked (runtime) exceptions
- errors vs. exceptions
- the throws clause
- the exception hierarchy
- creating your own exception type
- exception messages
- working with multiple exceptions
- rethrowing exceptions
- working with the stack trace
- the finally block
- the Formatter classes
- the Java logging API
- logging to files
- logging handlers

**Exceptions and
Logging**

- logging properties
- logging hierarchies

Exceptions

In Lesson 4, you saw how your code generated a `NullPointerException` when you did something with a variable that was not properly initialized.

Exceptions are objects that represent exceptional conditions in code. You can create an `Exception` object and *throw* it. If you are aware that there might be exceptions, you can write code that explicitly deals with, or *catches*, them. Or, you can declare that your code chooses to not deal with exceptions, and let someone else handle the problem.

A thrown exception represents a transfer of control. You may throw an exception at any point in code; other APIs may similarly throw exceptions at any time. The VM may also throw exceptions. From the point the exception is thrown, the Java VM transfers control to the first place that deals with, or catches, the exception. If no code catches the exception, the result is abnormal program termination.¹

You will ultimately need to build a user interface that allows for entry of test scores for students in a session. The user interface is where anything can go wrong: a user can type a number that is too large, they can type nothing, or they can type garbage. Your job will be to deal with invalid input as soon as it is typed in.

Test scores typed into the user interface come in as `String` objects. You will need to convert these strings into `int`s. To do so, you can use the `Integer` wrapper class utility method named `parseInt`. The `parseInt` method takes a `String` as a number and attempts to convert it to a number. If successful, `parseInt` returns the appropriate `int`. If the source `String` contains invalid input, the code in `parseInt` throws a `NumberFormatException`.

Start with a simple test representing a successful case:

```
package sis.studentinfo;

import junit.framework.TestCase;

public class ScorerTest extends TestCase {
    public void testCaptureScore() {
        Scorer scorer = new Scorer();
        assertEquals(75, scorer.score("75"));
    }
}
```

Exceptions

¹This is not necessarily the case when your application is executing multiple threads. See the lesson on threading.

Get this to pass:

```
package sis.studentinfo;

public class Scorer {
    public int score(String input) {
        return Integer.parseInt(input);
    }
}
```

Then write a second test that shows what happens when invalid input is passed to the `score` method:

```
public void testBadScoreEntered() {
    Scorer scorer = new Scorer();
    scorer.score("abd");
}
```

This isn't the complete test, but it will demonstrate what happens when code throws an exception. Compile the code and run the tests. JUnit will report an error instead of a test failure. An error means that code within the test (or, of course, code that the test called) generated an exception that was not handled. The exception's stack trace appears in the details window in JUnit, showing that the `Integer` class generated a `NumberFormatException`.

You want your test to demonstrate that the `score` method generates an exception when clients pass invalid input to it. The test case documents a case for which you expect to get an exception. Thus, you want the test to pass if the exception occurs and fail if it does not.

Java provides a construct called a try-catch block that you use to trap exceptions. The standard form of a try-catch block contains two blocks of code. The try block consists of code that might throw an exception. The catch block contains code to execute *if* an exception is generated.

```
public void testBadScoreEntered() {
    Scorer scorer = new Scorer();
    try {
        scorer.score("abd");
        fail("expected NumberFormatException on bad input");
    }
    catch (NumberFormatException success) {
    }
}
```

Exceptions

The code in `testBadScoreEntered` presents the most common idiom used when testing for exceptions. The try block wraps the `score` message send, since it is the code that could generate the `NumberFormatException`. If the code in `score` generates an exception, the Java VM immediately transfers control from the point the exception was thrown to the catch block.

If execution of the code in `score` does not raise an exception, control proceeds normally and Java executes the next statement in the try block. In `testBadScoreEntered`, the next statement is a call to the JUnit method `fail`. `ScorerTest` inherits `fail` (indirectly) from `junit.framework.TestCase`. Execution of `fail` immediately halts the test method; the VM executes no more of its code. JUnit reports the test method as a failure. Calling `fail` is equivalent to calling `assertTrue(false)` (or `assertFalse(true)`).

You expect `score` to generate an exception. If it does not, something has gone wrong, and you want to fail the test. If `score` does generate an exception, the `fail` statement is skipped, since the VM transfers control to the catch block.

The catch block is empty. Tests are about the only place where your catch blocks should ever be empty. Normally, you want to deal with a trapped exception somehow. You would do this in the catch block. In this lesson, I will present you with options for managing a caught exception.

In `testBadScoreEntered`, receiving an exception is a good thing. You document that by naming the `NumberFormatException` object in the catch clause `success`. Most code you will encounter will use a generic name for the exception object, such as `e`. Nothing says that you can't improve on this. Use an exception name to describe why you expect an exception.

The “exception test” represents what other client code will have to deal with. Somewhere in the user interface code, you will have code that you structure very similar to the code in `testBadScoreEntered`. The test documents the potential for `score` to generate a `NumberFormatException` and under what circumstance that potential exists.

Dealing with Exceptions

One tactic for managing exceptions is to catch them as close to the source (the point at which they are generated) as possible. Once you catch an exception, you might be able to provide code that deals with it. Providing code that converts a problem into a legitimate action is a way of encapsulating an exception.

You can also head off the exception. In the `Scorer` class, you can provide an additional method, `isValid`, that allows the client to test the validity of the input string. You put the burden on the client to guard against calling `score` with invalid input.

```
// ScorerTest.java
public void testIsValid() {
    Scorer scorer = new Scorer();
```

```

        assertTrue(scorer.isValid("75"));
        assertFalse(scorer.isValid("bd"));
    }

// Scorer.java
public boolean isValid(String input) {
    try {
        Integer.parseInt(input);
        return true;
    }
    catch (NumberFormatException e) {
        return false;
    }
}

```

A client first calls `isValid`. If `isValid` returns `true`, the client could safely call the `score` method. If `isValid` returns `false`, the client could apprise the user of the invalid input.

One benefit of using this construct is that the client would not have to code a try-catch block. You want to use exceptions as little as possible to control the flow of your code. By catching an exception as close to the source as possible, you eliminate the need for clients to use try-catch blocks to manage their control flow.

Another possibility would be for the `score` method to have a try-catch block. When it caught a `NumberFormatException`, it would return `-1` or some other special int value that the client could recognize.

Checked Exceptions



Each course session at the university will have its own web page. After creating a `Session` object, you can send it a `String` representing the URL² of its web page. `Session` should create a `java.net.URL` object using this string. One of the constructors of `java.net.URL` takes a *properly formed* URL string. To be properly formed, a URL must follow a number of rules for its component parts, such as the protocol name and host name. If you create a `java.net.URL` with an improperly formed URL, the constructor of `java.net.URL` throws an exception of type `java.net.MalformedURLException`.

A simple test in `SessionTest` sets the URL into the session as a string. The test then retrieves the URL as a `java.net.URL` object. Finally, the test ensures

**Checked
Exceptions**

²Uniform Resource Locator, a pointer to a resource on the World Wide Web.

that the URL object's string representation (`toString`) is a match for the URL string argument.

```
public void testSessionUrl() {
    final String url = "http://course.langrsoft.com/cmsc300";
    session.setUrl(url);
    assertEquals(url, session.getUrl().toString());
}
```

The code in `Session` is just a setter and getter:

```
package sis.studentinfo;

import java.util.*;
import java.net.*;

abstract public class Session
    implements Comparable<Session>, Iterable<Student> {
    ...
    private URL url;
    ...
    public void setUrl(String urlString) {
        this.url = new URL(urlString);
    }

    public URL getUrl() {
        return url;
    }
    ...
}
```

When you compile, however, you will receive an error:

```
unreported exception java.net.MalformedURLException; must be caught or declared to be thrown
    this.url = new URL(urlString);
    ^
```

Java defines the exception type `MalformedURLException` as a *checked exception* (as opposed to an *unchecked* exception). A checked exception is an exception that you must explicitly deal with in your code. You may ignore an unchecked exception (such as `NumberFormatException`), but remember that ignoring an exception may not be safe.

Any code that can generate a checked exception *must be handled by the method in which the code appears*. You handle an exception either by using a try-catch block or by declaring that the method simply propagates the exception to any of *its* callers. To declare that the method propagates the exception (and thus otherwise ignores it), use the `throws` clause:

```
public void setUrl(String urlString) throws MalformedURLException {
    this.url = new URL(urlString);
}
```

Checked Exceptions

The code phrase throws `MalformedURLException` says that the method `setUrl` can possibly generate an exception. Any code that calls the `setUrl` method must either be enclosed in a try-catch block or it must similarly appear in a method that contains an appropriate `throws` clause.

The test itself, then, must either catch `MalformedURLException` or declare that it throws `MalformedURLException`. Compile to see for yourself.

```
public void testSessionUrl() throws MalformedURLException {
    final String url = "http://course.langrsoft.com/cmsc300";
    session.setUrl(url);
    assertEquals(url, session.getUrl().toString());
}
```

(Don't forget to add the `java.net` import statement to `SessionTest`.)

Unless you are specifically expecting to receive an exception as part of your test, simply declare that the test throws an exception. Do not enclose the `setUrl` call in a try-catch block. You are defining the test. For this test, you are defining a “happy case” in which you know that the URL is valid. Given these two things under your control, an exception should never be thrown during the execution of the test unless something is horribly wrong. You can safely ignore any exceptions for the purpose of such a positive test. In the unlikely event that an exception *is* thrown, JUnit will catch it and fail the test with an error.

You still need a negative test, one that demonstrates what the client will see when there *is* a problem:

```
public void testInvalidSessionUrl() {
    final String url = "https://course.langrsoft.com/cmsc300";
    try {
        session.setUrl(url);
        fail("expected exception due to invalid protocol in URL");
    }
    catch (MalformedURLException success) {
    }
}
```

You will need to import the `MalformedURLException` class.

Exception Hierarchy

Exception
Hierarchy

In order to be able to throw an object as an exception, it must be of type `Throwable`. The class `Throwable` is at the root of the exception hierarchy as defined in `java.lang`. `Throwable` has two subclasses: `Error` and `Exception`.

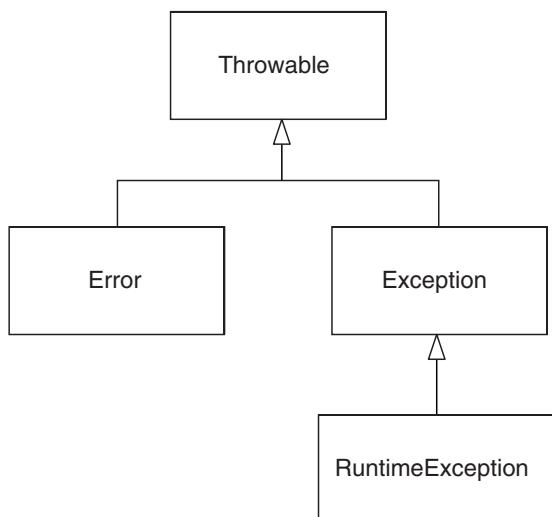


Figure 8.1 *The Exception Hierarchy*

The checked and unchecked exceptions you will work with derive from the class `Exception`.

Sun reserves the `Error` class for serious problems that can go wrong with the Java environment itself. Errors are unchecked, and you should not have any usual reason for catching instances of the `Error` class. Error exception types include things such as `OutOfMemoryError` and `InternalError`. You are not expected to write code to recover from any of these situations.

The `Exception` class is the superclass of all other exception types. Sun defines exceptions generated by the VM or the Java class library as `Exception` subclasses. Any exceptions that you define should also be `Exception` subclasses. Subclasses of `Exception` are sometimes referred to as application exceptions.

Unchecked application exceptions must derive from `RuntimeException`, which in turn directly subclasses `Exception`.

Figure 8.1 shows the core of the exception hierarchy.

Creating Your Own Exception Type

**Creating Your
Own Exception
Type**

You can create and throw an exception object that is of an existing Java exception type. But creating your own custom exception type can add clarity to your application. A custom exception type can also make testing easier.

As mentioned earlier, you want to propagate exceptions for things that you cannot otherwise control in your application. Prefer any approach where you have a means of preventing an exception from being thrown.

One case where exceptions are more appropriate is when a constructor takes a parameter that must be validated, as in the URL example. You could control validation via a static method call (for example, `public static boolean URL.isValid(String url)`). But it is reasonable for a constructor to generate an exception as an alternative to forcing clients to first make a static method call.

 For example, you may want to reject creation of students with more than three parts to their name, as in this test implemented in `StudentTest`:

```
public void testBadlyFormattedName() {
    try {
        new Student("a b c d");
        fail("expected exception from 4-part name");
    }
    catch (StudentNameFormatException success) {
    }
}
```

The first step toward realizing this test is creating a new Exception subclass. Before extending `Exception` directly, you should peruse the `Exception` types that Sun provides in the Java class library. Look for a similar `Exception` type on which you might want to base yours. In this case, `java.lang.IllegalArgumentException` is the most appropriate place to start. Create your `StudentNameFormatException` class as a subclass of `IllegalArgumentException`:

```
package sis.studentinfo;

public class StudentNameFormatException
    extends IllegalArgumentException {
}
```

That's all you need to do to define the exception type. Your new test should now compile and fail, since you do not yet generate the exception in the code. To make the test pass:

```
public Student(String fullName) {
    this.name = fullName;
    credits = 0;
    List<String> nameParts = split(fullName);
    final int maximumNumberOfNameParts = 3;
    if (nameParts.size() > maximumNumberOfNameParts)
        throw new StudentNameFormatException();
    setName(nameParts);
}
```

The code demonstrates that you create a new `Exception` object just like any other object:

```
throw new StudentNameFormatException();
```

Creating Your
Own Exception
Type

You throw the exception object using the `throw` keyword:

```
throw new StudentNameFormatException();
```

Since your class `StudentNameFormatException` is a subclass of `IllegalArgumentException`, and since `IllegalArgumentException` is a subclass of `RuntimeException`, you do not need to declare the `throws` clause on the `Student` constructor.

Note that there is no test class for `StudentNameFormatException`! The reason is that there is virtually no code that could break. Yes, you could forget to extend from an `Exception` superclass, but your code would then not compile.

Checked Exceptions vs. Unchecked Exceptions

As the code designer, you may designate `StudentNameFormatException` as checked or unchecked. There is considerable debate on which to prefer. One argument suggests that checked exceptions cause more trouble than they are worth.³ One reason is that changing method signatures to add new `throws` types breaks client code—potentially lots of it!

The reality is that most code can't do anything about most exceptions. Often, the only appropriate place to manage exceptions is at the user interface layer: “Something went horribly wrong in our application; please contact the support desk.” An exception generated by code buried deep in the system must trickle out to the user interface code layer. Forcing all intermediary classes to acknowledge this potential problem clutters your code.

A general recommendation is to eliminate the need for clients to deal with exceptions as much as possible. Consider a way of returning a default value that the client can manage as part of normal processing. Or consider instead requiring a client to follow a protocol where they execute some sort of validation. See the example above, where clients of the `Scorer` class first execute the method `isValid`.

Some developers promote checked exceptions, claiming that forcing client developers to immediately and always acknowledge exceptions is beneficial. Unfortunately, many developers choose a far worse solution when confronted with checked exception types:

Checked
Exception vs.
Unchecked
Exceptions

³[Venners2003].

```
try {
    doSomething();
}
catch (Exception e) {
    // log or do nothing
}
```

Doing nothing within a catch block is an anti-pattern known as Empty Catch Clause.⁴ Logging the exception is tantamount to doing nothing. There are a few rare cases where Empty Catch Clause is appropriate. But doing nothing means that you are hiding a potentially serious problem that could cause even worse problems down the road.



Avoid propagating exceptions, but do not create empty catch blocks.

Messages

Most exceptions store an associated message. The message is more often than not for developer consumption. Only sometimes is the message appropriate to present to an end user of an application (although the message might be used as the determining basis for an appropriate end user message). You typically use the message for debugging purposes. Often you will record the message in an application log file to help document what caused a problem.⁵

The class `Exception` allows you to create a new exception object with or without a message. You can retrieve the message from an exception using the method `getMessage`. For an exception object created without a message, `getMessage` returns `null`.

You may want to store an error message within a `StudentNameFormatException` object that provides more information.

```
public void testBadlyFormattedName() {
    try {
        new Student("a b c d");
        fail("expected exception from 4-part name");
    }
    catch (StudentNameFormatException expectedException) {
        assertEquals(
            "Student name 'a b c d' contains more than 3 parts",
            expectedException.getMessage());
    }
}
```

Messages

⁴[Wiki2004a].

⁵See the second half of this lesson for a detailed discussion of logging.

Instead of declaring a temporary variable `success` to hold the exception object, you now declare the variable name as `expectedException`. Just catching the exception is no longer enough for success; the message stored in `expectedException` must also be as expected.

The test will fail, since `e.getMessage()` returns `null` by default. You must modify the `StudentNameFormatException` constructor to take a message parameter. The constructor can then call the corresponding superclass constructor directly.

```
package sis.studentinfo;

public class StudentNameFormatException
    extends IllegalArgumentException {
    public StudentNameFormatException(String message) {
        super(message);
    }
}
```

The `Student` constructor can then put together a message and pass it off to the exception constructor.

```
public Student(String fullName) {
    this.name = fullName;
    credits = 0;
    List<String> nameParts = split(fullName);
    final int maximumNumberOfNameParts = 3;
    if (nameParts.size() > maximumNumberOfNameParts) {
        String message =
            "Student name '" + fullName +
            "' contains more than " + maximumNumberOfNameParts +
            " parts";
        throw new StudentNameFormatException(message);
    }
    setName(nameParts);
}
```

The exception message string does represent duplication between the test and code. You may refactor it now if you choose. Later in this lesson, you will learn how to dynamically construct the message using the `String` method `format`. I will refactor the code at that time.

Catching Multiple Exceptions

Catching Multiple Exceptions

It is possible for a single line of code to generate more than one exception. Each exception can be of a different exception type. A method that throws more than one exception can list each exception type individually:

```
public void send() throws ObjectOutputStream, UnknownHostException
```

If both exceptions derive from the same class, you can instead declare that the method throws an exception of the common supertype:

```
public void send() throws Exception
```

The try-catch block can have multiple catch clauses, each representing an exception that might be thrown.

```
try {
    new Student("a b c d");
}
catch (StudentNameFormatException e) {
    ...
}
catch (NoSuchElementException e) {
    ...
}
```

If code within the try block throws an exception, the Java VM transfers control to the first catch block. If the type of the exception thrown matches the type declared in the catch clause, the VM executes code within that catch block. Otherwise, the VM transfers control to the next catch block, and so on. Once code in a catch block is executed, the VM ignores all other catch blocks.

In the following code, the second catch block catches any exception that is of type Exception. Since all (normal) exceptions extend from Exception, this “catch-all” block will trap any application exception other than StudentNameFormatException, which you already trapped with the first catch block.

```
try {
    new Student("a b c d");
}
catch (StudentNameFormatException e) {
    ...
}
catch (Exception e) {
    ...
}
```

A catch-all is useful for trapping any unexpected exceptions. You should usually include a catch-all only in your top-level class, the one closest to the user interface layer. About the only thing you will be able to do in the catch-all is to log the error and possibly show something to the end user. What you gain is the ability to keep the application from crashing or visibly failing.

Use extreme discretion when introducing a catch-all. If you introduce new code in the try block or in the code it calls, this code can generate a new

**Catching
Multiple
Exceptions**

exception. The problem is that you might be unaware of the new exception, even if it is a checked exception. Hiding errors is bad.

Rethrowing Exceptions

Nothing prevents you from catching an exception and then throwing it (or another exception) from within the catch block. This is known as *rethrowing* an exception.

A common reason to rethrow an exception is to catch it as close to its source as possible, log it, and then propagate it again. This technique can make it easier to determine the source of a problem.

```
public void setUrl(String urlString) throws MalformedURLException {
    try {
        this.url = new URL(urlString);
    }
    catch (MalformedURLException e) {
        log(e);
        throw e;
    }
}

private void log(Exception e) {
    // Logging code. The second half of this lesson contains more
    // info on logging. For now, this method is empty.
}
```

In the above changes to `setUrl`, code in the catch block passes the caught exception object to the `log` method, then rethrows it. A refinement of this technique is to create and throw a new exception of a more application-specific type. Doing so can encapsulate the specific implementation details that caused the exception.

Instead of throwing a `MalformedURLException`, you want to throw an instance of the application-specific exception type `SessionException`. Once you have created the exception class:

```
package sis.studentinfo;

public class SessionException extends Exception {
}

You can alter test code to reflect the new exception type:

public void testSessionUrl() throws SessionException {
    final String url = "http://course.langrsoft.com/cmsc300";
```

**Rethrowing
Exceptions**

```

        session.setUrl(url);
        assertEquals(url, session.getUrl().toString());
    }

    public void testInvalidSessionUrl() {
        final String url = "https://course.langrsoft.com/cmsc300";
        try {
            session.setUrl(url);
            fail("expected exception due to invalid protocol in URL");
        }
        catch (SessionException success) {
        }
    }
}

```

Finally, you can change the production code to throw the new exception instead.

```

public void setUrl(String urlString) throws SessionException {
    try {
        this.url = new URL(urlString);
    }
    catch (MalformedURLException e) {
        log(e);
        throw new SessionException();
    }
}

```

The downside is that you lose information with this solution: If an exception is thrown, what is the true reason? In a different circumstance, code in the try block might throw more than one kind of exception. Rethrowing the application-specific exception would hide the root cause.

As a solution, you could extract the message from the root exception and store it in the SessionException instance. You would still lose the original stack trace information, however.

In J2SE 1.4, Sun added the ability to the Throwable class to store a root cause. The Throwable class provides two additional constructor forms: one that takes a Throwable as a parameter and another that takes both a message string and a Throwable as a parameter. Sun also added these constructors to the Exception and RuntimeException derivatives of Throwable. You may also set the root cause in the Throwable object after its construction via the method `initCause`. Later, you can retrieve the Throwable object from the exception by sending it the message `getCause`.

First, modify `testInvalidSessionUrl` to extract the cause from the SessionException instance. As part of the test verification, ensure that the cause of the exception is the reason expected.

Rethrowing
Exceptions

```

public void testInvalidSessionUrl() {
    final String url = "https://course.langrsoft.com/cmsc300";
    try {
        session.setUrl(url);
        fail("expected exception due to invalid protocol in URL");
    }
    catch (SessionException expectedException) {
        Throwable cause = expectedException.getCause();
        assertEquals(MalformedURLException.class, cause.getClass());
    }
}

```

The `assertEquals` statement ensures that the class of the cause is `MalformedURLException`. The code to make this comparison demonstrates some *reflective* capabilities of Java—the ability of the language to dynamically derive information about types and their definitions at runtime. I will explain reflection capabilities in further depth in Lesson 12.

In short, you can send all objects the message `getClass`, which returns an appropriate class constant. A class constant is a class name followed by `.class`. `MalformedURLException.class` is a class constant that represents the `MalformedURLException` class.

The test will fail: If no cause is explicitly set, `getCause` returns `null`. You will first need to modify `SessionException` to capture the cause upon construction:

```

package studentinfo;

public class SessionException extends Exception {
    public SessionException(Throwable cause) {
        super(cause);
    }
}

```

You can then change the production code to embed the cause in the `SessionException` instance.

```

public void setUrl(String urlString) throws SessionException {
    try {
        this.url = new URL(urlString);
    }
    catch (MalformedURLException e) {
        log(e);
        throw new SessionException(e); // <- here's the change
    }
}

```

Rethrowing Exceptions

Stack Traces

Lesson 4 taught you how to read the exception stack to help decipher the source of an exception.

You can send the stack trace stored within an exception to a print stream or print writer using the `printStackTrace` method defined in `Throwable`. One implementation of `printStackTrace` takes no parameters, printing the stack trace on the system console (`System.out`) by default. A crude implementation of the `log` method might do just that.

```
private void log(Exception e) {  
    e.printStackTrace();  
}
```

(Try it.) For a production system, you will want a more robust logging solution. Sun introduced a logging API in J2SE 1.4; I discuss logging in the second half of this lesson.

The stack trace representation printed by `printStackTrace` contains information that might be useful for more advanced programming needs. You could parse the stack trace yourself in order to obtain the information, which many developers have done. Or, as of J2SE 1.4, you can send the message `getStackTrace` to the exception object in order to access the information in already parsed form.

The `finally` Block

When an exception is thrown, the Java VM immediately transfers control from its point of origin to the first catch block matching the exception type. Java executes no further lines of code within the try block. If no try block exists, the VM immediately transfers control out of the method. You may have the need, however, to ensure that some bit of code *is* executed before control transfers out.

The optional `finally` block ensures that the Java VM always executes some piece of code, regardless of whether or not an exception was thrown. You may attach a single `finally` block to the end of a try-catch block.

The prototypical reason for using a `finally` block is to clean up any local resources. If you have opened a file, the `finally` block can ensure that the file gets properly closed. If you have obtained a connection to a database, the `finally` block ensures that the connection is closed.

The `finally`
Block

```

public static Student findByLastName(String lastName)
    throws RuntimeException {
    java.sql.Connection dbConnection = null;
    try {
        dbConnection = getConnection();
        return lookup(dbConnection, lastName);
    }
    catch (java.sql.SQLException e) {
        throw new RuntimeException(e.getMessage());
    }
    finally {
        close(dbConnection);
    }
}

```

(Note that the example is here for demonstration purposes only. Refer to Additional Lesson III for a brief overview of interacting with databases via JDBC.)

Both the `lookup` method and the `getConnection` method can throw an `SQLException` object. You call the `lookup` method after obtaining a connection. If the `lookup` method subsequently throws an exception, you want to ensure that the database connection gets closed.

If no exception is thrown, Java transfers control to the `finally` block upon completion of the code in the `try` block. In the example, the `finally` block makes a call to close the database connection. If an exception *is* thrown, Java executes code in the `catch` block. As soon as either the `catch` block completes or the VM directs control out of the `catch` block (in the above example, through use of the `throw` statement), Java executes the `finally` block, which closes the connection.

If you supply a `finally` block, the `catch` block itself is optional. For example, you may not want to attempt to handle an SQL exception from within `findByLastName`. Instead, you declare that `findByLastName` can throw an `SQLException`. But you still need to close the connection before the VM transfers control out of `findByLastName`:

```

public static Student findByLastName(String lastName)
    throws java.sql.SQLException {
    java.sql.Connection dbConnection = null;
    try {
        dbConnection = getConnection();
        return lookup(dbConnection, lastName);
    }
    finally {
        close(dbConnection);
    }
}

```

The finally Block

Under no circumstances should you include a `return` statement within a `finally` block—it will “eat” any exceptions that might have been thrown by the `catch` block. Some developers also promote the more strict notion that you should not return from either a `try` block or a `catch` block; instead only return after completion of the whole `try-catch` block.

Refactoring

Creating a string to present to the user often involves concatenating several substrings and printable representations of primitives and objects. The `Student` constructor provides an example:

```
public Student(String fullName) {
    this.name = fullName;
    credits = 0;
    List<String> nameParts = split(fullName);
    final int maximumNumberOfNameParts = 3;
    if (nameParts.size() > maximumNumberOfNameParts) {
        String message =
            "Student name '" + fullName +
            "' contains more than " + maximumNumberOfNameParts +
            " parts";
        throw new StudentNameFormatException(message);
    }
    setName(nameParts);
}
```

In the `Student` constructor, you combine five separate elements to form a single string—three substrings, the full name, and a constant representing the maximum number of name parts allowed. You must likewise construct a string in the test method. Concatenations of larger strings, with more embedded information, become very difficult to read and maintain.

The `String` class supplies a class method named `format`. The `format` method allows you to pass a `format` String and a variable number of arguments (see the information on `varargs` in Lesson 7). In return, you receive a string formatted to your needs. A `format` String contains elements known as *format specifiers*. These specifiers are placeholders for the arguments. A `format` specifier tells the formatter how to interpret and format each argument. Often, but not always, you will have one `format` specifier for each argument.

For those of you who are familiar with the language C, Java’s `String` formatting capability derives from a comparable C feature used in functions such as `printf`. The Java implementation is similar but provides more features

Refactoring

and safety. Do not assume that a Java format specifier works the same as an equivalent C format specifier.

Start with `testBadlyFormattedName` and replace the concatenation with a call to `format`. Also, introduce a class constant for the maximum number of name parts allowed.

```
public void testBadlyFormattedName() {
    final String studentName = "a b c d";
    try {
        new Student(studentName);
        fail("expected exception from 4-part name");
    }
    catch (StudentNameFormatException expectedException) {
        assertEquals(
            String.format("Student name '%s' contains more than %d parts",
                studentName, Student.MAX_NAME_PARTS),
            expectedException.getMessage());
    }
}
```

The call to `format` has three arguments: the format string and two format string arguments. Within the format string are two format specifiers: `%s` and `%d`. The first format specifier corresponds to the first argument, `studentName`. The second format specifier corresponds to the second argument, `Student.MAX_NAME_PARTS`. A format specifier always begins with a percent sign (%) and ends with a conversion. A conversion is a character or character that tells the `format` method how to interpret the corresponding argument.

The character `s`, as in `%s`, indicates a string conversion. When the `format` method encounters a string conversion in a format specifier, it replaces the format specifier with the corresponding argument. In the test, `format` replaces `%s` with the contents of `studentName`:

```
Student name 'a b c d' contains more than %d parts
```

The character `d`, as in `%d`, indicates an integral conversion to decimal. In the test, `format` replaces `%d` with the value of `Student.MAX_NAME_PARTS` (which you will set to the `int` literal 3):

```
Student name 'a b c d' contains more than 3 parts
```

Refactoring

The `String` method `format` is a utility method that delegates all the work to an instance of the `java.util.Formatter` class. The `Formatter` class supplies more than a dozen different conversions, including date conversions and more-sophisticated conversions for numeric values.

One very useful conversion is `\n`, which stands for “new line.” If you provide the format specifier `%n`, the formatter replaces it with the platform-specific line separator (usually `"\n"` or `"\r\n"`). This saves you from having to provide code that accesses the system line separator property.

Some of the other things Formatter supports:

- internationalization
- the ability to provide arguments in an order that does not match the order of format specifiers
- sending output incrementally to a `StringBuilder` or other sink

The API documentation for `java.util.Formatter` is long and detailed. Refer to it to understand how to use additional conversions or how to work with the additional features of `Formatter`.

As a final step, you can refactor to introduce a class constant for the error message. The following code also shows the implementation in `Student`:

```
// StudentTest.java
public void testBadlyFormattedName() {
    final String studentName = "a b c d";
    try {
        new Student(studentName);
        fail("expected exception from 4-part name");
    }
    catch (StudentNameFormatException expectedException) {
        assertEquals(
            String.format(Student.TOO_MANY_NAME_PARTS_MSG,
                studentName, Student.MAX_NAME_PARTS),
            expectedException.getMessage());
    }
}

// Student.java
static final String TOO_MANY_NAME_PARTS_MSG =
    "Student name '%s' contains more than %d parts";
...
public Student(String fullName) {
    this.name = fullName;
    credits = 0;
    List<String> nameParts = split(fullName);
    if (nameParts.size() > MAX_NAME_PARTS) {
        String message =
            String.format(Student.TOO_MANY_NAME_PARTS_MSG,
                fullName, MAX_NAME_PARTS);
```

Refactoring

```
        throw new StudentNameFormatException(message);
    }
    setName(nameParts);
}
```

Logging

In most enterprises, it is valuable to track interesting history as applications execute. The job of *logging* code is to record significant events and data supporting those events. As a developer, it is your job to determine what is relevant enough to track. Once you make this determination, you insert code at judicious points in the application to write the information out to (typically) a log file or files. You later can use the logs to answer questions about performance, problems, and so on.

Figuring out just what to track is the hard part. Currently, you are logging nothing. You lose information on any errors that occur. The lack of relevant problem data can deprive you of the ability to solve problems. Worse, you may not know you have a problem until it is too late (this is one reason to avoid empty catch blocks).

Minimally, you should log all unexpected error conditions (exceptions). You may also want to log critical calculations, execution of troublesome routines, or interesting data. You can also choose to log virtually everything that occurs, such as every time a method executes.

Logging everything, or logging too much, has its problems. Log files can grow at an extreme rate, particularly as use of the system increases. Logging has a minor performance penalty; excessive logging can lead to system slowdown. A worse problem, however, is that if there is too much information being logged, you will not be able to physically analyze the voluminous amount of data. Problems may get lost in the forest. Logging becomes useless. On the code side, logging has a cluttering effect and can bloat your code.

It will be up to you to determine how much to log. In some circumstances, where it is easy for you to deploy updated code (for example, a web application), err on the side of logging too little information. Where it is difficult to update deployed code, err on the side of logging too much information. Having more information will increase your prospects of solving a problem. Err on the side of logging too much information when you know that a section of code has problems or when the code presents a high risk (for example, code that controls critical components).

Logging

Logging in Java

Java supplies complete logging facilities in the package `java.util.logging`.⁶ For an exercise, you will learn to use the logging package to capture exception events. The `Student` constructor throws an exception. Let's log it. Here is a slightly modified `Student` constructor with a placeholder for logging.

```
public Student(String fullName) {
    this.name = fullName;
    credits = 0;
    List<String> nameParts = split(fullName);
    if (nameParts.size() > MAX_NAME_PARTS) {
        String message =
            String.format(Student.TOO_MANY_NAME_PARTS_MSG,
                         fullName, MAX_NAME_PARTS);
        // log the message here
        throw new StudentNameFormatException(message);
    }
    setName(nameParts);
}
```

The first question is how to test that an appropriate message gets logged when you throw the exception. Or should you bother with such a test?

Remember the first rule of testing: Test everything that can possibly break. Logging can definitely break. Take a stab at a test:

```
public void testBadlyFormattedName() {
    final String studentName = "a b c d";
    try {
        new Student(studentName);
        fail("expected exception from 4-part name");
    }
    catch (StudentNameFormatException expectedException) {
        String message =
            String.format(Student.TOO_MANY_NAME_PARTS_MSG,
                         studentName, Student.MAX_NAME_PARTS);
        assertEquals(message, expectedException.getMessage());
        assertTrue(wasLogged(message));
    }
}

private boolean wasLogged(String message) {
    // ???
}
```

Logging in Java

⁶While Java's logging facilities are adequate for most needs, many developers prefer to use the freely available Log4J package (<http://logging.apache.org/log4j/docs/>).

How do you determine whether or not a message was logged? The method name `wasLogged` represents the intent of *what* you need to determine. The contents of `wasLogged` will represent *how* you determine it.

Learning how to test an unfamiliar API such as logging, sometimes involves playing around with the API until you understand how it works. Without this more complete understanding of the API, you may not be able to figure out how to test it. In this exercise, you'll write a bit of "spike" code that provides you with a better understanding of the logging API. Then you'll throw away the spike code, write a test, and write the production code to meet the test specification.

In `Student`, create an "intention message" in the constructor:

```
public Student(String fullName) {
    this.name = fullName;
    credits = 0;
    List<String> nameParts = split(fullName);
    if (nameParts.size() > MAX_NAME_PARTS) {
        String message =
            String.format(Student.TOO_MANY_NAME_PARTS_MSG,
                          fullName, MAX_NAME_PARTS);
        log(message);
        throw new StudentNameFormatException(message);
    }
    setName(nameParts);
}
```

Then code the `log` method.

```
private void log(String message) {
    Logger logger = Logger.getLogger(getClass().getName());
    logger.info(message);
}
```

To log a message, you need a `Logger` object. To obtain a `Logger` object, you call the `Logger` factory method `getLogger`. As a parameter to `getLogger`, you pass the name of the subsystem within which you are logging. Later in this lesson (in the section entitled Logging Hierarchies), you'll learn more about the relevance of the subsystem name. Passing in the name of the class is typical and will suffice.

If a `Logger` object with that name already exists, `getLogger` returns that logger. Otherwise, `getLogger` creates and returns a new `Logger` object.

The second line in `log` calls the `Logger` method `info`, passing the `message` parameter. By using the `info` method, you request that the message be logged at an informational level. The logging API supports several levels of message log.

Logging in Java

The Logger class provides a method that corresponds to each level. From highest to lowest, the levels are severe, warning, info, config, fine, finer, and finest.

Each logger object maintains its own logging level. If you attempt to log a message at a lower level than the logger's level, the logger discards the message. For example, if a logger is set to warning, it logs only severe and warning messages.

You can obtain and set the logging level on a logger using `getLevel` and `setLevel`. You represent the logging level with a `Level` object. The `Level` class defines a set of `Level` class constants, one to represent each logging level. The `Level` class also supplies two additional class constants: `Level.ALL` and `Level.OFF`, to designate either logging all messages or logging no messages.

Modify the `wasLogged` method in `StudentTest` to return `false`. Run all your tests. By returning `false`, you cause `testBadlyFormattedName` to fail. This failure reminds you that you will need to flesh out the test.

On the console, you should see something like:

```
Apr 14, 2005 2:45:04 AM sis.studentinfo.Student log
INFO: Student name 'a b c d' contains more than 3 parts
```

You have successfully logged a message!

The UML class diagram in Figure 8.2 shows the relationship between `Logger` and `Level`. It also includes `Logger`'s relationship to other relevant classes that you will learn about in the upcoming sections.

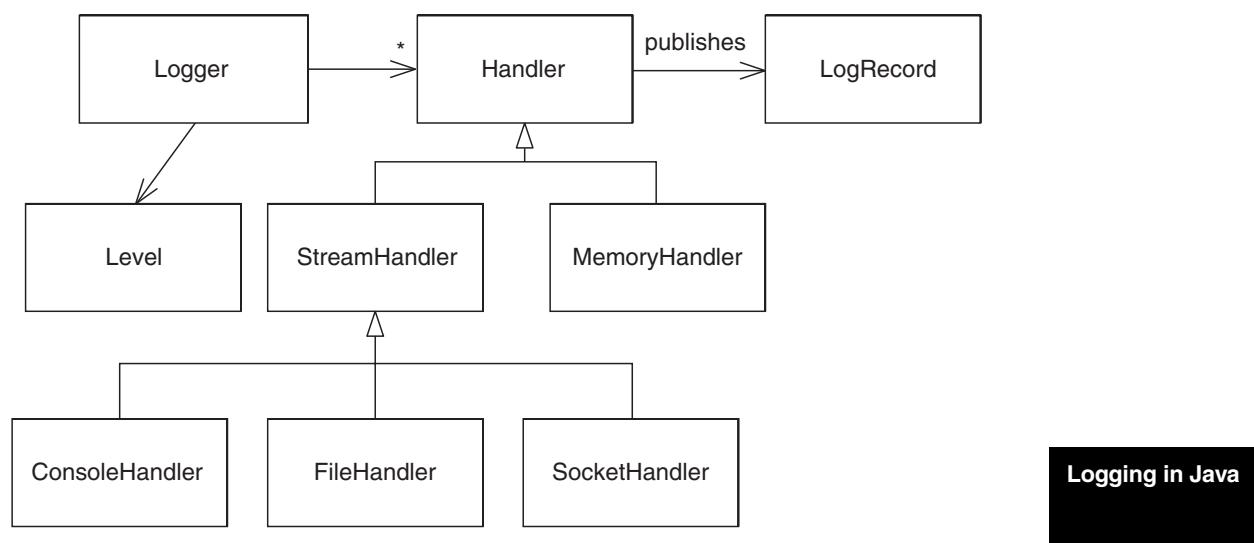


Figure 8.2 The Logger Hierarchy

Logging in Java

Testing Logging

Now your problem is figuring out how to test the logging. First, however, as I promised, you must throw away the spike code in `Student`. Discard the `log` method and update the constructor by removing the call to `log`.

```
public Student(String fullName) {
    this.name = fullName;
    credits = 0;
    List<String> nameParts = split(fullName);
    if (nameParts.size() > MAX_NAME_PARTS) {
        String message =
            String.format(Student.TOO_MANY_NAME_PARTS_MSG,
                fullName, MAX_NAME_PARTS);
        throw new StudentNameFormatException(message);
    }
    setName(nameParts);
}
```

Don't forget that writing code without tests is a good way of getting into trouble. Make sure you specify what you're going to code first.

While you have successfully logged a message, its output went to the console. How are you going to intercept this message for the purpose of writing a test?

The logging facility lets you direct messages to destinations other than the console. You can also direct logging to more than one destination at a time.⁷ The `Logger` stores handler objects that it uses to determine the logging destinations. A handler object is a subclass of `java.util.logging.Handler`. The class `ConsoleHandler` represents the behavior of sending output to the console. Most shops prefer to redirect logging output to files, using a `FileHandler`, so that developers can later peruse the files. You can tell the logger to route messages to alternate destinations by sending it `Handler` objects.

Unfortunately, you wouldn't be able to write test code that reads a log file, since you haven't learned about Java IO yet (you will in Lesson 11). Instead, you will create a new handler class, `TestHandler`. Within this handler class, you will trap all messages being logged. You will provide a getter method

Testing Logging

⁷The UML diagram in Figure 8.2 shows that a `Logger` can have multiple `Handler` objects by using an asterisk (*) at the end of the navigable association between the two classes. The * is a multiplicity indicator. The absence of a multiplicity indicates either 1 or that the multiplicity is uninteresting or irrelevant. The relationship between `Logger` and `Handler` is a one-to-many relationship.

that returns the last message that was logged. Once you have built this handler class, you can pass an instance of it to the logger.

For each message you log, the Logger object calls the method `publish` on the Handler object, passing it a `java.util.logging.LogRecord` parameter object.

It is the `publish` method that you will override in your Handler subclass to trap the message being logged. You must also supply definitions for the other two abstract methods in Handler, `flush` and `close`. They can be empty.

```
package sis.studentinfo;

import java.util.logging.*;

class TestHandler extends Handler {
    private LogRecord record;

    public void flush() {}
    public void close() {}
    public void publish(LogRecord record) {
        this.record = record;
    }

    String getMessage() {
        return record.getMessage();
    }
}
```

In `testBadlyFormattedName`, you first obtain the same logger that `Student` will use. Since both the test and the code in `Student` pass the `Student` class name to `getLogger`, they will both reference the same Logger object. Subsequently, you construct an instance of `TestHandler` and add it as a handler to the logger.

```
public void testBadlyFormattedName() {
    Logger logger = Logger.getLogger(Student.class.getName());
    TestHandler handler = new TestHandler();
    logger.addHandler(handler);
    ...
}
```

Any messages you log will be routed to the `TestHandler` object. To prove that `Student` code indeed logs a message, you can ask the handler for the last message it received.

```
public void testBadlyFormattedName() {
    ...
    final String studentName = "a b c d";
    try {
        new Student(studentName);
        fail("expected exception from 4-part name");
    }
    catch (StudentNameFormatException expectedException) {
        String message =
```

Testing Logging

```

        String.format(Student.TOO_MANY_NAME_PARTS_MSG,
                      studentName, Student.MAX_NAME_PARTS);
        assertEquals(message, expectedException.getMessage());
        assertTrue(wasLogged(message, handler));
    }
}

private boolean wasLogged(String message, TestHandler handler) {
    return message.equals(handler.getMessage());
}

```

On a stylistic note, I prefer coding the test as follows:

```

public void testBadlyFormattedName() {
    Logger logger = Logger.getLogger(Student.class.getName());
    Handler handler = new TestHandler();
    logger.addHandler(handler);

    final String studentName = "a b c d";
    try {
        new Student(studentName);
        fail("expected exception from 4-part name");
    }
    catch (StudentNameFormatException expectedException) {
        String message =
            String.format(Student.TOO_MANY_NAME_PARTS_MSG,
                          studentName, Student.MAX_NAME_PARTS);
        assertEquals(message, expectedException.getMessage());
        assertTrue(wasLogged(message, (TestHandler)handler));
    }
}

```

Instead of assigning the new instance of `TestHandler` to a `TestHandler` reference, you assign it to a `Handler` reference. This can clarify the test, so you understand that the `addHandler` method expects a `Handler` as a parameter (and not a `TestHandler`). The `wasLogged` method needs to use the `getMessage` method you created in `TestHandler`, so you must cast back to a `TestHandler` reference.

The test should fail, particularly since you should have no production code in `Student` to meet the specification of the test (you deleted it, correct?). Demonstrate the failure before proceeding. For all you know, the new test code might not do anything, and if you get a green bar, alarms should go off in your head. (It happens. Sometimes it means you forgot to compile your code.) Sticking to the method will help you keep your cool and make fewer mistakes.

Another reason you should have deleted the spike code is that you will produce a better, refactored version this time around. You currently have du-

Testing Logging

plicate code: Both StudentTest and Student include a complex line of code to retrieve a Logger object.

In addition to using a Student class variable for the logger, the modified test inlines the wasLogged method:

```
public void testBadlyFormattedName() {
    Handler handler = new TestHandler();
    Student.logger.addHandler(handler);

    final String studentName = "a b c d";
    try {
        new Student(studentName);
        fail("expected exception from 4-part name");
    }
    catch (StudentNameFormatException expectedException) {
        String message =
            String.format(Student.TOO_MANY_NAME_PARTS_MSG,
                         studentName, Student.MAX_NAME_PARTS);
        assertEquals(message, expectedException.getMessage());
        assertEquals(message, ((TestHandler)handler).getMessage());
    }
}
```

Here's the resurrected, refactored Student code:

```
package sis.studentinfo;

import java.util.*;
import java.util.logging.*;

public class Student {
    ...
    final static Logger logger =
        Logger.getLogger(Student.class.getName());
    ...
    public Student(String fullName) {
        this.name = fullName;
        credits = 0;
        List<String> nameParts = split(fullName);
        if (nameParts.size() > MAX_NAME_PARTS) {
            String message =
                String.format(Student.TOO_MANY_NAME_PARTS_MSG,
                             fullName, MAX_NAME_PARTS);
            Student.logger.info(message);
            throw new StudentNameFormatException(message);
        }
        setName(nameParts);
    }
    ...
}
```

Testing Logging

Logging to Files

Sun designed the logging facility to allow you to change logging characteristics at runtime. You can quickly route log messages to a file instead of to the console without having to change, recompile, and redeploy code.

The default behavior is to send logging messages to the console. This behavior is not hardcoded somewhere in the `Logger` class; it is located in an external properties file that you can freely edit.

Browse to the directory in which you installed Java. Within this directory, you should be able to navigate into the subdirectory `jre/lib` (or `jre\lib` if you are using Windows). Within this subdirectory you should see the file `logging.properties`.⁸ Edit the file using any editor.

You can use properties files in many circumstances to store values that you may want to change with each execution of an application. Sun chose to allow you the ability to configure logging behavior using a properties file. The layout of a properties file should be self explanatory. Comment lines begin with the pound sign (#). Blank lines are ignored. The remainder of the lines are key-value pairs, like entries in a hash table. Each key-value pair, or property, appears in the form:

```
key = value
```

Several lines down into `logging.properties`, you should see:

```
# "handlers" specifies a comma separated list of log Handler
# classes. These handlers will be installed during VM startup.
# Note that these classes must be on the system classpath.
# By default we only configure a ConsoleHandler, which will only
# show messages at the INFO and above levels.
handlers= java.util.logging.ConsoleHandler

# To also add the FileHandler, use the following line instead.
#handlers=java.util.logging.FileHandler,java.util.logging.ConsoleHandler
```

Swap the commenting around. Comment out the line that sets `handlers` to only the `ConsoleHandler`. Uncomment the line that sets the property `handlers` to both the `FileHandler` and the `ConsoleHandler`. By doing so, you tell the logger to route output to objects of each handler type.

Nearer to the bottom of `logging.properties`, you will find these lines:

```
java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter
```

⁸If you don't have this file, you can create it. Use the bits of content that I demonstrate.

Logging to Files

The logging facility uses these lines to determine how the FileHandler works.

The value for the `java.util.logging.FileHandler.pattern` is a pattern for naming the log file. The `%h` and `%u` that currently appear in the pattern are known as fields. The `%h` field tells the FileHandler to store log files in your home directory.

The file handler replaces the `%u` field in the pattern with a unique number. The goal is to avoid the conflict of two log files sharing the same filename.

Rerun your tests and then navigate to your home directory. Under most versions of Windows, you can do so using the command:⁹

```
cd %USERPROFILE%
```

Under most Unix shells you can navigate to your home directory using the command:

```
cd $HOME
```

Within your home directory, you should see a file named `java0.log`. The number `0` is the unique number replacement for the `%u` field.

View the contents of `java0.log`. They should look similar to this:

```
<?xml version="1.0" encoding="windows-1252" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
<date>2004-04-15T03:27:05</date>
<millis>1082021225078</millis>
<sequence>0</sequence>
<logger>sis.studentinfo.Student</logger>
<level>INFO</level>
<class>sis.studentinfo.Student</class>
<method>log</method>
<thread>10</thread>
<message>Student name 'a b c d' contains more than 3 parts</message>
</record>
</log>
```

It's not the two lines of logging message you expected. Instead, the message appears in XML format.¹⁰ The logging facility allows you attach a different formatter to each handler. A formatter is a subclass of `java.util.logging.Formatter`; Sun supplies two formatter implementations. `SimpleFormatter`

Logging to Files

⁹You must change to the drive on which the user profile is located. You can view the full path of your home directory, including the drive on which it is located, by executing `set USERPROFILE`.

¹⁰eXtensible Markup Language. See <http://www.w3.org/XML/>.

ter produces the two log lines you saw earlier. XMLFormatter produces the output you're currently seeing in `java0.log`. You can code your own formatter class to produce log output however you wish.

Find the `java.util.logging.FileHandler.formatter` property in `logging.properties`. Its current value is `java.util.logging.XMLFormatter`. Change the value to `java.util.logging.SimpleFormatter` and rerun your tests.

The contents of `java0.log` should now look the same as the output produced by the `ConsoleHandler`.

Testing Philosophy for Logging

In the previous section, Logging to Files, you changed logging behavior by modifying the `logging.properties` file. The ability to make dynamic changes using properties files is very powerful and allows your system to remain flexible.

Note that the test you wrote executed regardless of where the logging output went. Your test code instead proved that your code sent a specific message to the logger object. You were able to prove this by inspecting the message sent from the logger to one of its handlers.

However, the destination of logging output is another specification detail, one that needs to be adequately tested. It would be very easy to make invalid changes to the properties file. When the application shipped, those changes would cause serious problems. It is imperative that you test not only your Java code but also how the configuration of the system impacts that code.

You could write JUnit tests to ensure that log files were correctly created.¹¹ The strategy for the test would be:

1. write out a properties file with the proper data
2. force the logging facility to load this properties file
3. execute code that logs a message
4. read the expected log file and ensure that it contains the expected message

Or you could consider that this test falls into the scenario of what is known as integration testing. (Some shops may refer to this as customer testing, system testing, or acceptance testing.) You have already testing that your unit of code—Student—interacts with the logging facility correctly. Testing

Testing
Philosophy for
Logging

¹¹... once you learned how to write code that works with files. See Lesson 11.

how the logging facility works in conjunction with dynamic configurations begins to fall outside the realm of unit testing.

Regardless of the semantics—regardless of whether you consider such a test a unit test or not—it is imperative that you test the configuration that you will ship. You might choose to do this manually and visually inspect the log files like you did in the previous section. You might translate the above 4-step strategy into an executable test and call it an integration test. You will want to execute such tests as part of any *regression test* suite. A regression test suite ensures that new changes don't break existing code through execution of a comprehensive set of tests against the entire system.

Logging is a tool for supportability. Supportability is a system requirement, just like any other functional requirement. Testing logging at some level is absolutely required. But should you write a test for each and every log message?

The lazier answer is no. Once you have proved that the basic mechanics of logging are built correctly, you might consider writing tests for additional loggings a waste of time. Often, the only reason you log is so that you as a developer can analyze what the code is doing. Adding new calls to the logger can't possibly break things.

The better answer is yes. It will be tedious to maintain a test for every log message, but it will also keep you from getting into trouble for a few reasons.

Many developers introduce try-catch blocks because the compiler insists upon it, not because they have a specification or test. They don't have a solution for what to do when an exception occurs. The nonthinking reaction by the developer is to log a message from within the catch block. This produces an Empty Catch Clause, effectively hiding what could be a serious problem.

By insisting that you write a test for all logged messages, you don't necessarily solve the problem of Empty Catch Clause. But two things happen: First, you must figure out how to emulate generating the exception in question in order to code the test. Sometimes just going through this process will educate you about how to eliminate the need for an Empty Catch Clause. Second, writing a test elevates the folly of Empty Catch Clause to a higher level of visibility. Such a test should help point out the problem.

Another benefit of writing a test for every logged message is that it's painful. Sometimes pain is a good thing. The pain here will make you think about each and every logging message going into the system. "Do I really need to log something here? What will this buy me? Does it fit into our team's logging strategy?" If you take the care to log only with appropriate answers to these questions, you will avoid the serious problem of overlogging.

Finally, it is possible to introduce logging code that breaks other things. Unlikely, but possible.

Testing
Philosophy for
Logging

More on FileHandler

FileHandler defines more fields, including `%t` and `%g`. FileHandler code replaces occurrences of the `%t` field in the name pattern with the system temporary directory. On Windows, this is usually `c:\temp`; on Unix, this is usually `/tmp`.

The FileHandler code replaces occurrences of the `%g` field with a generation number. You use this field in conjunction with the `java.util.logging.FileHandler.count` (I'll refer to this as the shortened property-name `count`) and `java.util.logging.FileHandler.limit` (`limit`) properties. The `limit` property represents how many bytes will be stored in a log file before it reaches its limit. A `limit` value of `0` means there is no limit. When a log file reaches its limit, the FileHandler object closes it. The `count` field specifies how many log files that FileHandler cycles through. If the value of `count` is `1`, the FileHandler will reuse the same log file each time it reaches its limit.

Otherwise the FileHandler tracks a generation number. The first log file uses a generation number of `0` in place of the `%g` field. With a log filename pattern of `java%g.log`, the first generated log filename is `java0.log`. Each time a log file reaches capacity, the FileHandler bumps up the generation number and uses this to replace the `%g` field in the filename pattern. The FileHandler writes subsequent log entries to the log file with this new name. The second log file is `java1.log`. Once the FileHandler closes `count` log files, it resets the generation number to `0`.

Logging Levels

You may want to log messages for special purposes, such as for timing how long a critical method takes to execute or for introducing “trace” messages that allow you as a developer to follow the flow of execution in the system. You may not want these messages to normally appear in the logs when the application is run in production. Instead, you may need the ability to “turn on” these kinds of messages from time to time without having to recode, recompile, and redistribute the application.

As mentioned earlier, the Java logging API supports seven logging levels: `severe`, `warning`, `info`, `config`, `fine`, `finer`, and `finest`. You should restrict normal production logging to `severe`, `warning`, and `info` levels. Reserve the other levels for special, ephemeral logging needs.

Suppose you need to log the execution time of the `Student` method `getGpa`. You are concerned that it may not perform well under heavy load. Introduce logging messages at the beginning and end of `getGpa`, using the `fine` Logger method.

```

double getGpa() {
    Student.logger.fine("begin getGpa " + System.currentTimeMillis());
    if (grades.isEmpty())
        return 0.0;
    double total = 0.0;
    for (Grade grade: grades)
        total += gradingStrategy.getGradePointsFor(grade);
    double result = total / grades.size();
    Student.logger.fine("end getGpa " + System.currentTimeMillis());
    return result;
}

```

Recompile and rerun your tests.¹² You won't see these new logging messages either on your console or within a log file.

In order to view the logging messages, you must configure the handler to accept messages at the lower logging level. You typically do this within `logging.properties`. Edit the `logging.properties` file and make the modifications noted in bold:

```

...
.level= FINE

#####
# Handler specific properties.
# Describes specific configuration info for Handlers.
#####

# default file output is in user's home directory.
java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.level = FINE
java.util.logging.FileHandler.formatter=java.util.logging.SimpleFormatter

# Limit the message that are printed on the console to INFO and above.
java.util.logging.ConsoleHandler.level = INFO
...

```

You designate both the global level (`.level`) and the `FileHandler` level (`java.util.logging.FileHandler.level`) to accept any message logged at a `FINE` or higher level. The logging facility first ensures that the request level is as high as the global level; if not, it does not send the message to the handler. If the logger sends the message to the handler, the handler similarly determines whether the message should be logged.

In other words, the global should either be the same level or lower than any handler levels. If the reverse is true, no messages will be logged by a

Logging Levels

¹²Here I've chosen to take a lazy logging route—no testing—because these are temporary log messages.

handler. For example, if you were to set the global level to `INFO` and the `FileHandler` to `FINE`, the `FileHandler` would never log any messages.

Rerun the tests one more time and ensure that the timing log messages appear in the log file but not on the console.

Logging Hierarchies

When you retrieve a logger using the `Logger` class method `getLogger`, the `Logger` code returns either a new `Logger` object or an existing `Logger` object if the `Logger` has already created one with the same name. Calling `Logger.getLogger(Student.class.getName())` creates a logger with the name `"sis.studentinfo.Student"`. Using this fully qualified class name for a `Logger` is usually sufficient.

The following language test shows how `getLogger` returns the same `Logger` object when called twice with the same name:

```
public void testLoggingHierarchy() {
    Logger logger = Logger.getLogger("sis.studentinfo.Student");
    assertTrue(logger == Logger.getLogger("sis.studentinfo.Student"));
}
```

The fully qualified name of a class represents a hierarchical relationship. The top level of the hierarchy, or tree, is `sis`. The next lower level is `studentinfo`. Each class within `studentinfo` is a leaf on the tree. You can see this hierarchy more clearly by looking at the directory structure in which Java creates compiled class files.

You can take advantage of this naming hierarchy with respect to logging. The logging facility creates an analogous hierarchy of `Logger` objects. If you create a logger named `sis.studentinfo.Student`, its parent is `sis.studentinfo`. The parent of a logger named `sis.studentinfo` is named `sis`. You can demonstrate this by adding to the language test:

```
public void testLoggingHierarchy() {
    Logger logger = Logger.getLogger("sis.studentinfo.Student");
    assertTrue(logger == Logger.getLogger("sis.studentinfo.Student"));

    Logger parent = Logger.getLogger("sis.studentinfo");
    assertEquals(parent, logger.getParent());
    assertEquals(Logger.getLogger("sis"), parent.getParent());
}
```

**Logging
Hierarchies**

The benefit of this hierarchy is that you can set logging levels at a higher level. For example, you can set the logging level to `Level.ALL` at the `sis` logger. A child logger uses the logging level of its parent if it has none itself.

Additional Notes on Logging

- The logging facility allows you to attach arbitrary filters to a logger or handler. The Filter interface defines one method—`isLoggable`—which takes a LogRecord as a parameter. Using information in the LogRecord, the job of `isLoggable` is to return `true` if the message should be logged or `false` otherwise. You might code a Filter implementation that ignores log messages over a certain length, for example.
- The logging facility provides support for internationalization. See the Java API documentation for more information and Additional Lesson III for a brief discussion of internationalization in general.
- You may need to change logging properties while the application is already executing. Once you have changed the physical properties file, you can use the LogManager class method `readConfiguration` to reload the properties file. In order to do so, you will need to represent the properties file as an `InputStream`. See Lesson 11 for information on how to read files using input streams.
- You learned to modify the `logging.properties` to define logging characteristics at runtime. You can supply a custom configuration file instead of using the `logging.properties` file. To do so, set the value of the system property `java.util.logging.config.file`. One way to do this is on the command line when you start the Java application. For example:

```
java -Djava.util.logging.config.file=sis.properties sis.MainApp
```

- The `Logger` class provides convenience methods for logging special situations. You can use `entering` and `existing` to simplify logging entry to a method and exits from it. You can use the `throwing` method to simplify logging an exception. Refer to the Java API documentation for `Logger` for more information.

Exercises

Exercises

1. Write a test that calls a method named `blowsUp`. Code the method `blowsUp` to throw a new `RuntimeException` with the message "Somebody should catch this!". Run the test. Verify that it fails with an appropriate stack trace.

2. Make the test pass without removing the call to `blowsUp`. Ensure that the test fails if `blowsUp` throws no exception.
3. Ensure that the caught exception contains the proper message. Change the message, and see the test fail. Change it back in order to keep the tests passing.
4. Create a test that calls a new method named `rethrows`. The `rethrows` method should call `blowsUp` and catch the exception. It should then wrap the exception in a new `RuntimeException` and throw it again. Use `getCause` to ensure that the caught exception contains the original exception.
5. Modify the test to expect a new exception type, `SimpleException`, when calling the `blowsUp` method. `SimpleException` should extend `RuntimeException`.
6. Which of the following test methods will not compile? Of those that compile, which will pass and which will fail?

```
public void testExceptionOrder1() {  
    try {  
        blowsUp();  
        rethrows();  
        fail("no exception");  
    }  
    catch (SimpleException yours) {  
        fail("caught wrong exception");  
    }  
    catch (RuntimeException success) {  
    }  
}  
  
public void testExceptionOrder2() {  
    try {  
        rethrows();  
        blowsUp();  
        fail("no exception");  
    }  
    catch (SimpleException success) {  
    }  
    catch (RuntimeException failure) {  
        fail("caught wrong exception");  
    }  
}  
  
public void testExceptionOrder3() {  
    try {  
        blowsUp();  
        rethrows();  
        fail("no exception");  
    }  
    catch (RuntimeException success) {  
    }  
}
```

Exercises

```
    }
    catch (SimpleException yours) {
        fail("caught wrong exception");
    }
}

public void testExceptionOrder4() {
    try {
        blowsUp();
        rethrows();
        fail("no exception");
    }
    catch (RuntimeException fail) {
        fail("exception unacceptable");
    }
    catch (SimpleException yours) {
        fail("caught wrong exception");
    }
    finally {
        return;
    }
}

public void testExceptionOrder5() {
    try {
        blowsUp();
        rethrows();
        fail("no exception");
    }
    catch (SimpleException yours) {
        fail("caught wrong exception");
    }
    catch (RuntimeException success) {
    }
}

public void testExceptionOrder6() {
    try {
        rethrows();
        blowsUp();
        fail("no exception");
    }
    catch (SimpleException yours) {
        fail("caught wrong exception");
    }
    catch (RuntimeException success) {
    }
}

public void testExceptionOrder7() {
    try {
        rethrows();
        blowsUp();
        fail("no exception");
    }
```

Exercises

```
        }
        catch (SimpleException success) {
        }
        catch (RuntimeException fail) {
            fail("caught wrong exception");
        }
    }

public void testErrorException1() {
    try {
        throw new RuntimeException("fail");
    }
    catch (Exception success) {
    }
}

public void testErrorException2() {
    try {
        new Dyer();
    }
    catch (Exception success) {
    }
}

public void testErrorException3() {
    try {
        new Dyer();
    }
    catch (Error success) {
    }
}

public void testErrorException4() {
    try {
        new Dyer();
    }
    catch (Throwable success) {
    }
}

public void testErrorException5() {
    try {
        new Dyer();
    }
    catch (Throwable fail) {
        fail("caught exception in wrong place");
    }
    catch (Error success) {
    }
}

public void testErrorException6() {
    try {
        new Dyer();
    }
```

Exercises

```

    }
    catch (Error fail) {
        fail("caught exception in wrong place");
    }
    catch (Throwable success) {
    }
}

public void testErrorException7() {
    try {
        new Dyer();
    }
    catch (Error fail) {
        fail("caught exception in wrong place");
    }
    catch (Throwable success) {
    }
    finally {
        return;
    }
}

Dyer:
class Dyer {
    Dyer() {
        throw new RuntimeException("oops.");
    }
}

```

7. What compiler error would you expect the following code to generate? Copy it into a test class of your own and make it pass.

```

public void testWithProblems() {
    try {
        doSomething();
        fail("no exception");
    }
    catch (Exception success) {}
}

void doSomething() {
    throw new Exception("blah");
}

```

8. What is wrong with the following commonly seen code?

```

public void doSomething() {
    try {
        complexOperationWithSideEffects();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

Exercises

9. Write a method to log an exception to a given log with the stack trace in reverse order (so that the point of failure is at the tail of the log instead of the head).
10. Create a custom Handler which discards the messages it receives and counts the number of times that messages were logged at various levels of severity. Use a map to store the count by severity.
11. Create a custom log message formatter that you can optionally construct with a CountingLogHandler. If no CountingLogHandler is passed in, the formatter should produce output in the form:

LEVEL: message

For example:

WARNING: watch out

If a CountingLogHandler is passed in, each message should show the count for the current level. For example:

WARNING: watch out (WARNING total = 1)

Make sure you write tests for both circumstances.

Then ensure that the CountingLogHandler uses the custom formatter as its default. Modify the CountingLogHandler to store formatted output in a StringBuilder so that your test can request the complete logging summary.

Finally, edit the logging property file and assign the custom formatter to the ConsoleHandler. Visually ensure that logging messages sent to the ConsoleHandler come out as expected.

Exercises

Lesson 9

Logical
Operators

Maps and Equality

In this lesson you will learn more about the hash table data structure classes in Java. You will also learn about the fundamental concept of equality. Understanding these two fundamental, related topics is critical to your ability to master Java. Yet many developers with years of Java experience do not fully understand these concepts. Insidious defects and severe performance issues can arise from this lack of education. Before we start on these important topics, however, you'll learn about logical operators, an equally important topic.

You will learn about:

- logical operators
- hash tables
- equality
- `toString`
- `Map` and `Set` implementations
- Strings and equality

Logical Operators

Suppose you want to execute a line of code only if two separate conditions both hold true. You can use cascading `if` statements to guard that line of code:

```
if (isFullTime)
    if (isInState)
        rate *= 0.9;
```

Using *logical operators*, you can combine multiple conditionals into a single complex boolean expression:

```
if (isFullTime && isInState)
    rate *= 0.9;
```

The double ampersands symbol (`&&`) is the logical *and* operator. It is a binary operator—it separates two boolean expressions—the operands. If the result of both operands is true, the result of the entire conditional is true. In any other case, the result of the entire conditional is false. Here is the test-driven truth table (TDTT¹) that represents the possible combinations of the logical *and* operator:

```
assertTrue(true && true);
assertFalse(true && false);
assertFalse(false && true);
assertFalse(false && false);
```

The logical *or* operator (`||`), also a binary operator, returns true if either one of the two operands returns true or if both operands return true. The logical *or* operator returns false only if both operands are false. The TDTT for logical *or*:

```
assertTrue(true || true);
assertTrue(true || false);
assertTrue(false || true);
assertFalse(false || false);
```

The logical *not* operator (`!`) is a unary operator that reverse the boolean value of an expression. If the source expression is true, the not operator results in the entire expression returning false, and vice versa. The TDTT:

```
assertFalse(!true);
assertTrue(!false);
```

The logical exclusive-or (*xor*) operator (`^`) is a less frequently used binary operator that returns false if both operands result in the same boolean value. The TDTT:

```
assertFalse(true ^ true);
assertTrue(true ^ false);
assertTrue(false ^ true);
assertFalse(false ^ false);
```

You can combine multiple logical operators. If you do not provide parentheses, the order of precedence is `!`, then `&&`, then `^`, then `||`.

¹A fabricated but cute acronym.

Short-Circuiting

The `&&` and `||` logical operators just shown are known as *short-circuited* logical operators. When the Java VM executes an expression using a short-circuited logical operator, it evaluates the left operand (the expression to the left of the operator) of the expression first. The result of the left operand may immediately determine the result of the entire expression.

In the case of *or* (`||`), if the left operand is `true`, there is no need to evaluate the right operand. The entire expression will always be `true`. The Java VM saves time by executing only half of the expression. With respect to *and* (`&&`), if the left operand is `false`, the entire expression will always be `false`. The Java VM will not evaluate the right operand.

You may have a rare need to always execute both sides of the expression. The code on the right-hand side may do something that you need to always happen. This is poor design. Lesson 4 points out that methods should either effect actions or return information, but not both. Likewise, operands should be viewed as atomic operations that are either commands or queries. You should be able to restructure the code so that the action occurs before executing the complex conditional.

If you still feel compelled to always execute both operands, you can use *non-short-circuited* logical operators. The non-short-circuited *and* operator is a single ampersand (`&`). The non-short-circuited *or* operator is a single pipe (`|`).

Note that *xor* (`^`) is always non-short-circuited. Java cannot discern the result of an xor expression from only one operand. Refer to the TDIT above to see why.

Hash Tables



You need to create a student directory to contain students enrolled in the university. The system must allow each student to be assigned a unique identifier, or ID. The system must allow you to later retrieve each student from the directory using its ID.

To build this story, you will create a `StudentDirectory` class. While you could use an `ArrayList` to store all students, you will instead use a `Map`. You learned a bit about maps in Lesson 6. As a reminder, `Map` is the interface for a data structure that allows elements to be stored and received based on a key. A map is a collection of key-value pairs. You store, or `put`, an element at a certain key in a map. You later retrieve, or `get`, that element by supplying the proper key.

In Lesson 6, you learned about the EnumMap implementation of the Map interface. If the keys in a Map are of enum type, you should use an EnumMap. Otherwise you will usually want to use the workhorse implementation of Map, java.util.HashMap. HashMap is a general-use map based on a data structure known as a hash table. Most Map code you encounter will use the HashMap implementation (or the legacy Hashtable implementation—see Lesson 7).

A hash table provides a specific way to implement a map. It is designed for rapid insertion and retrieval.

To build the student directory, start with the test:

```
package sis.studentinfo;

import junit.framework.*;
import java.io.*;

public class StudentDirectoryTest extends TestCase {
    private StudentDirectory dir;

    protected void setUp() {
        dir = new StudentDirectory();
    }

    public void testStoreAndRetrieve() throws IOException {
        final int numberofStudents = 10;

        for (int i = 0; i < numberofStudents; i++)
            addStudent(dir, i);

        for (int i = 0; i < numberofStudents; i++)
            verifyStudentLookup(dir, i);
    }

    void addStudent(StudentDirectory directory, int i)
        throws IOException {
        String id = "" + i;
        Student student = new Student(id);
        student.setId(id);
        student.addCredits(i);
        directory.add(student);
    }

    void verifyStudentLookup(StudentDirectory directory, int i)
        throws IOException {
        String id = "" + i;
        Student student = dir.findById(id);
        assertEquals(id, student.getLastName());
        assertEquals(id, student.getId());
        assertEquals(i, student.getCredits());
    }
}
```

The test, `testStoreAndRetrieve`, uses a `for` loop to construct and insert a number of students into the `StudentDirectory`. The test verifies, again using a `for` loop, that each student can be found in the directory using the `Student` directory method `findById`.

The `StudentDirectory` code is brief.

```
package sis.studentinfo;

import java.util.*;

public class StudentDirectory {
    private Map<String,Student> students =
        new HashMap<String,Student>();
    public void add(Student student) {
        students.put(student.getId(), student);
    }
    public Student findById(String id) {
        return students.get(id);
    }
}
```

A `StudentDirectory` object encapsulates a `HashMap`, `students`. When client code invokes the `add` method, the code in `add` inserts the `student` argument into the `HashMap`. The key for the inserted student is the student's ID.

The ID must be unique. If you put a new value at an ID for which an entry already exists, the old value is replaced.

Later in this lesson, you will revisit hash tables. First, however, you will learn about equality. And before you can learn about equality, you have a story to implement.

Courses



A `Session` represents the teaching of a course for a given start date. Most courses are taught once per term. Each course may thus be represented by many `Session` instances.²

Currently the `Session` class contains the department and number information for the associated course, as well as its number of credits. If there can be multiple sessions of a course, this implementation is inefficient and trouble-

²This is a small school, so we're ignoring the possibility that there may need to be two separate sections of a session in order to accommodate a large number of students.

Courses

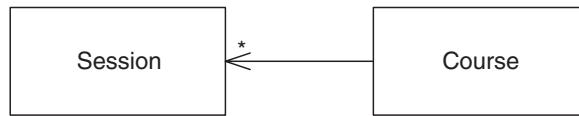


Figure 9.1 A Course Has Many Sessions

some, since the course information will be repeated throughout each Session object.

What you want instead is a way of tying many session objects back to a single Course object. The UML diagram in Figure 9.1 shows this many-to-one relationship from Session to a new class named Course.

You will refactor your code to the above relationship. Start by creating a simple Course class that captures the department, number, and number of credits. A simple test declares that department and number are required to create a Course. The department and number combined represent the unique *key* for a Course. Two separate courses cannot have the same department and number.

```

package sis.studentinfo;

import junit.framework.*;

public class CourseTest extends TestCase {
    public void testCreate() {
        Course course = new Course("CMSC", "120");
        assertEquals("CMSC", course.getDepartment());
        assertEquals("120", course.getNumber());
    }
}
  
```

Course starts out as a simple data object. It contains a constructor and two getters for the key fields.

```

package sis.studentinfo;

public class Course {
    private String department;
    private String number;

    public Course(String department, String number) {
        this.department = department;
        this.number = number;
    }

    public String getDepartment() {
        return department;
    }
}
  
```

```
public String getNumber() {
    return number;
}
```

Refactoring Session

Refactoring Session

Here's an overview of the steps you'll undertake in this refactoring:

1. Construct session objects using a Course object. Impacts: SessionTest, Session, CourseSessionTest, SummerCourseSessionTest, RosterReporterTest, CourseReportTest.
2. Change the creation methods in CourseSession and SummerCourseSession to take a Course as a parameter.
3. Modify CourseSession and SummerCourseSession constructors to take a Course.
4. Modify the Session constructor to take a Course.
5. Store a Course reference in Session instead of department and number strings.

You will make these changes incrementally. With each change, you'll use the compiler to help you find problems, then run your tests to ensure that you didn't break anything.

Starting in SessionTest:

Currently, the abstract method `createSession` constructs Session objects using department and number strings:

```
abstract protected Session createSession(
    String department, String number, Date startDate);
```

You want to change this creation method to reflect the need to construct Session objects using a Course object:

```
abstract public class SessionTest extends TestCase {
    ...
    abstract protected Session createSession(
        Course course, Date startDate);
```

This change will necessitate quite a few other changes, but overall it shouldn't take more than 5 minutes to make them. In SessionTest, you will need to change both the `setUp` method and `testComparable`. Here's the existing `setUp` method in SessionTest (I've boldfaced the code that will need to change):

```
public void setUp() {
    startDate = createDate(2003, 1, 6);
    session = createSession("ENGL", "101", startDate);
    session.setNumberOfCredits(CREDITS);
}
```

I've shown the change to `setUp` here; changes to `testComparable` are similar:

```
protected void setUp() {
    startDate = new Date();
    session = createSession(new Course("ENGL", "101"), startDate);
    session.setNumberOfCredits(CREDITS);
}
```

Use the compiler to guide you through making these changes. As you fix one problem, the compiler can take you to the next problem to fix.

You will need to change both `SummerCourseSessionTest` and `CourseSessionTest`. In the interest of incrementalism, try to make the smallest amount of changes that will get you to a green bar. You can modify both `CourseSessionTest` and `SummerCourseSessionTest` without having to touch `CourseSession` or `SummerCourseSession`.

Here are the changes to the tests. This time I've included the old lines of code from the previous version of the code and struck them out so you could see the difference line to line. I'll use this approach for the remainder of the refactoring.

```
// CourseSessionTest
...
public class CourseSessionTest extends SessionTest {
    public void testCourseDates() {
        Date startDate = DateUtil.createDate(2003, 1, 6);
        Session session = createSession("ENGL", "200", startDate);
        Session session = createSession(createCourse(), startDate);
        Date sixteenWeeksOut = createDate(2003, 4, 25);
        assertEquals(sixteenWeeksOut, session.getEndDate());
    }

    public void testCount() {
        CourseSession.resetCount();
        createSession("", "", new Date());
        createSession(createCourse(), new Date());
        assertEquals(1, CourseSession.getCount());
        createSession("", "", new Date());
        createSession(createCourse(), new Date());
        assertEquals(2, CourseSession.getCount());
    }

    private Course createCourse() {
        return new Course("ENGL", "101");
    }
}
```

```

protected Session createSession(
    String department, String number, Date date) {
    return CourseSession.create(department, number, date);
}

protected Session createSession(Course course, Date date) {
    return CourseSession.create(
        course.getDepartment(), course.getNumber(), date);
}

// SummerCourseSessionTest.java
package sis.summer;

import junit.framework.*;
import java.util.*;
import sis.studentinfo.*;

public class SummerCourseSessionTest extends SessionTest {
    public void testEndDate() {
        Date startDate = DateUtil.createDate(2003, 6, 9);
        Session session = createSession("ENGL", "200", startDate);
        Session session =
            createSession(new Course("ENGL", "200"), startDate);
        Date eightWeeksOut = DateUtil.createDate(2003, 8, 1);
        assertEquals(eightWeeksOut, session.getEndDate());
    }

    protected Session createSession(
        String department, String number, Date date) {
        return SummerCourseSession.create(department, number, date);
    }

    protected Session createSession(Course course, Date date) {
        return SummerCourseSession.create(
            course.getDepartment(), course.getNumber(), date);
    }
}

```

Tests should pass at this point. Now modify the Session subclass creation methods to take a Course directly.

```

// CourseSessionTest.java

protected Session createSession(Course course, Date date) {
    return CourseSession.create(
        course.getDepartment(), course.getNumber(), date);
    return CourseSession.create(course, date);
}

// SummerCourseSessionTest.java
protected Session createSession(Course course, Date date) {
    return SummerCourseSession.create(
        course.getDepartment(), course.getNumber(), date);
    return SummerCourseSession.create(course, date);
}

```

This change impacts RosterReporterTest:

```
package sis.report;

import junit.framework.TestCase;
import sis.studentinfo.*;
import static sis.report.ReportConstant.NEWLINE;

public class RosterReporterTest extends TestCase {
    public void testRosterReport() {
        Session session =
        CourseSession.create(
        "ENGL", "101", DateUtil.createDate(2003, 1, 6)) +
        Session session =
            CourseSession.create(
                new Course("ENGL", "101"),
                DateUtil.createDate(2003, 1, 6));
        ...
    }
}
```

And also CourseReportTest³:

```
package sis.report;

import junit.framework.*;
import java.util.*;
import sis.studentinfo.*;
import static sis.report.ReportConstant.NEWLINE;

public class CourseReportTest extends TestCase {
    public void testReport() {
        final Date date = new Date();
        CourseReport report = new CourseReport();
        report.add(CourseSession.create("ENGL", "101", date));
        report.add(CourseSession.create("CZEC", "200", date));
        report.add(CourseSession.create("ITAL", "410", date));
        report.add(CourseSession.create("CZEC", "220", date));
        report.add(CourseSession.create("ITAL", "330", date));
        report.add(create("ENGL", "101", date));
        report.add(create("CZEC", "200", date));
        report.add(create("ITAL", "410", date));
        report.add(create("CZEC", "220", date));
        report.add(create("ITAL", "330", date));
    }
}
```

³I've taken some liberties by updating both RosterReporterTest and CourseReportTest with Java features you have learned since originally coding them. For example, the code now assigns a CourseSession object to a Session reference. Doing so will also force you to change to RosterReporter and CourseReport.

```

        assertEquals(
            String.format(
                "CZEC 200%n" +
                "CZEC 220%n" +
                "ENGL 101%n" +
                "ITAL 330%n" +
                "ITAL 410%n"),
            report.text());
    }

    private Session create(String name, String number, Date date) {
        return CourseSession.create(new Course(name, number), date);
    }
}

```

Change the creation methods in CourseSession and SummerCourseSession to take a Course as a parameter, but—for the moment—continue to pass along the department and number to the constructor.

```

// CourseSession.java
public static Session create(
    String department, String number, Date startDate) {
    incrementCount();
    return new CourseSession(department, number, startDate);
}

public static Session create(Course course, Date startDate) {
    incrementCount();
    return new CourseSession(
        course.getDepartment(), course.getNumber(), startDate);
}

// SummerCourseSession.java
public static SummerCourseSession create(
    String department, String number, Date startDate) {
    return new SummerCourseSession(department, number, startDate);
}

public static Session create(Course course, Date startDate) {
    return new SummerCourseSession(
        course.getDepartment(), course.getNumber(), startDate);
}

```

Tests pass. Now push the course into the constructor and rerun the tests.

```

// CourseSession.java
public static Session create(Course course, Date startDate) {
    incrementCount();
    return new CourseSession(course, startDate);
}

protected CourseSession(
    String department, String number, Date startDate) {
    super(department, number, startDate);
}

```

```

protected CourseSession(Course course, Date startDate) {
    super(course.getDepartment(), course.getNumber(), startDate);
}

// SummerCourseSession.java
public static Session create(Course course, Date startDate) {
    return new SummerCourseSession(course, startDate);
}

private SummerCourseSession(
    String department, String number, Date startDate) {
    super(department, number, startDate);
}
private SummerCourseSession(Course course, Date startDate) {
    super(course.getDepartment(), course.getNumber(), startDate);
}

```

Tedious? Perhaps. Simple? Each one of the steps in this refactoring is very basic and takes only seconds to execute. Safe? Extremely. The overall time expenditure should only be a few minutes before everything is working. You get to see a green bar several times during the gradual code transformation. The foremost authority on refactoring, Martin Fowler, suggests that you can never run your tests too frequently.⁴

The alternative would be to make the changes all at once and hope they work. You might save a couple of minutes, but there is a likely possibility that you will make a mistake. Correcting the mistake will take you more time than you saved.

You're almost there. Push the Course object into the superclass Session constructor.

```

// CourseSession.java
protected CourseSession(Course course, Date startDate) {
    super(course, startDate);
}

// SummerCourseSession.java
private SummerCourseSession(Course course, Date startDate) {
    super(course, startDate);
}

// Session.java
protected Session(
    String department, String number, Date startDate) {
    this.department = department;
    this.number = number;
    this.startDate = startDate;
}

```

⁴[Fowler2000], p. 94.

```
protected Session(Course course, Date startDate) {
    this.department = course.getDepartment();
    this.number = course.getNumber();
    this.startDate = startDate;
}
```

Equality

Finally, you can change the Session class to store a Course reference instead of the separate department and number String references.

```
// Session.java
abstract public class Session implements Iterable<Student> {
    private String department;
    private String number;
    private Course course;
    // ...
    protected Session(Course course, Date startDate) {
        this.course = course;
        this.startDate = startDate;
    }

    String getDepartment() {
        return department;
        return course.getDepartment();
    }

    String getNumber() {
        return number;
        return course.getNumber();
    }
    // ...
}
```

Equality

 In an upcoming lesson, you will build a course catalog. One requirement for the course catalog will be to ensure that duplicate courses are not added to the catalog. Courses are *semantically equal* if they have the same department and course number.

Now that you have a functional Course class, you will want to override the equals method to supply a semantic definition for equality. If you have two Course objects and the department and number fields in both objects contain the same (corresponding) String, comparing the two should return true.

```
// in CourseTest.java:
public void testEquality() {
```

```
Course courseA = new Course("NURS", "201");
Course courseAPrime = new Course("NURS", "201");
assertEquals(courseA, courseAPrime);
}
```

This test fails. Remember from Lesson 1 that the object `courseA` is a separate object in memory, distinct from `courseAPrime`. The `assertEquals` method translates roughly⁵ to the underlying lines of code:

```
if (!courseA.equals(courseAPrime))
    fail();
```

In other words, the comparison between two objects in `assertEquals` uses the `equals` method defined on the receiver. The receiver is the first parameter, which JUnit refers to as the expected value. The receiver in this example is `courseA`.

You have not yet defined an `equals` method in `Course`. Java therefore searches up the hierarchy chain to find one. The only superclass of `Course` is the implicit superclass `Object`. In `Object` you will find the default definition of `equals`:

```
package java.lang;
public class Object {
    // ...
    public boolean equals(Object obj) {
        return (this == obj);
    }
    // ...
}
```

The `equals` implementation in `Object` compares the reference—the memory location—of the receiver to the reference of the object being passed in.



If you do not provide a definition for `equals` in your subclass, you get the default definition, memory equivalence.

Memory equivalence exists if two references point to the exact same object in memory.

You will want to override the `equals` method in the `Course` object. The `Course` definition of `equals` will override the `Object` definition of `equals`. The `equals` method should return `true` if the receiver object is semantically equal to the parameter object. For `Course`, `equals` should return `true` if the department and course number of the receiver is equal to that of the parameter `Course`.

Let's code this slowly, one step at a time. For now, code the `equals` method in `Course` to most simply pass the test:

⁵The actual code is slightly more involved.

```
@Override
public boolean equals(Object object) {
    return true;
}
```

Equality

The `equals` method must match the signature that `assertEquals` expects. The `assertEquals` method expects to call an `equals` method that takes an `Object` as parameter. If you were to specify that `equals` took a `Course` object as parameter, the `assertEquals` method would not find it:

```
@Override
public boolean equals(Course course) { // THIS WILL NOT WORK
    return true;
}
```

The `assertEquals` method would instead invoke the `Object` definition of `equals`.

Add a comparison to the test that should fail (since `equals` always returns `true`):

```
public void testEquality() {
    Course courseA = new Course("NURS", "201");
    Course courseAPrime = new Course("NURS", "201");
    assertEquals(courseA, courseAPrime);

    Course courseB = new Course("ARTH", "330");
    assertFalse(courseA.equals(courseB));
}
```

Once you see that this test fails, update your `equals` method:

```
@Override
public boolean equals(Object object) {
    Course that = (Course)object;
    return
        this.department.equals(that.department) &&
        this.number.equals(that.number);
}
```

You must first cast the `object` argument to a `Course` reference so that you can refer to its instance variables. The local variable name you give to the argument helps differentiate between the receiver (`this`) and the argument (`that`). The `return` statement compares this `Course`'s department and number to the department and number of that `Course`. If both (`&&`) are equal, the `equals` method returns `true`.

The Contract for Equality

The Java API documentation for the `equals` method in `Object` provides a list of what defines an equivalence relation between two objects:

Reflexivity:	<code>x.equals(x)</code>
Symmetry:	<code>x.equals(y) if-and-only-if (iff) y.equals(x)</code>
Transitivity:	<code>if x.equals(y) and y.equals(z), then x.equals(z)</code>
Consistency:	<code>x.equals(y) returns a consistent value given consistent state</code>
Comparison to null:	<code>!x.equals(null)</code>

The above contract should hold true for all non-null values of `x` and `y`. You can implement each of these rules in your unit test for equality.⁶

```
public void testEquality() {
    Course courseA = new Course("NURS", "201");
    Course courseAPrime = new Course("NURS", "201");
    assertEquals(courseA, courseAPrime);

    Course courseB = new Course("ARTH", "330");
    assertFalse(courseA.equals(courseB));

    // reflexivity
    assertEquals(courseA, courseA);

    // transitivity
    Course courseAPrime2 = new Course("NURS", "201");
    assertEquals(courseAPrime, courseAPrime2);
    assertEquals(courseA, courseAPrime2);

    // symmetry
    assertEquals(courseAPrime, courseA);

    // consistency
    assertEquals(courseA, courseAPrime);

    // comparison to null
    assertFalse(courseA.equals(null));
}
```

⁶A nice set of JUnit extensions, located at <http://sourceforge.net/projects/junit-addons>, provides an automated means of testing the equality contract.

When you run your tests, everything will pass but the final assertion in `testEquality`. You will receive a `NullPointerException`: The `that` argument is `null` and the `equals` code attempts to access its fields (`department` and `object`). You can add a guard clause to return `false` if the argument is `null`:

```
@Override
public boolean equals(Object object) {
    if (object == null)
        return false;
    Course that = (Course)object;
    return
        this.department.equals(that.department) &&
        this.number.equals(that.number);
}
```

You might choose to represent each of the “qualities of equality” (reflexivity, transitivity, symmetry, consistency, and comparison to `null`) in a separate test. There are differing views on approaches to TDD. Some developers feel that you should structure your code so that each test contains a single assertion.⁷ I take the viewpoint that the goal of a test is to prove a single piece of functionality. Doing so may require many assertions/postconditions. In this example, the many assertions you just coded represent a complete contract for a single piece of functionality—equality.

Apples and
Oranges

Apples and Oranges

Since the `equals` method takes an `Object` as parameter, you can pass anything to it (and the code will still compile). Your `equals` method should handle this situation:

```
// apples & oranges
assertFalse(courseA.equals("CMSC-120"));
```

Even though the `String` value matches the `department` and `number` of `courseA`, a `Course` is not a `String`. You would expect this comparison to return `false`. When Java executes the `equals` method, however, you receive a `ClassCastException`. The line of the `equals` method that executes the cast is the cause:

```
@Override
public boolean equals(Object object) {
    if (object == null)
        return false;
    Course that = (Course)object;
```

⁷[Astels2004].

Apples and Oranges

```

        return
        this.department.equals(that.department) &&
        this.number.equals(that.number);
    }
}

```

The code attempts to cast the String parameter to a Course reference. This invalid cast generates the ClassCastException.

You need a guard clause that immediately returns false if someone throws an orange at your equals method. The guard clause ensures that the type of the parameter matches the type of the receiver.

```

@Override
public boolean equals(Object object) {
    if (object == null)
        return false;
    if (this.getClass() != object.getClass())
        return false;
    Course that = (Course)object;
    return
        this.department.equals(that.department) &&
        this.number.equals(that.number);
}

```

You learned about the getClass method in Lesson 8. It returns a class constant. For the example comparison that threw the ClassCastException, the class constant of the receiver is Course.class, and the class constant of the parameter is String.class. Class constants are unique—there is only one “instance” of the Class object Course.class. This uniqueness allows you to compare class constants using the operator != (not equal to) instead of comparing them using equals.

Some developers choose to use the instanceof operator. The instanceof operator returns true if an object is an instance of, or subclass of, the target class. Here is the equals method rewritten to use instanceof.

```

@Override
public boolean equals(Object object) {
    if (object == null)
        return false;
    if (!(object instanceof Course))
        return false;
    Course that = (Course)object;
    return
        this.department.equals(that.department) &&
        this.number.equals(that.number);
}

```

You should initially prefer to compare the classes of the receiver and argument, instead of using instanceof. The class comparison only returns true if both objects are of the exact same type. Introduce instanceof later if you need to

compare objects irrespective of their position in an inheritance hierarchy. See Lesson 12 for more information on `instanceof`.

JUnit and Equality

Sometimes I will code `equals` tests to be a bit more explicit. Instead of saying:

```
assertEquals(courseA, courseB);
```

I'll express the comparison as:

```
assertTrue(courseA.equals(courseB));
```

You should be clear on the fact that `assertEquals` uses the `equals` method for reference comparisons. But it can be helpful to explicitly emphasize that fact, and show that the `equals` method is what you're actually testing.

You may want to compare two references to see if they are the same. In other words, do they point to the same object in memory? You could code this comparison as:

```
assertTrue(courseA == courseB);
```

or as:

```
assertSame(courseA, courseB);
```

You should prefer the second assertion form, since it supplies a better error message upon failure. As before, though, you might choose to explicitly show the `==` comparison if for purposes of building an `equals` method test.

Collections and Equality

The student information system will need to create a course catalog that contains all of the `Course` objects. You will thus need to be able to store a `Course` in a collection, then verify that the collection contains the object.

```
package sis.studentinfo;

import junit.framework.*;
import java.util.*;

public void testEquality() {
    Course courseA = new Course("NURS", "201");
    Course courseAPrime = new Course("NURS", "201");
    assertEquals(courseA, courseAPrime);
```

Hash Tables

```

    ...
    // containment
    List<Course> list = new ArrayList<Course>();
    list.add(courseA);
    assertTrue(list.contains(courseAPrime));
}
```

The `contains` method defined in `ArrayList` loops through all contained objects, comparing each to the parameter using the `equals` method. This test should pass with no further changes. In fact, the test for containment is more of a language test. It tests the functionality of `ArrayList`, by demonstrating that `ArrayList` uses the `equals` method for testing containment. The containment test does not add to the equality contract. Feel free to remove it.

What about using the `Course` as a key in a `HashMap` object?

```

public void testEquality() {
    Course courseA = new Course("NURS", "201");
    Course courseAPrime = new Course("NURS", "201");

    // ...

    Map<Course, String> map = new HashMap<Course, String>();
    map.put(courseA, "");
    assertTrue(map.containsKey(courseAPrime));
}
```

The above bit of test demonstrates a `Map` method, `containsKey`. This method returns `true` if a matching key exists in the map. Even though a `Map` is a collection, just as a `List` is, this test fails! To understand why, you must understand how the `HashMap` class is implemented.

Hash Tables

An `ArrayList` encapsulates an array, a contiguous block of space in memory. Finding an object in an array requires serially traversing the array in order to find an element. The class `java.util.HashMap` is built around the hash table construct mentioned at the beginning of this lesson. It too is based on a contiguous block of space in memory.

I can pictorially represent a hash table as a bunch of slots (Figure 9.2).

When inserting an element into a hash table, the code rapidly determines a slot for the element by first asking it for its *hash code*. A hash code is simply an integer, ideally as unique as possible. The contract for hash code is based on the definition of equality for a class: If two objects are equal, their hash

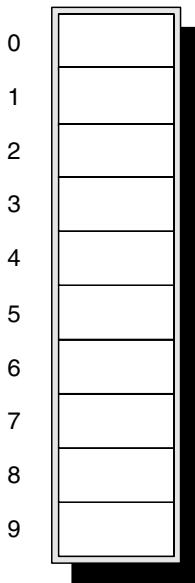


Figure 9.2 A Hash Table with Ten Slots

code should be equal. If two objects are not equal, then their hash codes are ideally (but not necessarily) unequal.

Once the hash code is available, a bit of arithmetic determines the slot:

```
hash code % table size = slot number
```

(Remember that the modulus operator, `%`, returns the remainder of the equivalent integer division.) For example, if you insert a Course object into a hash table of size 10 and the hash code value of the course is 65, it would go into the fifth slot:

$$65 \% 10 = 5$$

Just as in an array, Java calculates the memory address for the start of a slot using the formula:

```
offset + (slot size * slot number)
```

Figure 9.3 shows a Course element slotted into a hash table.

The hash code is an `int` that is retrieved by sending the message `hashCode` to any `Object`. The class `java.lang.Object` supplies a default definition of `hashCode`. It returns a unique number based upon the memory address of the object.

To retrieve an object from a hash table, you recalculate the slot by asking for the object's `hashCode` again. You can then do the same offset calculation to immediately retrieve the object.

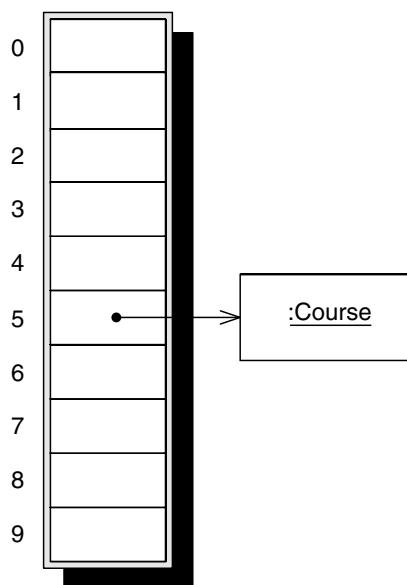
Collisions

Figure 9.3 Hash Table with Course in the Fifth Slot

So what goes in a `hashCode` method? It must return an `int`, and as mentioned, two equal objects must return the same `hashCode`. The simplest, but worst, solution would be to return a constant, such as the number 1.

Collisions

The hash code for an object should be as unique as possible. If two unequal objects return the same hash code, they will resolve to the same slot in the hash table. This is known as a *collision*. The implication of a *collision* is extra logic and extra time to maintain a list of collided objects. A few solutions for resolving collisions exist. The simplest is manage a list of collided objects for each slot. Figure 9.4 shows a pictorial representation.

Since the possibility of collisions exists, when retrieving an object, the hash table code must also compare the object retrieved from a slot to ensure that it is the one expected. If not, the code must traverse the list of collided objects to find the match.

If all objects were to return the same `hashCode`, such as 1, then all objects will collide to the same slot in a `HashMap`. The end result would be that every insertion and deletion would require serially traversing the list of collided

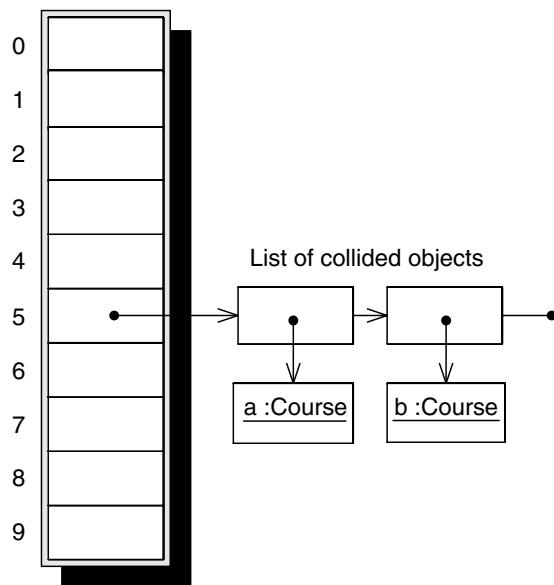


Figure 9.4 Hash Table Collisions

objects. This list would include every object inserted. In this degenerate case, you might as well use an `ArrayList`.

If the hash codes of all objects generate different slots, you have an optimally mapped hash table. Performance is the best possible: All insertions to and retrievals from the table execute in constant time. There is no need to serially traverse a collision list.

An Ideal Hash Algorithm

Java already provides a good `hashCode` implementation for the class `String`, which is more often than not the key type for a `HashMap`. The integral numeric wrapper classes (`Integer`, `Long`, `Byte`, `Char`, `Short`) are also often used as a `HashMap` key; their hash code is simply the number they wrap.

The fields that uniquely identify your object are typically one of these object types (`String` or numeric). To obtain a decent hash code that satisfies the equality condition, you can simply return the hash code of the unique key. If more than one field is used to represent the unique key, you will need to devise an algorithm to generate the hash code.

The following code, to be added to the `Course` class, generates a `hashCode` based on arithmetically combining the `hashCodes` of `department` and `number`. Hash

An Ideal Hash Algorithm

code algorithms typically use prime numbers. The use of primes tends to give better distribution when combined with the modulus operation against the table size.

```
@Override
public int hashCode() {
    final int hashMultiplier = 41;
    int result = 7;
    result = result * hashMultiplier + department.hashCode();
    result = result * hashMultiplier + number.hashCode();
    return result;
}
```

The tests should pass once you implement this `hashCode` method.⁸ Separate the hash code tests into a new test method that will enforce the complete contract for `hashCode`.

```
public void testHashCode() {
    Course courseA = new Course("NURS", "201");
    Course courseAPrime = new Course("NURS", "201");

    Map<Course, String> map = new HashMap<Course, String>();
    map.put(courseA, "");
    assertTrue(map.containsKey(courseAPrime));

    assertEquals(courseA.hashCode(), courseAPrime.hashCode());
    // consistency
    assertEquals(courseA.hashCode(), courseA.hashCode());
}
```

The test proves that two semantically equal courses produce the same hash code. It also demonstrates consistency. The return value from `hashCode` is the same (consistent) with each call to it, as long as the key values for the courses do not change.

The part of the test that demonstrates stuffing the `Course` into a hash map can be removed, since the contract assertions for `hashCode` are all that is required. The final set of equality and hash code contract tests:

```
public void testEquality() {
    Course courseA = new Course("NURS", "201");
    Course courseAPrime = new Course("NURS", "201");
    assertEquals(courseA, courseAPrime);

    Course courseB = new Course("ARTH", "330");
    assertFalse(courseA.equals(courseB));

    // reflexivity
    assertEquals(courseA, courseA);
```

⁸This algorithm was obtained from the example at <http://mindprod.com/jgloss/hashcode.html>.

```

// transitivity
Course courseAPrime2 = new Course("NURS", "201");
assertEquals(courseAPrime, courseAPrime2);
assertEquals(courseA, courseAPrime2);

// symmetry
assertEquals(courseAPrime, courseA);

// consistency
assertEquals(courseA, courseAPrime);

// comparison to null
assertFalse(courseA.equals(null));

// apples & oranges
assertFalse(courseA.equals("CMSC-120"));
}

public void testHashCode() {
    Course courseA = new Course("NURS", "201");
    Course courseAPrime = new Course("NURS", "201");

    assertEquals(courseA.hashCode(), courseAPrime.hashCode());
    // consistency
    assertEquals(courseA.hashCode(), courseA.hashCode());
}

```

A Final Note on
hashCode

These tests contain a good amount of code! Again, consider JUnit-addons (see footnote 6 above) as a way of automatically enforcing equality and hash code contracts. If you choose to not use JUnit-addons, make sure you refactor to eliminate duplication when writing these tests.

A Final Note on hashCode



If you code an `equals` method for a class, also code a `hashCode` method.

In the absence of `hashCode`, the compiler won't complain, and you may not encounter any problems. However, unexpected behavior will likely result if you put objects of that class into a hash table based collection (the class `java.util.Set` also uses a hash table implementation). Figuring out that there is a missing `hashCode` method can waste a considerable amount of time. Get in the habit of coding `hashCode` if you code `equals`!

If performance is a consideration, you may want to write unit tests for the hash code. One technique is to load a `HashSet` with a large number of elements, testing the performance to ensure that the insertions execute in a reasonable time. If not, the hash table probably managed a high number of collisions.

A Final Note on hashCode

```
public void testHashCodePerformance() {
    final int count = 10000;
    long start = System.currentTimeMillis();
    Map<Course, String> map = new HashMap<Course, String>();
    for (int i = 0; i < count; i++) {
        Course course = new Course("C" + i, "" + i);
        map.put(course, "");
    }
    long stop = System.currentTimeMillis();
    long elapsed = stop - start;
    final long arbitraryThreshold = 200;
    assertTrue("elapsed time = " + elapsed,
               elapsed < arbitraryThreshold);
}
```

There are many ways to test performance, but the above test provides a simple, adequate mechanism. The class `java.lang.System` contains a static method, `currentTimeMillis`, that returns the milliseconds representing the current time stamp. Before starting, you store the current timestamp in a local variable (`start`). You execute the desired code *n* times using a `for` loop. When the loop completes, you capture the timestamp (`stop`). You obtain the elapsed time by subtracting the start time from the stop time. The assertion ensures that the loop completed in a reasonable amount of time (`arbitraryThreshold`).

The `assertTrue` method call contains a `String` as its first parameter. The `String` gets printed by JUnit as part of the error message only if the assertion fails. In this example, the failure will show how long the loop actually took to execute.

You may want to isolate performance tests in a separate suite. You will still want to run them regularly (perhaps daily, or after you have completed a significant task). The nature of performance tests makes them slow, however, so you may not want the constant negative impact to the performance of your functional unit test suite.

If you find you need more comprehensive unit testing, either build your own framework by eliminating duplication or consider a tool such as Junit-Perf.⁹ JUnit itself contains some rudimentary hooks to help you with repetitive testing.

To demonstrate the usefulness of the example performance test, change the `hashCode` implementation in `Course` to return a constant number:

```
@Override
public int hashCode() {
    return 1;
}
```

⁹See <http://www.clarkware.com/software/JUnitPerf.html>.

Even with only 10,000 entries into the `HashMap`, the test should fail miserably. Revert the `hashCode` to its previous form; the test should now pass.

Another technique for testing `hashCode` is to ensure that the variance that a number of hash codes produces is reasonably high. The variance is a measure of how spread out a distribution of numbers is. You calculate the variance as the average of the squared deviation of each number from the mean.

The problem with both of these testing techniques is that they are based on arbitrary measures of what is acceptable. As a developer, you can determine the value of this kind of testing and choose to defer it until you can pinpoint a true performance problem to the hash table.

Minimally, you should at least write enough tests to prove that the contract for `hashCode` holds true.

More on Using HashMaps

Like a list, you will often want to iterate through the contents of a hash table. Your needs may include iterating through only the keys, through only the values or through both simultaneously.

In Lesson 6, you built a `ReportCard` class. You used `ReportCard` to store a map of grades to corresponding messages for printing on a report card. As a reminder, here is the source for `ReportCard`:

```
package sis.report;

import java.util.*;
import sis.studentinfo.*;

public class ReportCard {
    static final String A_MESSAGE = "Excellent";
    static final String B_MESSAGE = "Very good";
    static final String C_MESSAGE = "Hmmm...";
    static final String D_MESSAGE = "You're not trying";
    static final String F_MESSAGE = "Loser";

    private Map<Student.Grade, String> messages = null;

    public String getMessage(Student.Grade grade) {
        return getMessages().get(grade);
    }

    private Map<Student.Grade, String> getMessages() {
        if (messages == null)
            loadMessages();
        return messages;
    }
}
```

```

private void loadMessages() {
    messages =
        new EnumMap<Student.Grade, String>(Student.Grade.class);
    messages.put(Student.Grade.A, A_MESSAGE);
    messages.put(Student.Grade.B, B_MESSAGE);
    messages.put(Student.Grade.C, C_MESSAGE);
    messages.put(Student.Grade.D, D_MESSAGE);
    messages.put(Student.Grade.F, F_MESSAGE);
}
}

```

The following three new tests for ReportCard demonstrate how to iterate through each of the three possibilities. (These are unnecessary tests, and act more like language tests.) Each of the three tests show different but valid testing approaches.

```

package sis.report;

import junit.framework.*;
import sis.studentinfo.*;
import java.util.*;

public class ReportCardTest extends TestCase {
    private ReportCard card;

    protected void setUp() {
        card = new ReportCard();
    }

    public void testMessage() {
        // remove declaration of card since it is declared in setUp
        ...
    }

    public void testKeys() {
        Set<Student.Grade> expectedGrades = new HashSet<Student.Grade>();
        expectedGrades.add(Student.Grade.A);
        expectedGrades.add(Student.Grade.B);
        expectedGrades.add(Student.Grade.C);
        expectedGrades.add(Student.Grade.D);
        expectedGrades.add(Student.Grade.F);

        Set<Student.Grade> grades = new HashSet<Student.Grade>();
        for (Student.Grade grade: card.getMessages().keySet())
            grades.add(grade);
        assertEquals(expectedGrades, grades);
    }
}

```

The first test, `testKeys`, provides one technique for ensuring that you iterate through all expected values in a collection. You first create a set and populating it with the objects (grades in this example) you expect to receive. You

**More on Using
HashMaps**

then create a secondary set and add to it each element that you iterate through. After iteration, you compare the two sets to ensure that they are equal.

Perhaps you remember sets from grade school. (Remember Venn diagrams?) If not, no big deal. A set is an unordered collection that contains unique elements. If you attempt to add a duplicate element to a set, the set rejects it. In Java, the `java.util.Set` interface declares the behavior of a set. The class `java.util.HashSet` is the workhorse implementation of the `Set` interface. The `HashSet` class compares objects using `equals` to determine if a candidate object is a duplicate.

Based on your understanding of how a hash table works, it should be clear that you cannot guarantee that its keys will appear in any specific order. It should also be clear that each key is unique. You cannot have two entries in a hash table with the same key. As such, when you send the message `keySet` to a `HashMap` object, it returns the keys to you as a set.

As shown in `testKeys`, you iterate a `Set` much as you iterate a `List`, using a `for-each` loop.

In contrast, the values in a hash table can be duplicates. For example, you might choose to print the same message (“get help”) for a student with a D or an F. A `HashMap` thus cannot return the values as a `Set`. Instead, it returns the values as a `java.util.Collection`.

`Collection` is an interface implemented by both `HashSet` and `ArrayList`. The `Collection` interface declares basic collection functionality, including the abilities to add objects to a collection and obtain an iterator from a collection. The `Collection` interface does not imply any order to a collection, so it does not declare methods related to indexed access as does the `List` interface. Both `java.util.Set` and `java.util.List` interfaces extend the `Collection` interface.

```
public void testValues() {
    List<String> expectedMessages = new ArrayList<String>();
    expectedMessages.add(ReportCard.A_MESSAGE);
    expectedMessages.add(ReportCard.B_MESSAGE);
    expectedMessages.add(ReportCard.C_MESSAGE);
    expectedMessages.add(ReportCard.D_MESSAGE);
    expectedMessages.add(ReportCard.F_MESSAGE);

    Collection<String> messages = card.getMessages().values();
    for (String message: messages)
        assertTrue(expectedMessages.contains(message));
    assertEquals(expectedMessages.size(), messages.size());
}
```

The test, `testValues`, uses a slightly different technique. You again create a `Set` to represent expected values, then add to it each possible value. You

**More on Using
HashMaps**

obtain the hash table values by sending the message values to the `HashMap` object.

As you iterate through the values, you verify that the list of expected values contains each element. The `contains` method uses the `equals` method to determine if the collection includes an object. Finally, you must also test the size of the secondary set to ensure that it does not contain more elements than expected.

The third test, `testEntries`, shows how you can iterate through the keys and associated values (the maps) simultaneously. If you send the message `entrySet` to a `HashMap`, it returns a `Set` of key-value pairs. You can iterate through this set using a `for-each` loop. For each pass through the loop you get a `Map.Entry` reference that stores a key-value pair. You can retrieve the key and value from a `Map.Entry` by using the methods `getKey` and `getValue`. The `Map.Entry` is bound to the same key type (`Student.Grade` in the example) and value type (`String`) as the `HashMap` object.

```
public void testEntries() {
    Set<Entry> entries = new HashSet<Entry>();

    for (Map.Entry<Student.Grade, String> entry:
        card.getMessages().entrySet())
        entries.add(
            new Entry(entry.getKey(), entry.getValue()));

    Set<Entry> expectedEntries = new HashSet<Entry>();
    expectedEntries.add(
        new Entry(Student.Grade.A, ReportCard.A_MESSAGE));
    expectedEntries.add(
        new Entry(Student.Grade.B, ReportCard.B_MESSAGE));
    expectedEntries.add(
        new Entry(Student.Grade.C, ReportCard.C_MESSAGE));
    expectedEntries.add(
        new Entry(Student.Grade.D, ReportCard.D_MESSAGE));
    expectedEntries.add(
        new Entry(Student.Grade.F, ReportCard.F_MESSAGE));

    assertEquals(expectedEntries, entries);
}
```

You need to test that the `for-each` loop iterates over all the entries. You can create a secondary set of expected entries, like in the previous tests, and compare this expected set to the set populated by iteration. To the expected set you need to add the appropriate key-value entries, but unfortunately, `Map.Entry` is an interface. No concrete class to store a key-value pair is available to you.

Instead, you will create your own `Entry` class to store a key-value pair. The process of iteration will instantiate `Entry` objects and add them to the “actu-

als” set. Also, you will populate the expected set with Entry objects. You can then compare the two sets to ensure that they contain equal Entry objects.

```
class Entry {
    private Student.Grade grade;
    private String message;
    Entry(Student.Grade grade, String message) {
        this.grade = grade;
        this.message = message;
    }

    @Override
    public boolean equals(Object object) {
        if (object.getClass() != this.getClass())
            return false;
        Entry that = (Entry)object;
        return
            this.grade == that.grade &&
            this.message.equals(that.message);
    }

    @Override
    public int hashCode() {
        final int hashMultiplier = 41;
        int result = 7;
        result = result * hashMultiplier + grade.hashCode();
        result = result * hashMultiplier + message.hashCode();
        return result;
    }
}
```

The class `Entry` is a quick-and-dirty class used solely for testing. I therefore chose to write no tests for it. The `Entry` class contains an `equals` method and its companion method `hashCode`. Since this is a quick-and-dirty test class, I might have chosen to omit the `hashCode` method, but a successful test requires it. Try running the tests without `hashCode`, and make sure you understand why the test fails without it.

Additional Hash Table and Set Implementations

The package `java.util` includes several additional classes that implement the `Map` and `Set` interfaces. I'll briefly overview some of these. Consult the Java API documentation for the package `java.util` for more details on their use.

EnumSet

You learned about EnumMap in Lesson 6. The EnumSet class works a little differently. It is defined as an abstract class. You create an instance of an (unnamed) EnumSet subclass by calling one of almost a dozen EnumSet class methods. I summarize these factory methods in Table 9.1.

Using EnumSet, you can simplify the ReportCardTest method `testKeys` (above) as follows:

```
public void testKeys() {
    Set<Student.Grade> expectedGrades =
        EnumSet.allOf(Student.Grade.class);
    Set<Student.Grade> grades =
        EnumSet.noneOf(Student.Grade.class);
    for (Student.Grade grade: card.getMessages().keySet())
        grades.add(grade);
    assertEquals(expectedGrades, grades);
}
```

TreeSet and TreeMap

A TreeSet maintains the order of its elements based on their natural sorting order (that you define using Comparable) or a sort order you define using a Comparator object. A TreeMap maintains the order of its keys in a similar manner. As an example, you might choose to ensure that the report card messages stay ordered by student grade. You pay for this behavior. Instead of

Table 9.1 *EnumSet Factory Methods*

Method	Description
<code>allOf</code>	creates an EnumSet using all possible elements of the specified enum type
<code>complementOf</code>	creates an EnumSet as the complement of another EnumSet—i.e., the new EnumSet contains all enum instances not contained by the source EnumSet
<code>copyOf (+1 variant)</code>	creates an EnumSet using elements from an existing collection
<code>noneOf</code>	creates an empty EnumSet of the specified type
<code>of (+4 variants)</code>	creates an EnumSet with up to five different initial values
<code>range</code>	

almost immediate insertion and retrieval time, as you expect when using a `HashMap` or `HashSet`, operation times increase logarithmically as the size of the collection increases.

LinkedHashSet and LinkedHashMap

A `LinkedHashSet` maintains the order of its elements chronologically: It remembers the order in which you inserted elements. It does so by maintaining a linked list behind the scenes. Its performance is comparable to that of `HashSet`: `add`, `contains`, and `remove` all execute in constant time but with slightly more overhead. Iterating against a `LinkedHashSet` can actually execute faster than iterating against a regular `HashSet`. The Map analog is `LinkedHashMap`.

IdentityHashMap

An `IdentityHashMap` uses reference equality (`==`) to compare keys instead of equality (`equals`). You would use this Map implementation if you needed keys to represent unique instances. This class is not a general-purpose map implementation since it violates the contract that Maps should compare keys using `equals`.

toString

You have seen how you can send the `toString` message to a `StringBuilder` object in order to extract its contents as a `String`.

The `toString` method is one of the few methods that the Java designers chose to define in `java.lang.Object`. Its primary purpose is to return a printable (`String`) representation of an object. You could use this printable representation as part of an end-user application, but don't. The `toString` method is more useful for developers, to help in debugging or deciphering a JUnit message.

You want objects of the `Course` class to display a useful string when you print them out in a JUnit error message. One way would be to extract its elements individually and produce a concatenated string. Another way is to use the `toString` method.

```
public void testToString() {  
    Course course = new Course("ENGL", "301");  
    assertEquals("ENGL 301", course.toString());  
}
```

toString

Since `toString` is inherited from `Object`, you get a default implementation that fails the test:

```
expected: <ENGL 301> but was: <studentinfo.Course@53742e9>
```

This is the implementation that passes the test:

```
@Override
public String toString() {
    return department + " " + number;
}
```

Many IDEs will display objects in watch or inspection windows by sending them the `toString` message. JUnit uses `toString` to display an appropriate message upon failure of an `assertEquals` method. If you code:

```
assertEquals(course, new Course("SPAN", "420"));
```

then JUnit will now display the following `ComparisonFailure` message:

```
expected: <ENGL 301> but was: <SPAN 420>
```

You can concatenate an object to a `String` by using the `+` operator. Java sends the `toString` message to the object in order to “convert” it to a `String` object. For example:

```
assertEquals("Course: ENGL 301", "Course: " + course);
```

The default implementation of `toString` is rarely useful. You will probably want to supply a more useful version in most of your subclasses. If you only use the `toString` implementation for the purpose of debugging or understanding code, writing a test is not necessary.

In fact, `toString`’s definition is often volatile. It depends upon what a developer wants to see as a summary string at a given time. Creating an unnecessary `toString` test means that you have made the production class a little harder to change.

One of the main reason you write tests is to allow you to change code with impunity, without fear. For each test that is missing, your confidence in modifying code diminishes. Writing tests enables change. The irony is that the more tests you have, the more time-consuming it will be to make changes.

You should never use the difficulty of maintaining tests as an excuse to not write tests. But you should also not write tests for things that are unnecessary to test. You should also code your tests so that they are as resistant as possible to change. Finding the right balance is a matter of experience and good judgment.

Ultimately, it's your call. My usual recommendation is to start out with more tests than necessary. Cut back only when they become a problem that you can't fix through other means.

Strings and Equality

Strings and Equality

The String class is optimized for heavy use. In most applications, the number of String objects created vastly outweighs the number of any other type being created.

The current version of the Java VM provides a set of command-line switches to turn on various types of profiling. You can enter:

```
java -agentlib:hprof=help
```

at the command line for further details.¹⁰

I ran a heap profile against the JUnit run of the current suite of almost 60 tests. The Java VM created about 19,250 objects in the fraction of the second required to run the tests. Out of those objects, over 7,100 were String objects. The object count for the next most frequently instantiated type was around 2,000. Counts dropped off dramatically after that.

These numbers demonstrate the key role that the String class plays in Java. String objects will account for a third to over a half of all objects created during a typical application execution. As such, it is critical that the String class perform well. It is also critical that the creation of so many strings does not bloat the memory space.

To minimize the amount of memory used, the String class uses a literal pool. The main idea is that if two String objects contain the same characters, they share the same memory space (the "pool"). The String literal pool is an implementation of a design pattern called Flyweight.¹¹ Flyweight is based on sharing and is designed for efficient use of large numbers of fine-grained objects.

To demonstrate:

```
String a = "we have the technology";
String b = "we have the technology";
assertTrue(a.equals(b));
assertTrue(a == b);
```

¹⁰This feature is not guaranteed to be part of future versions of Java.

¹¹[Gamma1995].

Exercises

The strings `a` and `b` are semantically equal (`.equals`): Their character content is the same. The string references also have memory equivalence (`==`): They point to the same `String` object in memory.

Because of this optimization, as a beginning Java programmer it is easy to think you should compare `String` objects using `==`.

```
if (name == "David Thomas")
```

But this is almost always a mistake. It is possible for two strings to contain the same characters but not be stored in the same memory location. This occurs because Java cannot optimize `Strings` that are constructed through the use of `StringBuilder` or concatenation.¹²



Compare Strings using `equals`, not `==`.

```
String c = "we have";
c += " the technology";
assertTrue(a.equals(c));
assertFalse(a == c);
```

This test passes. The Java VM stores `c` in a different memory location than `a`.

You might be able to compare `String` objects using `==` in certain controlled circumstances in your code. Doing so can provide a boost in performance. However, you should use `==` for a `String` comparison only as a performance optimization and document it as such with explicit comments. Being clever and using memory equivalence comparisons for `String` in other circumstances is a great way to invite insidious bugs that are difficult to track down.

Exercises

1. Create a `String` literal using the first two sentences of this exercise. You will create a `WordCount` class to parse through the text and count the number of instances of each word. Punctuation should not appear in the word list, either as part of a word or as a separate “word.” Use a map to store the frequencies. The `WordCount` class should be able to return a set of strings, each showing the word and its frequency. Track the frequency without respect to case. In other

¹²When compiled, code using the `+` for `String` concatenation translates to equivalent code that uses the `StringBuffer` class.

words, two words spelled the same but with different case are the same word. Hint: You can use the regular expression \w+ in conjunction with the String split method in order to extract the words from the String.

2. Create a class, Name, that declares a single string field. Create an equals method on the class that uses the string for comparisons. Do not create a hashCode method. Build a test that verifies the contract of equality for Name.
3. Create a Set<Name> object that contains a variety of Name objects, including new Name("Foo"). Verify that the Set contains method returns false if you ask the Set whether or not it contains the Name("Foo") instance. Show that if you create:

```
Name foo = new Name("Foo");
```

the set *does* contain foo.

4. Modify the test for Name (see the previous question) to pass only if the list *does* contain new Name("Foo").

This page intentionally left blank

Lesson 10

Mathematics

Mathematics

In this lesson you will learn about the mathematical capabilities of Java. Java is designed as a multipurpose language. In addition to standard, general-purpose arithmetic support, Java supports a small number of more-advanced math concepts such as infinity. Support is provided through a mixture of Java language features and library classes.

Java supplies a wide range of floating point and integral numerics in both fixed and arbitrary precision form. The representation of numerics and related functions in Java generally adheres to IEEE standards. In some cases, you can choose to use either algorithms designed for speed or algorithms whose results strictly adhere to published standards.

Topics:

- BigDecimal
- additional integral types and operations
- numeric casting
- expression evaluation order
- NaN (Not a Number)
- infinity
- numeric overflow
- bit manipulation
- java.lang.Math
- static imports

BigDecimal

BigDecimal

The class `java.math.BigDecimal` allows you to do arbitrary-precision decimal (base 10) floating-point arithmetic. This means that arithmetic with `BigDecimal` works like you learned in grade school. The `BigDecimal` class provides a comprehensive set of arithmetic methods and extensive control over rounding. A `BigDecimal` object represents a decimal number of arbitrary precision, meaning that you decide the number of significant digits it will contain. It is immutable—you cannot change the number that a `BigDecimal` stores.

The most frequent use of `BigDecimal` is in financial applications. Financial applications often require you to accurately account for money down to the last hundredth of a cent or even at more granular levels.

Floating-point numbers, as implemented by the Java primitive types `float` and `double`, do not allow every number to be represented exactly as specified. This is because it is not mathematically possible to represent certain decimal numbers (such as 0.1) using binary.¹ `BigDecimal` *can* exactly represent every number.

There are two major downsides to using `BigDecimal`. First, hardware normally optimizes binary floating-point operations, but `BigDecimal`'s decimal floating point is implemented in software. You may experience poor performance when doing extremely heavy math calculations with `BigDecimal`.

Second, there is no mathematical operator support for `BigDecimal`. You must execute common operations such as addition and multiplication via method calls. The requisite code is wordy and thus tedious to write and read.

Using BigDecimal



The student information system must maintain an account of charges and credits for each student. The first test demonstrates the ability of an `Account` to track a balance based on applied charges and credits.

```
package sis.studentinfo;

import java.math.BigDecimal;
import junit.framework.*;

public class AccountTest extends TestCase {
    public void testTransactions() {
        Account account = new Account();
```

¹See <http://www.alphaworks.ibm.com/aw.nsf/FAQs/bigdecimal>.

```

        account.credit(new BigDecimal("0.10"));
        account.credit(new BigDecimal("11.00"));
        assertEquals(new BigDecimal("11.10"), account.getBalance());
    }
}

```

Big Decimal

The preferred way to construct a new `BigDecimal` is to pass a String to its constructor. The String contains the value you want the `BigDecimal` to represent. You could also pass a `double`, but it may not precisely represent the number you expect, due to the nature of floating-point representation in Java.² As a result, the `BigDecimal` may not precisely represent the number you expect.

You may also construct `BigDecimal` objects from other `BigDecimal` objects. This allows you to directly use an expression returning a `BigDecimal` as the constructor parameter.

The Account implementation:

```

package sis.studentinfo;

import java.math.BigDecimal;

public class Account {
    private BigDecimal balance = new BigDecimal("0.00");

    public void credit(BigDecimal amount) {
        balance = balance.add(amount);
    }

    public BigDecimal getBalance() {
        return balance;
    }
}

```

`BigDecimal` objects are immutable. Sending an `add` message to a `BigDecimal` does not alter it. Instead, the `add` method takes the argument's value and adds it to the value the `BigDecimal` stores. The `add` method then creates and returns a *new* `BigDecimal` object with this sum. In the Account implementation, the result of sending the `add` message to `balance` is a new `BigDecimal` object. You must reassign this new object to `balance` in order to maintain the total.

`BigDecimal` provides a full complement of methods to represent arithmetic operations. They include `abs`, `add`, `divide`, `max`, `min`, `multiply`, `negate`, and `subtract`. Additional methods help you to manage scaling and to extract values as different types from a `BigDecimal`.

²See the Java Glossary entry at <http://mindprod.com/jgloss/floatingpoint.html> for a discussion of floating-point representation.

BigDecimal

Scale

In `testTransactions`, the assertion expects the result to be a `BigDecimal` with value "11.10"—with two places after the decimal point. In contrast, you would expect the result of an equivalent Java `float` or `double` operation to be "11.1". Adding such an assertion:

```
assertEquals(new BigDecimal("11.1"), account.getBalance());
```

fails the test.

The number of digits in the fractional part represents the *scale* of the number. When one does arithmetic operations on `BigDecimal` numbers, the result is a new `BigDecimal` whose scale is the larger between the receiving `BigDecimal` and the parameter `BigDecimal`. For example:

```
assertEquals(new BigDecimal("5.300"),
    new BigDecimal("5.000").add(new BigDecimal("0.3")));
```

That is, the scale as the result of adding a 1-scale number to a 3-scale number is 3.

Division and Rounding

When dividing numbers, the result often contains a larger number of fractional digits than either the divisor or the dividend. By default, `BigDecimal` does not expand the scale of the result to this larger amount; instead, it restricts it to the larger of the dividend and divisor scales. You can also explicitly define a new scale for the result.

 Constraining the scale means that Java may need to round the result of a division operation. Java provides eight different rounding modes. You are probably most familiar with the rounding mode that corresponds to `BigDecimal.ROUND_HALF_UP`. This mode tells `BigDecimal` to round the result up if the distance toward the nearest neighbor is greater than or equal to 0.5, and to round it down otherwise. For example, 5.935 would round up to the 2-scale number 5.94, and 5.934 would round down to 5.93.

For an account, you must be able to capture the average transaction amount. A test demonstrates this need:

```
public void testTransactionAverage() {
    Account account = new Account();
    account.credit(new BigDecimal("0.10"));
    account.credit(new BigDecimal("11.00"));
    account.credit(new BigDecimal("2.99"));
    assertEquals(new BigDecimal("4.70"), account.transactionAverage());
}
```

The modified Account class:

```
package sis.studentinfo;

import java.math.BigDecimal;

public class Account {
    private BigDecimal balance = new BigDecimal("0.00");
    private int transactionCount = 0;

    public void credit(BigDecimal amount) {
        balance = balance.add(amount);
        transactionCount++;
    }

    public BigDecimal getBalance() {
        return balance;
    }

    public BigDecimal transactionAverage() {
        return balance.divide(
            new BigDecimal(transactionCount), BigDecimal.ROUND_HALF_UP);
    }
}
```

More on
Primitive
Numerics

Review the Java API documentation for details on the other seven rounding modes.

More on Primitive Numerics

To this point, you have learned about the numeric primitive types `int`, `double`, and `float`. You have learned about the basic arithmetic operators, the increment/decrement operators for `int`, and compound assignment.

Other Integer Types

You have been using the `int` type for integer numerics. There are other integer types available, each representing different sizes available in which to store numbers of that type. Table 10.1 lists all integral types, their size, and the range of values that they support.

The `char` type is also a numeric type. Refer to Lesson 3 for more information on `char`.

You usually express numeric literals as decimal, or base 10, numbers. Java also allows you to represent them in hexadecimal (base 16) or octal (base 8).

You prefix hexadecimal literals with `0x`.

Table 10.1 Integer Types

Type	Size	Range	Example Literals
byte	8 bits	-128 to 127	0 24
char	16 bits	0 to 65535	17 'A' '%' \u0042 \062
short	16 bits	-32768 to 32767	1440 0x1AFF
int	32 bits	-2,147,483,648 to 2,147,483,647	-1440525 0x41F0 083
long	64 bits	-9223372036854775808 to 9223372036854775807	-90633969363693 77L 42

```
assertEquals(12, 0xC);
```

You prefix octal literals with `0`:

```
assertEquals(10, 012);
```

Integral literals are by default of `int` type. You can force an integral literal to be a `long` by suffixing it with the letter `L`. The lowercase letter `l` can also be used, but prefer use of an uppercase `L`, since the lowercase letter is difficult to discern from the number 1.

Integer Math

Let's revisit the code in the class `Performance` (from Lesson 7) that calculates the average for a series of tests.

```
public double average() {
    double total = 0.0;
    for (int score: tests)
        total += score;
    return total / tests.length;
}
```

Suppose the `total` local variable had been declared as an `int` instead of a `double`, which is reasonable since each test score is an `int`. Adding an `int` to an `int` returns another `int`.

```
public double average() {  
    int total = 0;  
    for (int score: tests)  
        total += score;  
    return total / tests.length;  
}
```

Numeric Casting

This seemingly innocuous change will break several tests in `PerformanceTest`. The problem is, dividing an `int` by an `int` also returns an `int`. Integer division always produces integer results; any remainders are discarded. The following two assertions demonstrate correct integer division:

```
assertEquals(2, 13 / 5);  
assertEquals(0, 2 / 4);
```

If you need the remainder while doing integer division, use the *modulus* operator (%).

```
assertEquals(0, 40 % 8);  
assertEquals(3, 13 % 5);
```

Numeric Casting

Java supports numeric types of different ranges in value. The `float` type uses 32 bits for its implementation and expresses a smaller range of numbers than does `double`, which uses 64 bits. The integral primitive types—`char`, `byte`, `short`, `int`, and `long`—each support a range that is different from the rest of the integral primitive types.

You can always assign a variable of a smaller primitive type to a larger primitive type. For example, you can always assign a `float` reference to a `double` reference:

```
float x = 3.1415f;  
double y = x;
```

Behind the scenes, this assignment implicitly causes a type conversion from `float` to `double`.

Going the other way—from a larger primitive type to a smaller—implies that there might be some loss of information. If a number stored as a `double` is larger than the largest possible `float`, assigning it to a `float` would result in the truncation of information. In this situation, Java recognizes the possible loss of information and forces you to cast so that you explicitly recognize that there might be a problem.

**Expression
Evaluation
Order**

If you attempt to compile this code:

```
double x = 3.1415d;
float y = x;
```

you will receive a compiler error:

```
possible loss of precision
found   : double
required: float
float y = x;
^
```

You must cast the `double` to a `float`:

```
double x = 3.1415d;
float y = (float)x;
```

You may want to go the other way and cast from integral primitives to floating-point numbers in order to force noninteger division. A solution to the above problem of calculating the average for a Performance is to cast the dividend to a `double` before applying the divisor:

```
public double average() {
    int total = 0;
    for (int score: tests)
        total += score;
    return (double)total / tests.length;
}
```

As an added note: When it builds an expression that uses a mix of integral and float values, Java converts the result of the expression to an appropriate float value:

```
assertEquals(600.0f, 20.0f * 30, 0.05);
assertEquals(0.5, 15.0 / 30, 0.05);
assertEquals(0.5, 15 / 30.0, 0.05);
```

Expression Evaluation Order

The order of evaluation in a complex expression is important. The basic rules:

- Java evaluates parenthesized expressions first, from innermost parentheses to outermost parentheses.
- Certain operators have higher precedence than others; for example, multiplication has a higher precedence than addition.
- Otherwise, expressions are evaluated from left to right.

You should use parentheses in particularly complex expressions, since the precedence rules can be difficult to remember. In fact, some of them are counterintuitive. Parenthesizing is worth the effort to ensure that there is no misunderstanding about how an expression will be evaluated. Just don't go overboard with the parentheses—most developers should be familiar with the basic precedence rules.



Use parentheses to simplify understanding of complex expressions.

NaN

Here are a few examples of how precedence affects the result of an expression:

```
assertEquals(7, 3 * 4 - 5);      // left to right
assertEquals(-11, 4 - 5 * 3);   // multiplication before subtraction
assertEquals(-3, 3 * (4 - 5));  // parentheses evaluate first
```

When in doubt, write a test!

NaN



If no test scores exist, the average for a Performance should be zero. You haven't yet written a test for this scenario. Add the following brief test to `PerformanceTest`.

```
public void testAverageForNoScores() {
    Performance performance = new Performance();
    assertEquals(0.0, performance.average());
}
```

If you run this, oops! You get a `NullPointerException`. That can be solved by initializing the integer array of tests defined in `Performance` to an empty array:

```
private int[] tests = {};
```

When rerunning the tests, you get a different exception:

```
junit.framework.AssertionFailedError: expected:<0.0> but was:<NaN>
```

`NaN` is a constant defined on both the `java.lang.Float` and `java.lang.Double` classes that means “Not a Number.”

If there are no scores, `tests.length()` returns 0, meaning you are dividing by zero in this line of code:

```
return total / tests.length;
```

Infinity

When it works with integers, Java throws an `ArithmaticException` to indicate that you are dividing by zero. With floating-point numbers, you get `NaN`, which *is* a legitimate floating-point number.

Fix the problem by checking the number of test scores as the very first thing in average.

```
public double average() {
    if (tests.length == 0)
        return 0.0;
    int total = 0;
    for (int score: tests)
        total += score;
    return (double)total / tests.length;
}
```

`NaN` has some interesting characteristics. Any boolean comparisons against `NaN` always result in `false`, as these language tests demonstrate:

```
assertFalse(Double.NaN > 0.0);
assertFalse(Double.NaN < 1.0);
assertFalse(Double.NaN == 1.0);
```

You might *want* the average method to return `NaN`. But how would you write the test, since you cannot compare `NaN` to any other floating-point number? Java supplies the `isNaN` static method, defined on both `Float` and `Double`, for this purpose:

```
public void testAverageForNoScores() {
    Performance performance = new Performance();
    assertTrue(Double.NaN(performance.average()));
}
```

Infinity

The `java.lang.Float` class provides two constants for infinity: `Float.NEGATIVE_INFINITY` and `Float.POSITIVE_INFINITY`. The class `java.lang.Double` provides corresponding constants.

While integral division by zero results in an error condition, `double` and `float` operations involving zero division result in the mathematically correct infinity value. The following assertions demonstrate use of the infinity constants.

```
final float tolerance = 0.5f;
final float x = 1f;

assertEquals(
    Float.POSITIVE_INFINITY, Float.POSITIVE_INFINITY * 100, tolerance);
```

```

assertEquals(Float.NEGATIVE_INFINITY,
    Float.POSITIVE_INFINITY * -1, tolerance);

assertEquals(Float.POSITIVE_INFINITY, x / 0f, tolerance);
assertEquals(Float.NEGATIVE_INFINITY, x / -0f, tolerance);
assertTrue(Float.isNaN(x % 0f));

assertEquals(0f, x / Float.POSITIVE_INFINITY, tolerance);
assertEquals(-0f, x / Float.NEGATIVE_INFINITY, tolerance);
assertEquals(x, x % Float.POSITIVE_INFINITY, tolerance);

assertTrue(Float.isNaN(0f / 0f));
assertTrue(Float.isNaN(0f % 0f));

assertEquals(
    Float.POSITIVE_INFINITY, Float.POSITIVE_INFINITY / x, tolerance);
assertEquals(
    Float.NEGATIVE_INFINITY, Float.NEGATIVE_INFINITY / x, tolerance);
assertTrue(Float.isNaN(Float.POSITIVE_INFINITY % x));

assertTrue(
    Float.isNaN(Float.POSITIVE_INFINITY / Float.POSITIVE_INFINITY));
assertTrue(
    Float.isNaN(Float.POSITIVE_INFINITY % Float.POSITIVE_INFINITY));
assertTrue(
    Float.isNaN(Float.POSITIVE_INFINITY / Float.NEGATIVE_INFINITY));
assertTrue(
    Float.isNaN(Float.POSITIVE_INFINITY % Float.NEGATIVE_INFINITY));
assertTrue(
    Float.isNaN(Float.NEGATIVE_INFINITY / Float.POSITIVE_INFINITY));
assertTrue(
    Float.isNaN(Float.NEGATIVE_INFINITY % Float.POSITIVE_INFINITY));
assertTrue(
    Float.isNaN(Float.NEGATIVE_INFINITY / Float.NEGATIVE_INFINITY));
assertTrue(
    Float.isNaN(Float.NEGATIVE_INFINITY % Float.NEGATIVE_INFINITY));

```

Numeric
Overflow

Numeric Overflow

When working with integral variables, you must be careful to avoid overflow, which can create incorrect results.

For each numeric type, Java provides constants representing its range. For example, the constants `Integer.MAX_VALUE` and `Integer.MIN_VALUE` represent the largest ($2^{31} - 1$) and smallest (-2^{31}) values an `int` can have, respectively.

Java will internally assign the results of an expression to an appropriately large primitive type. The example test shows how you can add one to a byte at its maximum value (127). Java stores the result of the expression as a larger primitive value.

```
byte b = Byte.MAX_VALUE;  
assertEquals(Byte.MAX_VALUE + 1, b + 1);
```

But if you attempt to assign the result of an expression that is too large back to the byte, the results are probably not what you expected. The following test passes, showing that adding 1 to a byte at maximum value results in the smallest possible byte:

```
byte b = Byte.MAX_VALUE;  
assertEquals(Byte.MAX_VALUE + 1, b + 1);  
b += 1;  
assertEquals(Byte.MIN_VALUE, b);
```

The result is due to the way Java stores the numbers in binary. Any additional significant bits are lost in an overflow condition.

Floating-point numbers overflow to infinity.

```
assertTrue(Double.isInfinite(Double.MAX_VALUE * Double.MAX_VALUE));
```

Floating-point numbers can also underflow—that is, have a value too close to zero to represent. Java makes these numbers zero.

Bit Manipulation

Java supports bit-level operations on integers. Among myriad other uses, you might use bit manipulation for mathematical calculations, for performance (some mathematical operations may be faster using bit shifting), in encryption, or for working with a compact collection of flags.

Binary Numbers

Your computer ultimately stores all numbers internally as a series of bits. A bit is represented by a binary, or base 2, number. A binary number is either 0 or 1. The following table on top of page 381 counts in binary from 0 through 15.

The table also shows the hexadecimal (“hex”) representation of each number. Java does not allow you to use binary literals, so the easiest way to understand bit operations is to represent numbers with hexadecimal (base 16) literals. Each hex digit in a hex literal represents four binary digits.

Java Binary Representation

Java represents an `int` as 32 binary digits. The leading bit indicates whether the `int` is positive (0) or negative (1). Java stores positive integers as pure bi-

Decimal	Binary	Hex	Decimal	Binary	Hex
0	0	0x0	8	1000	0x8
1	1	0x1	9	1001	0x9
2	10	0x2	10	1010	0xA
3	11	0x3	11	1011	0xB
4	100	0x4	12	1100	0xC
5	101	0x5	13	1101	0xD
6	110	0x6	14	1110	0xE
7	111	0x7	15	1111	0xF

Bit Manipulation

nary—each binary digit adds into a positive sum. Negative integers are stored in *two's complement* form. To obtain the two's complement representation of a number, take its positive binary representation, invert all binary digits, and add 1.

For example, the number 17 is represented in two's complement as:

0000_0000_0000_0000_0000_0001_0001³

The number -17 is represented in two's complement as:

1111_1111_1111_1111_1111_1111_1110_1111

Logical Bit Operations

Java contains four logical operators that allow you to manipulate bits: bit-and, bit-or, negation, and bit-xor (exclusive-or). A truth table defines how each logical bit operator works, by specifying the results for all possible bit pairings.

The bit-and, bit-or, and bit-xor logical operators operate on two integers; they are therefore referred to as binary operators. The negation operator operates on a single integer and is thus a unary operator.

To execute a binary logical operation, Java compares all corresponding bits from each of the two numbers involved. When doing a bitwise-and operation on two 32-bit integers, then, this means that 32 individual bit-ands

³I have added underscores for readability.

take place. To execute a unary logical operation against an integer, Java negates each bit in the integer individually.

You can perform logical bit operations against two integers of type `int` or smaller. However, Java internally translates integers declared as smaller than `int`—`byte`, `short`, and `char`—to type `int` before the bit operation takes place.

The bit-and operator (`&`) is also known as bitwise multiplication. If both bits are 1, bitwise multiplication returns a 1; otherwise it returns a 0. The following TDTT demonstrates bit-and.

```
assertEquals(0, 0 & 0);
assertEquals(0, 0 & 1);
assertEquals(0, 1 & 0);
assertEquals(1, 1 & 1);
```

The bit-or operator (`|`) is also known as bitwise addition. If both bits are 0, bitwise addition returns a 0; otherwise it returns a 1. The following TDTT demonstrates bit-or.

```
assertEquals(0, 0 | 0);
assertEquals(1, 0 | 1);
assertEquals(1, 1 | 0);
assertEquals(1, 1 | 1);
```

The xor (exclusive-or) operator (`^`) is also known as bitwise difference. If both bits are the same, bitwise difference returns 0; otherwise it returns a 1. The following TDTT demonstrates xor.

```
assertEquals(0, 0 ^ 0);
assertEquals(1, 0 ^ 1);
assertEquals(1, 1 ^ 0);
assertEquals(0, 1 ^ 1);
```

The logical negation operator (`~`) flips all bits in the integer so that 0s become 1s and 1s become 0s.

```
int x = 0xFFFFFFFF1;           //0111_1111_1111_1111_1111_1111_0001
assertEquals(0x8000000E, ~x); //1000_0000_0000_0000_0000_0000_1110
```

Java supports the use of compound assignment with logical bit operators. Thus, `x &= 1` is equivalent to `x = x & 1`.

Using Bit-And, Bit-Or and Negation

If you have a series of related flags (boolean values) to represent, you can use an integer variable to compactly represent them. A less-efficient alternative would be to define each as a separate boolean variable or as an array of booleans. Each bit in the integer represents a different flag. Thus, you could represent eight flags in a single byte. For example, the binary value 00000001 would mean

that the first flag is on and all other flags are off. The binary value `00000101` would mean that the first and third flags are on and all other flags are off.

To set the values of each flag, you first define a *mask* for each binary position. The mask for the first position is `00000001` (decimal 1), the mask for the second position is `00000010` (decimal 2), and so on. Setting a value is done using bit-or, and extracting a value is done using bit-and.

 You need to be able to set four yes-no flags on each student: Does the student reside on campus, is the student tax-exempt, is the student a minor, and is the student a troublemaker? You have 200,000 students and memory is at a premium.

```
public void testFlags() {
    Student student = new Student("a");
    student.set(
        Student.Flag.ON_CAMPUS,
        Student.Flag.TAX_EXEMPT,
        Student.Flag.MINOR);
    assertTrue(student.isOn(Student.Flag.ON_CAMPUS));
    assertTrue(student.isOn(Student.Flag.TAX_EXEMPT));
    assertTrue(student.isOn(Student.Flag.MINOR));

    assertFalse(student.isOff(Student.Flag.ON_CAMPUS));
    assertTrue(student.isOff(Student.Flag.TROUBLEMAKER));

    student.unset(Student.Flag.ON_CAMPUS);
    assertTrue(student.isOff(Student.Flag.ON_CAMPUS));
    assertTrue(student.isOn(Student.Flag.TAX_EXEMPT));
    assertTrue(student.isOn(Student.Flag.MINOR));
}
```

You should normally prefer explicit query methods for each flag (`isOnCampus`, `isTroublemaker`, etc.). The above approach may be preferable if you have a large number of flags and more dynamic needs.

The relevant code in `Student` follows.

```
public enum Flag {
    ON_CAMPUS(1),
    TAX_EXEMPT(2),
    MINOR(4),
    TROUBLEMAKER(8);

    private int mask;

    Flag(int mask) {
        this.mask = mask;
    }
}

private int settings = 0x0;
...
```

```

public void set(Flag... flags) {
    for (Flag flag: flags)
        settings |= flag.mask;
}

public void unset(Flag... flags) {
    for (Flag flag: flags)
        settings &= ~flag.mask;
}

public boolean isOn(Flag flag) {
    return (settings & flag.mask) == flag.mask;
}

public boolean isOff(Flag flag) {
    return !isOn(flag);
}

```

Bit Manipulation

Each flag is an `enum` constant, defined by the `Flag` `enum` located in `Student`. Each `enum` instantiation passes an integer representing a mask to the constructor of `Flag`. You store the flags in the `int` instance variable `settings`. You explicitly initialize this variable to all `0s` to demonstrate intent.

The `set` method takes a variable number of `Flag` `enum` objects. It loops through the array and applies the `Flag`'s mask to the `settings` variable by using a bit-or operator.

The `unset` method loops through `Flag` `enum` objects and bit-ands the negation of the `Flag`'s mask to the `settings` variable.

A demonstration of how the bit-or operation sets the correct bit:

<code>Flag.MINOR</code>	<code>0100</code>
<code>settings</code>	<code>0000</code>
<code>settings = settings Flag.MINOR</code>	<code>0100</code>
<code>Flag.ON_CAMPUS</code>	<code>0001</code>
<code>settings Flag.ON_CAMPUS</code>	<code>0101</code>

The `isOn` method bit-ands a `Flag`'s mask with the `settings` variable.

<code>settings</code>	<code>0000</code>
<code>Flag.MINOR</code>	<code>0100</code>
<code>settings & Flag.MINOR</code>	<code>0000</code>
<code>settings & Flag.MINOR == Flag.MINOR</code>	<code>false</code>
<code>settings = settings Flag.MINOR</code>	<code>0100</code>
<code>settings & Flag.MINOR</code>	<code>0100</code>
<code>settings & Flag.MINOR == Flag.MINOR</code>	<code>true</code>

Finally, a demonstration of the use of negation in the `unset` method:

settings	0101
<code>~Flag.MINOR</code>	1011
settings & <code>~Flag.MINOR</code>	0001

Bit Manipulation

Using bit operations to store multiple flags is a classic technique that comes from a need to squeeze the most possible information into memory. It is not normally recommended or needed for most Java development, since an array or other collection of `boolean` values provides a clearer, simpler representation.

Using Xor

The `xor` operator has the unique capability of being reversible.

```
int x = 5;          // 101
int y = 7;          // 111
int xPrime = x ^ y; // 010
assertEquals(2, xPrime);
assertEquals(x, xPrime ^ y);
```

You can also use `xor` as the basis for *parity checking*. Transmitting data introduces the possibility of individual bits getting corrupted. A parity check involves transmitting additional information that acts like a checksum. You calculate this checksum by applying an `xor` against all data sent. The receiver executes the same algorithm against the data and checksum. If checksums do not match, the sender knows to retransmit the data.

The parity check is binary: A stream of data has either even parity or odd parity. If the number of 1s in the data is even, parity is even. If the number of 1s in the data is odd, the parity is odd. You can use `xor` to calculate this parity. Here is a simple stand-alone test:

```
public void testParity() {
    assertEquals(0, xorAll(0, 1, 0, 1));
    assertEquals(1, xorAll(0, 1, 1, 1));
}

private int xorAll(int first, int... rest) {
    int parity = first;
    for (int num: rest)
        parity ^= num;
    return parity;
}
```

Xoring an even number of 1s will always result in an even parity (0). Xoring an odd number of 1s will always result in an odd parity (1).

The reason this works: Xoring is the same as adding two numbers, then taking modulus of dividing by 2. Here is an extension of the truth table that demonstrates this:

Bit Manipulation

$0 \wedge 0 = 0$	$(0 + 0) \% 2 = 0$
$0 \wedge 1 = 1$	$(0 + 1) \% 2 = 1$
$1 \wedge 0 = 1$	$(1 + 0) \% 2 = 1$
$1 \wedge 1 = 1$	$(1 + 1) \% 2 = 0$

Dividing mod 2 tells you whether a number is odd (1) or even (0). When you add a string of binary digits together, only the 1 digits will contribute to a sum. Thus, taking this sum modulus 2 will tell you whether the number of 1 digits is odd or even.

Take this one level further to calculate a checksum for any integer. The job of the ParityChecker class is to calculate a checksum for a byte array of data. The test shows how corrupting a single byte within a datum results in a different checksum.

```
package sis.util;

import junit.framework.*;

public class ParityCheckerTest extends TestCase {
    public void testSingleByte() {
        ParityChecker checker = new ParityChecker();
        byte source1 = 10; // 1010
        byte source2 = 13; // 1101
        byte source3 = 2; // 0010

        byte[] data = new byte[] { source1, source2, source3 };

        byte checksum = 5; // 0101

        assertEquals(checksum, checker.checksum(data));

        // corrupt the source2 element
        data[1] = 14; // 1110

        assertFalse(checksum == checker.checksum(data));
    }
}
```

ParityChecker loops through all bytes of data, xoring each byte with a cumulative checksum. In the test, I lined up the binary translation of each decimal number (source1, source2, and source3). This allows you to see how the checksum of 5 is a result of xoring the bits in each column.

```
package sis.util;

public class ParityChecker {
    public byte checksum(byte[] bytes) {
        byte checksum = bytes[0];
        for (int i = 1; i < bytes.length; i++)
            checksum ^= bytes[i];
        return checksum;
    }
}
```

Bit Manipulation

A simple xor parity check will not catch all possible errors. More-complex schemes involve adding a parity bit to each byte transmitted in addition to the a parity byte at the end of all bytes transmitted. This provides a matrix scheme that can also pinpoint the bytes in error.

Bit Shifting

Java provides three operators for shifting bits either left or right.

To shift bits left or right, and maintain the sign bit, use the bit shift left (`<<`) or bit shift right (`>>`) operators.

A bit shift left moves each bit one position to the left. Leftmost bits are lost. Rightmost bits are filled with zeroes.

For example, shifting the bit pattern `1011` one position to the left in a 4-bit unsigned number would result in `0110`. Some annotated Java examples:

```
//    101 = 5
assertEquals(10, 5 << 1); //    1010 = 10
assertEquals(20, 5 << 2); //    10100 = 20
assertEquals(40, 5 << 3); // 101000 = 40
assertEquals(20, 40 >> 1);
assertEquals(10, 40 >> 2);
// ... 1111 0110 = -10
assertEquals(-20, -10 << 1); // ... 1110 1100 = -20
assertEquals(-5, -10 >> 1); // ... 1111 1011 = -5
```

You'll note that bit shifting left is the equivalent of multiplying by powers of 2. Bit shifting right is the equivalent of dividing by powers of 2.

The unsigned bit shift right shifts all bits irrespective of the sign bit. Rightmost bits are lost, and the leftmost bits (including the sign bit) are filled with 0.

```
assertEquals(5, 10 >>> 1);
assertEquals(2147483643, -10 >>> 1);
// 1111_1111_1111_1111_1111_1111_0110 = -10
// 0111_1111_1111_1111_1111_1111_1011 = 2147483643
```

Note that an unsigned bit shift right always results in a positive number.


java.lang.Math

Bit shifting has uses in cryptography and in graphical image manipulation. You can also use bit shifting for some mathematical operations, such as dividing or multiplying by powers of 2. Only do so if you need extremely fast performance, and then only if your performance tests show you that the math is the performance problem.

BitSet

The Java API library includes the class `java.util.BitSet`. It encapsulates a vector of bits and grows as needed. `BitSet` objects are mutable—its individual bits may be set or unset. You can bit-and, bit-or, or bit-xor one `BitSet` object to another. You can also negate a `BitSet`. One of its few benefits is that it supports bit operations for numbers larger than the capacity of an `int`.

java.lang.Math

The class `java.lang.Math` provides a utility library of mathematical functions. It also provides constant definitions `Math.E`, for the base of the natural log, and `Math.PI`, to represent pi. These constants are double quantities that provide fifteen decimal places.

The Java API library also contains a class named `java.lang.StrictMath`, which provides bit-for-bit adherence to accepted standard results for the functions. For most purposes, `Math` is acceptable and also results in better performance.

I summarize the functions provided by the `Math` class in the following table. Refer to the Java API documentation for complete details on each function.

Method Name(s)	Functionality
<code>abs</code>	absolute value; overloaded for all primitive numeric types
<code>max, min</code>	maximum and minimum value
<code>acos, asin, atan</code>	arc cosine, arc sine, arc tangent
<code>cos, sin, tan</code>	cosine, sine, tangent
<code>atan2</code>	convert (x, y) coordinates to polar
<code>ceil, floor, round</code>	ceiling, floor, and rounding capabilities

Method Name(s)	Functionality
exp	e raised to a power
log	natural log
pow	x raised to y
sqrt	square root
random	random from 0.0 to 1.0
rint	closest int value for a double
toDegrees, toRadians	convert between degrees and radians
IEEEremainder	IEEE 754 remainder for double operation x / y

java.lang.Math

A method to calculate the hypotenuse of a right triangle provides a simple example. (The hypotenuse is the side opposite the right angle.)

```
// util.MathTest.java:
package util;

import junit.framework.*;

public class MathTest extends TestCase {
    static final double TOLERANCE = 0.05;

    public void testHypotenuse() {
        assertEquals(5.0, Math.hypotenuse(3.0, 4.0), TOLERANCE);
    }
}

// in util.Math:
package util;

import static java.lang.Math.*;

public class Math {
    public static double hypotenuse(double a, double b) {
        return sqrt(pow(a, 2.0) + pow(b, 2.0));
    }
}
```

The `pow` function is used to square the sides `a` and `b`. The `hypotenuse` method then sums these squares and returns the square root of the sum using `sqrt`.

Pervasive use of the `Math` class gives you a justifiable reason for using the static import facility introduced in Lesson 4. Code with a lot of `Math` method calls otherwise becomes a nuisance to code and read.

I've named the utility class `Math`, like the `java.lang.Math` class. While this is a questionable practice (it may introduce unnecessary confusion), it demonstrates that Java can disambiguate the two.

Numeric Wrapper Classes

As you saw in Lesson 7, Java supplies a corresponding wrapper class for each primitive type: `Integer`, `Double`, `Float`, `Byte`, `Boolean`, and so on. The primary use of the wrapper class is to convert primitives to object form so that they can be stored in collections.

The wrapper classes for the numeric types have additional uses. You learned earlier in this lesson that each numeric wrapper class provides a `MIN_VALUE` and `MAX_VALUE` constant. You also saw how the `Float` and `Double` classes provide constants for Not a Number (`NaN`) and infinity.

Printable Representations

The class `Integer` provides static utility methods to produce printable representations of an `int` in hex, octal, and binary form.

```
assertEquals("101", Integer.toBinaryString(5));
assertEquals("32", Integer.toHexString(50));
assertEquals("21", Integer.toOctalString(17));
```

The Java library includes general-purpose methods to produce a proper `String` for any radix (base). The following assertion shows how to get the trinary (base 3) `String` for an `int`.

```
assertEquals("1022", Integer.toString(35, 3));
```

Converting Strings to Numbers

In Lesson 8, you learned that the method `Integer.parseInt` takes a `String` and parses it to produce the corresponding `int` value. You typically use the `parseInt` method to convert user interface input into numeric form. All characters in the input `String` must be decimal digits, except for the first character, which may optionally be a negative sign. If the `String` does not contain a parseable integer, `parseInt` throws a `NumberFormatException`.

An additional form of `parseInt` takes a radix, representing the base of the input string, as the second parameter. The method `valueOf` operates the same as `parseInt`, except that it returns a new `Integer` object instead of an `int`.

Each of the corresponding numeric wrapper classes has a parsing method. For example, you may convert a String to a double using Double.parseDouble, or to a float using Float.parseFloat.

The parseInt method can accept only decimal input. The decode method can parse an input string that might be in hex or octal, as these assertions show:

```
assertEquals(253, Integer.decode("0xFD"));
assertEquals(253, Integer.decode("0XFd"));
assertEquals(253, Integer.decode("#FD"));
assertEquals(15, Integer.decode("017"));
assertEquals(10, Integer.decode("10"));
assertEquals(-253, Integer.decode("-0xFD"));
assertEquals(-253, Integer.decode("-0XFd"));
assertEquals(-253, Integer.decode("-#FD"));
assertEquals(-15, Integer.decode("-017"));
assertEquals(-10, Integer.decode("-10"));
```

**Random
Numbers**

Random Numbers

The class Math provides the method random to return a pseudorandom double between 0.0 and 1.0. A number in this range may be all you need. The generated number is not truly random—the elements of a sequence of pseudorandom numbers are only approximately independent from each other.⁴ For most applications, this is sufficient.

The class java.util.Random is a more comprehensive solution for generating pseudorandom numbers. It produces pseudorandom sequences of booleans, bytes, ints, longs, floats, Gaussians, or doubles. A pseudorandom sequence generator is not purely random; it is instead based upon an arithmetical algorithm.

You can create a Random instance with or without a *seed*. A seed is essentially a unique identifier for a random sequence. Two Random objects created with the same seed will produce the same sequence of values. If you create a Random object without explicitly specifying a seed, the class uses the system clock as the basis for the seed.⁵

You could create a simulation of coin flips through repeated use of the nextBoolean method on Random. A true value would indicate heads and a false

⁴http://en.wikipedia.org/wiki/Pseudo-random_number_generator.

⁵Which means that if you construct two Random objects and the construction happens to execute in the same nanosecond, you get the same sequence in both. In earlier versions of Java, the Random constructor used the current system time in milliseconds. This increased granularity significantly increased the likelihood of two Random objects having the same sequence.

value would indicate tails. The below test shows use of the seed value to produce two identical pseudorandom coin-flip sequences.

```
public void testCoinFlips() {
    final long seed = 100L;
    final int total = 10;
    Random random1 = new Random(seed);
    List<Boolean> flips1 = new ArrayList<Boolean>();
    for (int i = 0; i < total; i++) {
        flips1.add(random1.nextBoolean());

    Random random2 = new Random(seed);
    List<Boolean> flips2 = new ArrayList<Boolean>();
    for (int i = 0; i < total; i++)
        flips2.add(random2.nextBoolean());

    assertEquals(flips1, flips2);
}
```

The methods `nextInt`, `nextDouble`, `nextLong`, and `nextFloat` work similarly. An additional version of `nextInt` takes as a parameter a maximum value, such that the method always returns a number from 0 through the maximum value.

Testing Random Code

There are a few ways to write unit tests against code that must use random sequences. One way is to provide a subclass implementation of the `Random` class and substitute that for the `Random` class used in the code. The substitute `Random` class will provide a known sequence of values. This technique is known as *mocking* the `Random` class. I will cover mocking in considerable depth later.



In Lesson 8, you provided your own logging Handler class solely for purposes of testing. This was also a form of mocking.

You must assign passwords to students so they can access their accounts online. Passwords are eight characters long. Each character of the password must fall in a certain character range.

```
package sis.util;

import junit.framework.*;

public class PasswordGeneratorTest extends TestCase {
    public void testGeneratePassword() {
        PasswordGenerator generator = new PasswordGenerator();
        generator.setRandom(new MockRandom('A'));
        assertEquals("ABCDEFGH", generator.generatePassword());
```

```

        generator.setRandom(new MockRandom('C'));
        assertEquals("CDEFGHIJ", generator.generatePassword());
    }
}

```

The test method `testGeneratePassword` sets the `random` variable into the `PasswordGenerator` class to point to an instance of `MockRandom`. The `MockRandom` class extends from the `Random` class. The Java API documentation explains how to properly extend the `Random` class by overriding the method `next(int bits)` to return a random number based on a bit sequence. All other methods in the `Random` class are based on this method.

**Random
Numbers**

The *mock* implementation takes an `int` representing a starting character value as a parameter to its constructor. It uses this to calculate an initial value for the random sequence and stores it as `i`. The initial value is a number relative to the lowest valid character in the password. Its `next(int bits)` method simply returns and then increments the value of `i`.

```

package sis.util;

import junit.framework.*;

public class PasswordGeneratorTest extends TestCase {
    ...
    class MockRandom extends java.util.Random {
        private int i;
        MockRandom(char startCharValue) {
            i = startCharValue - PasswordGenerator.LOW_END_PASSWORD_CHAR;
        }
        protected int next(int bits) {
            return i++;
        }
    }
}

```

`MockRandom` is defined completely within `PasswordGeneratorTest`. It is a *nested class* of `PasswordGenerator`. You can directly instantiate `MockRandom` from within `PasswordGeneratorTest`, but not from any other class. There are additional kinds of nested classes, each kind with its own nuances. In Lesson 11, you will learn about nested classes in depth. Until then, use them with caution.

The implementation of `PasswordGenerator` follows.

```

package sis.util;

import java.util.*;

```

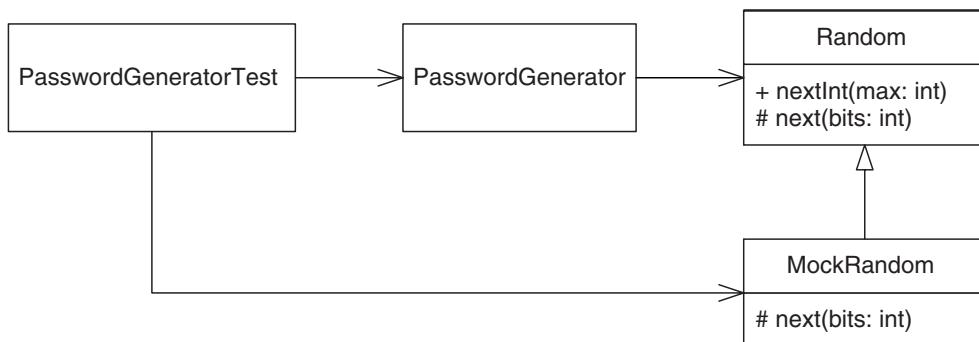
**Random
Numbers**

```
public class PasswordGenerator {  
    private String password;  
    private static final int PASSWORD_LENGTH = 8;  
    private Random random = new Random();  
  
    static final char LOW_END_PASSWORD_CHAR = 48;  
    static final char HIGH_END_PASSWORD_CHAR = 122;  
  
    void setRandom(Random random) {  
        this.random = random;  
    }  
  
    public String generatePassword() {  
        StringBuffer buffer = new StringBuffer(PASSWORD_LENGTH);  
        for (int i = 0; i < PASSWORD_LENGTH; i++)  
            buffer.append(getRandomChar());  
        return buffer.toString();  
    }  
  
    private char getRandomChar() {  
        final char max = HIGH_END_PASSWORD_CHAR - LOW_END_PASSWORD_CHAR;  
        return (char)(random.nextInt(max) + LOW_END_PASSWORD_CHAR);  
    }  
  
    public String getPassword() {  
        return password;  
    }  
}
```

Note that if the package-level `setRandom` method is not called, the `random` instance variable keeps its initialized value of a legitimate `java.util.Random` instance. Since `MockRandom` meets the contract of `java.util.Random`, you can substitute an instance of `MockRandom` for `Random` for purposes of testing.

The goal of `testGeneratePassword` is to prove that the `PasswordGenerator` class can generate and return a random password. The test does not need to reprove the functionality inherent in the `Random` class, but the test does need to prove that a `PasswordGenerator` object interacts with a `Random` instance (or subclass instance) through its published interface, `nextInt(int max)`. The test must also prove that `PasswordGenerator` uses the `nextInt` return value properly. In this example, `PasswordGenerator` uses the `nextInt` return value to help construct an 8-character String of random values.

Java supplies an additional random class, `java.util.SecureRandom`, as a standards-based, cryptographically strong pseudorandom number generator.



Exercises

Figure 10.1 Mocking the Random Class

Exercises

1. Create a test demonstrating the immutability of a BigDecimal. Add a second BigDecimal to the first and show that the first still has the original value.
2. Create one BigDecimal using the value "10.00" and another using the value "1". Show they are not equal. Use multiplication and scale to create a new BigDecimal from the second so that it equals the first. Now reverse the transformation, starting from 10.00, and get back to 1.
3. Show that 0.9 and 0.005 * 2.0 are not equal with floats. To what precision are they equal? With doubles?
4. Why won't the following compile? What are two ways it can be fixed?


```
public class CompilerError {
    float x = 0.01;
}
```
5. Write a brief spike program to discover the decimal value of 0xDEAD. How could you recode the literal in octal?
6. Find as many interesting variations of NaN and Infinity calculations as you can.
7. Challenge: Can you find ways in which the wrappers such as Float act differently than the corresponding primitive?
8. Create a method (test first, of course) that takes a variable number of int values. Code it to return only numbers in the list that are divisible by 3. Code it twice: The first time, use only the modulus (%) operator.

Exercises

The second time, use the division (/) operator and the multiplication operator if necessary, but not the % operator. Test against the sequence of integers from 1 through 10.

9. Change the chess code to use casting where appropriate. Eliminate the need for the `toChar` method on the `Board` class.

10. Which of the following lines will compile correctly? Why?

```
float x = 1;  
float x = 1.0;  
float x = (int)1.0;
```

11. What is the value of `(int)1.9`?

12. What is the value of `Math.rint(1.9)`?

13. When using `Math.rint`, how is rounding performed? Is 1.5 equal to 1 or 0? How about 2.5?

14. What are the final values of the following expressions, assuming `x` is 5 and `y` is 10 and `x` and `y` are both `ints`? Flag the lines that wouldn't compile, as well. What are the values of `x` and `y` afterward? (Consider each expression to be discrete—i.e., each expression starts with the values of `x` and `y` at 5 and 10.)

```
x * 5 + y++ * 7 / 4  
++x * 5 * y++  
x++ * 5 * ++y  
++x + 5 * 7 + y++  
++y++ % ++x++  
x * 7 == 35 || y++ == 0    // super tricky  
++x * ++y  
x++ * y++  
true && x * 7  
x * 2 == y- || ++y == 10   // super tricky  
x * 2 == -y || ++y == 10  // super tricky
```

15. Using only the `<<` operator, convert 17 into 34.

16. What is the decimal value of `~1`?

17. Demonstrate the difference between `>>` and `>>>`. Is there a difference between the two for positive numbers? For negative numbers?

18. Create a function which uses `Math.random` to generate a random integer from 1 through 50. How will you go about testing your function? Can you test it perfectly? How confident can you get that your code works?

19. Create a list of numbers from 1 to 100. Use a Random generator to randomly swap elements of the list around 100 times. Write a test that

does a reasonable verification: Ensure that the list size remains the same and that two numbers were swapped with each call to the swapper.

20. Show that the next `double` is not the same between a `Random` seeded with 1 and a `Random` with no seed. Can you make a test that absolutely proves that this is true?
21. Challenge: Swap two numbers without a temporary variable by using the `xor` operator.
22. Challenge: Programmatically demonstrate the number of bits required to store each integral type (`char`, `byte`, `short`, `int`, `long`). Hint: Use shift operators. Remember that signed types require an extra bit to store the sign.

Exercises

This page intentionally left blank

Lesson 11

IO

Organization

In this lesson you will learn about the input-output (IO) facilities of Java. IO refers to anything that transfers data to or from your application. IO in Java is wonderfully complete, wonderfully complex, and reasonably consistent. You might use Java's IO capabilities to write reports to files or to read user input from the console.

In this lesson, you will learn about:

- organization of the stream classes
- character streams versus byte streams
- the File class
- data streams
- redirecting System.in and System.out
- object streams
- random access files
- nested classes

Organization

The java.io package contains dozens of classes to manage input and output using data streams, serialization, and the file system. Understanding the organization and naming strategy of the classes in the package will help you minimize the otherwise overwhelming nature of doing IO in Java.

Java IO is based upon the use of *streams*. A stream is a sequence of data that you can write to or read from. A stream may have a source, such as the

console, or a destination, such as the file system. A stream with a source is an input stream; a stream with a destination is an output stream.

Java further distinguishes its streams by the format of the data they carry. Java contains character streams and byte streams. You use character streams to work with 2-byte (16-bit) Unicode character data. Java classes with Reader (input) and Writer (output) in their names are character streams. You use character streams to work with human-readable text.

Byte streams work with 8-bit binary data. Byte stream classes have the word Input or Output in their name. You normally use byte streams for working with nontext data such as image files or compiled byte codes.

Low-level Java streams support basic concepts such as reading or writing single bytes at a time. Were you able to work only with low-level streams, you would be forced to write tedious, repetitive code. Java supplies a number of higher-level streams that simplify your work by providing aggregate and additional functionality. These higher level streams are known as *wrapper* streams.

A wrapper stream object contains a reference to, or *wraps*, a low-level stream object. You interact with the wrapper stream. It in turn interacts with the wrapped low-level stream that does the dirty work.

Take a look at the java.io package, which contains a set of interfaces and classes to define the basic building blocks for IO. You will note that it is crowded—Java's IO facilities are complete but complex. The package includes specific stream implementations, including filtered, buffered, piped, and object streams.

In addition to the stream classes, the java.io package provides a set of classes to manipulate the underlying file system.

Character Streams

In Lesson 3, you created the RosterReporter class to write a report of students enrolled in a course session. You wrote the report to a string, then printed the report using System.out. Here is the existing RosterReporter class:

```
package sis.report;

import java.util.*;
import sis.studentinfo.*;
import static sis.report.ReportConstant.NEWLINE;
```

```

class RosterReporter {
    static final String ROSTER_REPORT_HEADER =
        "Student" + NEWLINE +
        "----" + NEWLINE;
    static final String ROSTER_REPORT_FOOTER =
        NEWLINE + "# students = ";

    private Session session;

    RosterReporter(Session session) {
        this.session = session;
    }

    String getReport() {
        StringBuilder buffer = new StringBuilder();

        writeHeader(buffer);
        writeBody(buffer);
        writeFooter(buffer);

        return buffer.toString();
    }

    void writeHeader(StringBuilder buffer) {
        buffer.append(ROSTER_REPORT_HEADER);
    }

    void writeBody(StringBuilder buffer) {
        for (Student student: session.getAllStudents()) {
            buffer.append(student.getName());
            buffer.append(NEWLINE);
        }
    }

    void writeFooter(StringBuilder buffer) {
        buffer.append(
            ROSTER_REPORT_FOOTER + session.getAllStudents().size() +
            NEWLINE);
    }
}

```

Character Streams

Currently the `RosterReporter` class builds a string representing the entire report. Another class using `RosterReporter` would take this string and send it to a desired destination—the console, a file, or perhaps the Internet. Thus, you write every character in the report twice—first to the `String`, then to a final destination. With a larger report, the receiver of the report may experience an unacceptable delay while waiting for the entire report to be produced.

A preferable solution would involve writing each character directly to a stream that represents the ultimate destination. This also means that you

wouldn't need to store the entire report in a large string buffer, something that has the potential to cause memory problems.

 You have been told to make the student information system flexible. Initially the system must be able to write reports to either the console or to local files. To meet this requirement, you will first update the RosterReporter class to write directly to a character stream. First, you will need to update the test.

Character Streams

```
package sis.report;

import junit.framework.TestCase;
import sis.studentinfo.*;
import static sis.report.ReportConstant.NEWLINE;
import java.io.*;

public class RosterReporterTest extends TestCase {
    public void testRosterReport() throws IOException {
        Session session =
            CourseSession.create(
                new Course("ENGL", "101"),
                DateUtil.createDate(2003, 1, 6));

        session.enroll(new Student("A"));
        session.enroll(new Student("B"));

        Writer writer = new StringWriter();
        new RosterReporter(session).writeReport(writer);
        String rosterReport = writer.toString();

        assertEquals(
            RosterReporter.ROSTER_REPORT_HEADER +
            "A" + NEWLINE +
            "B" + NEWLINE +
            RosterReporter.ROSTER_REPORT_FOOTER + "2" +
            NEWLINE, rosterReport);
    }
}
```

To use Java's IO classes, you must import the package `java.io`. Many IO operations can generate an exception. You'll need to declare that the test method throws an `IOException`.

You want the `RosterReporter` to write its report to a `Writer` object provided by the client. The class `java.io.Writer` is the base abstraction for character streams. For purposes of testing, you can create an object of the `Writer` subclass `StringWriter`. When you send the message `toString` to a `StringWriter`, it returns a `String` of all characters written to it.

Note the improvement to the design. Instead of *asking* the `RosterReporter` object for a report (`getReport`), you are *telling* it to write a report (`writeReport`).

In RosterReporter, like RosterReporter test, you will need to add an `import` statement and `throws` clauses to methods that perform IO operations. (Instead of believing me, the better approach is to code this without the `throws` clauses and let the compiler tell you what to do.)

```
package sis.report;

import java.util.*;
import sis.studentinfo.*;
import static sis.report.ReportConstant.NEWLINE;
import java.io.*;

class RosterReporter {
    static final String ROSTER_REPORT_HEADER =
        "Student" + NEWLINE +
        "---" + NEWLINE;
    static final String ROSTER_REPORT_FOOTER =
        NEWLINE + "# students = ";

    private Session session;
    private Writer writer;

    RosterReporter(Session session) {
        this.session = session;
    }

    void writeReport(Writer writer) throws IOException {
        this.writer = writer;
        writeHeader();
        writeBody();
        writeFooter();
    }

    private void writeHeader() throws IOException {
        writer.write(ROSTER_REPORT_HEADER);
    }

    private void writeBody() throws IOException {
        for (Student student: session.getAllStudents())
            writer.write(student.getName() + NEWLINE);
    }

    private void writeFooter() throws IOException {
        writer.write(
            ROSTER_REPORT_FOOTER + session.getAllStudents().size() +
            NEWLINE);
    }
}
```

Character Streams

In `writeHeader`, `writeBody`, and `writeFooter`, you have replaced calls to the `String-Builder` method `append` with calls to the `Writer` method `write`. Also, previously

the code passed a `StringBuilder` from method to method. Now you have a `Writer` instance variable. By not passing the same pervasive variable to every method, you can eliminate some duplication.

Watch your tests pass, then make the following refactorings to take advantage of Java String formatting.

Character Streams

```
package sis.report;

import java.util.*;
import sis.studentinfo.*;
import java.io.*;

class RosterReporter {
    static final String ROSTER_REPORT_HEADER = "Student%n---%n";
    static final String ROSTER_REPORT_FOOTER = "%n# students = %d%n";

    private Session session;
    private Writer writer;

    RosterReporter(Session session) {
        this.session = session;
    }

    void writeReport(Writer writer) throws IOException {
        this.writer = writer;
        writeHeader();
        writeBody();
        writeFooter();
    }

    private void writeHeader() throws IOException {
        writer.write(String.format(ROSTER_REPORT_HEADER));
    }

    private void writeBody() throws IOException {
        for (Student student: session.getAllStudents())
            writer.write(String.format(student.getName() + "%n"));
    }

    private void writeFooter() throws IOException {
        writer.write(
            String.format(ROSTER_REPORT_FOOTER,
                session.getAllStudents().size()));
    }
}
```

The requisite changes to the test:

```

package sis.report;

import junit.framework.TestCase;
import sis.studentinfo.*;
import java.io.*;

public class RosterReporterTest extends TestCase {
    public void testRosterReport() throws IOException {
        Session session =
            CourseSession.create(
                new Course("ENGL", "101"),
                DateUtil.createDate(2003, 1, 6));

        session.enroll(new Student("A"));
        session.enroll(new Student("B"));

        Writer writer = new StringWriter();
        new RosterReporter(session).writeReport(writer);
        String rosterReport = writer.toString();

        assertEquals(
            String.format(RosterReporter.ROSTER_REPORT_HEADER +
                "A%n" +
                "B%n" +
                RosterReporter.ROSTER_REPORT_FOOTER, 2),
            rosterReport) ;
    }
}

```

Writing to a File

Writing to a File

You will now need to update `RosterReporter` to be able to take a filename as a parameter. The test will need to make sure that the report is properly written to an operating system file.

First, you will refactor `RosterReporterTest` to create a `setUp` method and an assertion method `assertReportContents`. By doing so, you will set the stage to quickly add a new test, `testFiledReport`, that uses these common methods and thus introduces no duplication.

Within `assertReportContents`, you can modify the assertion to obtain the number of expected students from the `session` object. You will need to change `getNumberOfStudents` in `Session` from package to public.

The refactored `RosterReporterTest`:

```

package sis.report;

import junit.framework.TestCase;
import sis.studentinfo.*;
import java.io.*;

```

```

public class RosterReporterTest extends TestCase {
    private Session session;

    protected void setUp() {
        session =
            CourseSession.create(
                new Course("ENGL", "101"),
                DateUtil.createDate(2003, 1, 6));

        session.enroll(new Student("A"));
        session.enroll(new Student("B"));
    }

    public void testRosterReport() throws IOException {
        Writer writer = new StringWriter();
        new RosterReporter(session).writeReport(writer);
        assertEquals(writer.toString());
    }

    private void assertReportContents(String rosterReport) {
        assertEquals(
            String.format(RosterReporter.ROSTER_REPORT_HEADER +
                "A%n" +
                "B%n" +
                RosterReporter.ROSTER_REPORT_FOOTER,
                session.getNumberOfStudents(),
                rosterReport));
    }
}

```

Writing to a File

The new test, `testFiledReport`, calls an overloaded version of `writeReport`. This second version of `writeReport` takes a filename instead of a `Writer` as a parameter.

```

public void testFiledReport() throws IOException {
    final String filename = "testFiledReport.txt";
    new RosterReporter(session).writeReport(filename);

    StringBuffer buffer = new StringBuffer();
    String line;

    BufferedReader reader =
        new BufferedReader(new FileReader(filename));
    while ((line = reader.readLine()) != null)
        buffer.append(String.format(line + "%n"));
    reader.close();

    assertEquals(buffer.toString());
}

```

After calling `writeReport`, the test uses a `BufferedReader` to read the contents of the report file into a buffer. The test then passes the buffer's contents to the `assertReportContents` method.

A `BufferedReader` is a Reader subclass that can wrap another reader. Remember that you use Reader objects to read from character streams. The test constructs a `BufferedReader` with a `FileReader` as a parameter. A `FileReader` is a character-based input stream that can read data from a file. You construct a `FileReader` using a filename.

You could read the file's contents by directly using a `FileReader` instead of a `BufferedReader`. However, a `BufferedReader` is more efficient since it buffers characters as it reads. In addition, `BufferedReader` supplies the method `readLine` to help you simplify your code. The `readLine` method returns a logical line of input from the wrapped stream, using the system property "line.separator" to delineate lines.

The new method in `RosterReporter`:

```
void writeReport(String filename) throws IOException {
    Writer bufferedWriter =
        new BufferedWriter(new FileWriter(filename));
    try {
        writeReport(bufferedWriter);
    }
    finally {
        bufferedWriter.close();
    }
}
```

The `writeReport` method creates a `FileWriter` using the filename passed to it. It wraps the `FileWriter` in a `BufferedWriter`. The method then passes the writer to the existing `writeReport` method (which takes a `PrintWriter` as a parameter).

A `finally` block ensures that the `PrintWriter` is always closed, regardless of whether `writeReport` threw an exception or not. You must remember to close file resources, otherwise you may encounter locked file problems. Also, buffered information will not appear in the file until you close the Writer. You can also use the `Writer` method `flush` to force the `Writer` to write its contents to the destination.

Notice that you were able to add functionality to the existing `RosterReporter` class without modifying a single line of its code. You simply added a new method. Achieving this ideal happens more often if you strive to ensure that your code maintains an optimal design at all times. Prefer the use of abstractions in your code.

Writing to a File

java.io.File

The File class is not a stream-based class. Instead of working with streams, the File class provides you with an interface into the file and directory structure of your underlying file system. It contains a number of file-based utilities, such as the ability to delete files and to create temporary files.

java.io.File

When writing tests that work with files, you want to ensure that you have a clean environment for testing. You also want to ensure that you leave things the way they were when the test completes. With respect to `testFiledReport`, this means that you need to delete any existing report file as the first step in the test and delete the created report file as the last step in the test.

```
public void testFiledReport() throws IOException {
    final String filename = "testFiledReport.txt";
    try {
        delete(filename);

        new RosterReporter(session).writeReport(filename);

        StringBuffer buffer = new StringBuffer();
        String line;

        BufferedReader reader =
            new BufferedReader(new FileReader(filename));
        while ((line = reader.readLine()) != null)
            buffer.append(String.format(line + "%n"));
        reader.close();

        assertReportContents(buffer.toString());
    }
    finally {
        delete(filename);
    }
}

private void delete(String filename) {
    File file = new File(filename);
    if (file.exists())
        assertTrue("unable to delete " + filename, file.delete());
}
```

The `delete` method uses the File class to accomplish its goals. It creates a File object by passing a filename to its constructor. It calls the `exists` method to determine whether the file can be found in the file system. Finally, it calls the `delete` method, which returns `true` if the file system successfully removed the file, `false` otherwise. A file system might not be able to delete a file if it is locked or has its read-only attribute set, among other reasons.

Table 11.1 *java.io.File methods by category*

Category	Methods
temporary file creation	<code>createTempFile</code>
empty file creation	<code>createNewFile</code>
file manipulation	<code>delete, deleteOnExit, renameTo</code>
file and path name queries	<code>getAbsoluteFile, getAbsolutePath, getCanonicalFile, getCanonicalPath, getName, getPath, toURI, toURL</code>
classification	<code>isFile, isDirectory</code>
attribute query/manipulation	<code>isHidden, lastModified, length, canRead, canWrite, setLastModified, setReadOnly</code>
directory query/manipulation	<code>exists, list, listFiles, listRoots, mkdir, mkdirs, getParent, getParentFile</code>

**Byte Streams
and Conversion**

I have categorized most of the functionality of the class `java.io.File` in Table 11.1.

Byte Streams and Conversion

Java represents standard input (`stdin`) and standard output (`stdout`) with stream objects stored in `System.in` and `System.out`, respectively. `System.in` refers to an `InputStream`, while `System.out` refers to a `PrintStream`. A `PrintStream` is a specialized `OutputStream` that simplifies writing objects of various types.

Both `System.in` and `System.out` are byte streams, not character streams. Remember that Java uses the multibyte character set Unicode, while most operating systems use a single-byte character set.

The `java.io` package provides a means of converting between byte streams and character streams. The class `InputStreamReader` wraps an `InputStream` and converts each byte read into an appropriate character. The conversion uses your platform system's default encoding scheme (as understood by Java) by default; you may also supply a decoding class that defines the mapping. In a similar fashion, the class `OutputStreamWriter` wraps an output stream, converting each character to a single byte.

The predominant use for InputStreamReader and OutputStreamWriter is to map between Java's character streams and stdin/stdout. Reader and Writer subclasses allow you to work with input and output on a line-by-line basis.



Do not send messages directly to System.in and System.out.

Referring to Reader and Writer abstract classes instead allows for quick redirecting to a different medium (e.g., files), and also makes for easier testing.

A Student User Interface

A Student User Interface

 In this section, you will write a very simple console-based user interface (UI). The UI will allow an end user of the student information system to create student objects. You will prompt the user with a simple menu that either drives the addition of a new student or allows the user to terminate (exit) the application.

There are many ways to approach development of a user interface through TDD. The following test, implemented in the class StudentUITest, presents one solution. I have broken the listing into three parts for clarity. The first part shows the core test method, testCreateStudent.

```
package sis.ui;

import junit.framework.*;
import java.io.*;
import java.util.*;
import sis.studentinfo.*;

public class StudentUITest extends TestCase {
    static private final String name = "Leo Xerxes Schmoo";

    public void testCreateStudent() throws IOException {
        StringBuffer expectedOutput = new StringBuffer();
        StringBuffer input = new StringBuffer();
        setup(expectedOutput, input);
        byte[] buffer = input.toString().getBytes();

        InputStream inputStream = new ByteArrayInputStream(buffer);
        BufferedReader reader =
            new BufferedReader(new InputStreamReader(inputStream));

        OutputStream outputStream = new ByteArrayOutputStream();
        BufferedWriter writer =
            new BufferedWriter(new OutputStreamWriter(outputStream));

```

```

StudentUI ui = new StudentUI(reader, writer);
ui.run();

assertEquals(
    expectedOutput.toString(),
    outputStream.toString());
assertStudents(ui.getAddedStudents());
}

private String line(String input) {
    return String.format("%s%n", input);
}
...
}

```

A Student User Interface

The test creates two `StringBuffers`. One will hold expected output—what the console application displays—and the other will hold the input that a user (the test, in this case) types in. The `setUp` method explicitly populates each of these:

```

private void setup(StringBuffer expectedOutput, StringBuffer input) {
    expectedOutput.append(StudentUI.MENU);
    input.append(line(StudentUI.ADD_OPTION));
    expectedOutput.append(StudentUI.NAME_PROMPT);
    input.append(line(name));
    expectedOutput.append(line(StudentUI.ADDED_MESSAGE));
    expectedOutput.append(StudentUI.MENU);
    input.append(line(StudentUI.QUIT_OPTION));
}

```

The `setUp` method can be viewed as presenting a script to which the system must adhere. The buffer appends are intertwined to emulate the interaction between system output and user input. This natural flow should aid you in understanding the script. First, the user interface presents the user with a menu:

```
expectedOutput.append(StudentUI.MENU);
```

The user responds to the menu by first typing the option to add a student, then pressing enter.

```
input.append(line(StudentUI.ADD_OPTION));
```

The `line` method is a utility method that appends an end-of-line string to an input string. The remainder of the `setUp` method represents the expected sequence of events between the system presenting some text (`expectedOutput.append`) and the user entering a response (`input.append`).

The code in `testCreateStudent` creates an `InputStream` by wrapping a `ByteArrayInputStream` around the bytes from the input `StringBuffer`. It then

creates an InputStreamReader, wrapped with a BufferedReader, to convert incoming bytes to characters.

Similarly, the test creates and wraps a ByteArrayOutputStream with an OutputStreamWriter that will convert characters to bytes. A BufferedWriter wraps the OutputStreamWriter.

You pass these two streams, reader and writer, to the constructor of StudentUI after setup executes. The UI is told to run and should continually present a menu of options until the user chooses to quit.

After the UI completes processing, an assertion verifies that the application produced correct output. The assertion compares the contents of the ByteArrayOutputStream to the contents of the expectedOutput StringBuffer.

Finally, the test calls assertStudents to ensure that the list of students created and stored in the UI instance is as expected.

```
private void assertStudents(List<Student> students) {
    assertEquals(1, students.size());
    Student student = students.get(0);
    assertEquals(name, student.getName());
}
```

To verify the example user “script” coded in the setup method, assertStudents must ensure that the StudentUI object added only one student and that this student’s data (its name) is as expected.

The entire StudentUI implementation appears below. To build this class test-first and very incrementally, you should start with an even simpler set of unit tests. First, ensure that the menu is presented and that the user can immediately quit the application. Then add functionality to support adding students. Start with assertions against the user interface flow, then add the assertion to ensure that a student object was created. A more complete test than this would ensure that multiple students could be created.

```
package sis.ui;

import java.io.*;
import java.util.*;
import sis.studentinfo.*;

public class StudentUI {
    static final String MENU = "(A)dd or (Q)uit?";
    static final String ADD_OPTION = "A";
    static final String QUIT_OPTION = "Q";
    static final String NAME_PROMPT = "Name: ";
    static final String ADDED_MESSAGE = "Added";

    private BufferedReader reader;
    private BufferedWriter writer;
    private List<Student> students = new ArrayList<Student>();
```

```

public StudentUI(BufferedReader reader, BufferedWriter writer) {
    this.reader = reader;
    this.writer = writer;
}

public void run() throws IOException {
    String line;
    do {
        write(MENU);
        line = reader.readLine();
        if (line.equals(ADD_OPTION))
            addStudent();
    } while (!line.equals(QUIT_OPTION));
}

List<Student> getAddedStudents() {
    return students;
}

private void addStudent() throws IOException {
    write(NAME_PROMPT);
    String name = reader.readLine();

    students.add(new Student(name));
    writeln(ADDED_MESSAGE);
}

private void write(String line) throws IOException {
    writer.write(line, 0, line.length());
    writer.flush();
}

private void writeln (String line) throws IOException {
    write(line);
    writer.newLine();
    writer.flush();
}
}

```

Testing the Application

Testing the Application

The student user interface represents an actual application that someone might want to execute and interact with. You will need to provide a `main` method so that they can start the application. Currently, in order to execute the application, you would need to construct a student UI with `System.in` and `System.out` appropriately wrapped, then call the `run` method.

You can simplify the `main` method by encapsulating this work in the `StudentUI` class itself. This may or may not be an appropriate tactic—if you only

have one user interface class, it's fine. But if you have a dozen related UI classes, each controlling a portion of the user interface, you'd be better off constructing the console wrappers in a single class.

For this example, let's make the `main` method as simple as possible. You can do this by redirecting console input and output using the `System` methods `setIn` and `setOut` instead of wrapping the input and output streams in buffered streams. You must wrap the `ByteArrayOutputStream` in a `PrintStream` in order to call `setOut`.

Testing the Application

```
public void testCreateStudent() throws IOException {
    StringBuffer expectedOutput = new StringBuffer();
    StringBuffer input = new StringBuffer();
    setup(expectedOutput, input);
    byte[] buffer = input.toString().getBytes();

    InputStream inputStream = new ByteArrayInputStream(buffer);
    OutputStream outputStream = new ByteArrayOutputStream();

    InputStream consoleIn = System.in;
    PrintStream consoleOut = System.out;
    System.setIn(inputStream);
    System.setOut(new PrintStream(outputStream));
    try {
        StudentUI ui = new StudentUI();
        ui.run();

        assertEquals(
            expectedOutput.toString(),
            outputStream.toString());
        assertStudents(ui.getAddedStudents());
    }
    finally {
        System.setIn(consoleIn);
        System.setOut(consoleOut);
    }
}
```

You want to make sure you reset `System.in` and `System.out` by using a `try-finally` statement.

The new constructor for `StudentUI` must use an `InputStreamReader` to wrap `stdin` in a `BufferedReader` and an `OutputStreamWriter` to wrap `stdout` in a `BufferedWriter`.

```
public StudentUI() {
    this.reader =
        new BufferedReader(new InputStreamReader(System.in));
    this.writer =
        new BufferedWriter(new OutputStreamWriter(System.out));
}
```

After demonstrating that the test passes, you can now write a `main` method that kicks off the application.

```
public static final void main(String[] args) throws IOException {  
    new StudentUI().run();  
}
```

There are two trains of thought on this. First, it is possible to write a test against the `main` method, since you can call it like any other method (but you must supply an array of `String` objects that represent command-line arguments):

```
StudentUI.main(new String[] {});
```

Data Streams

The other testing philosophy is that the `main` method is virtually unbreakable. It is one line of code. You will certainly run the application from the command line at least once to ensure it works. As long as the `main` method does not change, it can't break, in which case you do not necessarily need a test against the `main` method.

The choice is yours as to whether to test `main` or not. Regardless, you should strive to minimize the `main` method to a single line. Refactor code from the `main` method into either static or instance-side methods. Create utility classes to help you manage command-line arguments. Test the code you move out of `main`.

I hope you've noticed that testing this simple console-based user interface required a good amount of code. Were you to write more of the console application, the work would simplify as you built utility methods and classes to help both testing and production coding.

Data Streams

You can write Java primitive data directly to a `DataOutputStream`. `DataOutputStream` is an example of a filtered stream. A filtered stream *wraps* another stream to either provide additional functionality or to alter the data along the way. The base filtered stream classes are `FilteredOutputStream`, `FilteredInputStream`, `FilteredWriter`, and `FilteredReader`.

The filter in `DataOutputStream` provides methods to output each Java primitive type: `writeBoolean`, `writeDouble`, and so on. It also provides the `writeUTF` method to output a `String`.

CourseCatalog

 The student information system requires a CourseCatalog class to store a list of all available course sessions. CourseCatalog will be responsible for persisting basic course information (department, course number, start date, and number of credits) to a file so that the application can be restarted without any data loss.

The CourseCatalog provides a `load` method that reads all Session objects from a DataOutputStream into a collection. It also provides a `store` method to write the collection to a DataOutputStream.

```
package sis.studentinfo;

import junit.framework.*;
import java.util.*;
import java.io.*;

public class CourseCatalogTest extends TestCase {
    private CourseCatalog catalog;
    private Session session1;
    private Session session2;
    private Course course1;
    private Course course2;

    protected void setUp() {
        catalog = new CourseCatalog();
        course1 = new Course("a", "1");
        course2 = new Course("a", "1");

        session1 =
            CourseSession.create(
                course1, DateUtil.createDate(1, 15, 2005));
        session1.setNumberOfCredits(3);

        session2 =
            CourseSession.create(
                course2, DateUtil.createDate(1, 17, 2005));
        session2.setNumberOfCredits(5);

        catalog.add(session1);
        catalog.add(session2);
    }

    public void testStoreAndLoad() throws IOException {
        final String filename = "CourseCatalogTest.testAdd.txt";
        catalog.store(filename);
        catalog.clearAll();
        assertEquals(0, catalog.getSessions().size());
        catalog.load(filename);
    }
}
```

```

List<Session> sessions = catalog.getSessions();
assertEquals(2, sessions.size());
assertSession(session1, sessions.get(0));
assertSession(session2, sessions.get(1));
}

private void assertSession(Session expected, Session retrieved) {
    assertNotSame(expected, retrieved);
    assertEquals(expected.getNumberOfWorkCredits(),
                retrieved.getNumberOfWorkCredits());
    assertEquals(expected.getStartDate(), retrieved.getStartDate());
    assertEquals(expected.getDepartment(), retrieved.getDepartment());
    assertEquals(expected.getNumber(), retrieved.getNumber());
}
}

```

CourseCatalog

The test loads a couple courses into the catalog, calls the `store` method, clears the catalog, and then calls the `load` method. It asserts that the catalog contains the two courses initially inserted.

The code other than `load` and `store` in `CourseCatalog` is trivial:

```

package sis.studentinfo;

import java.util.*;
import java.io.*;

public class CourseCatalog {
    private List<Session> sessions =
        new ArrayList<Session>();

    public void add(Session session) {
        sessions.add(session);
    }

    public List<Session> getSessions() {
        return sessions;
    }

    public void clearAll() {
        sessions.clear();
    }
    // ...
}

```

The `store` method creates a `DataOutputStream` by wrapping a `FileOutputStream`. It first writes an `int` that represents the number of course sessions to be stored. The method then loops through all sessions, writing the start date, number of credits, department, and course number for each.

```

public void store(String filename) throws IOException {
    DataOutputStream output = null;
    try {

```

```

        output =
            new DataOutputStream(new FileOutputStream(filename));
        output.writeInt(sessions.size());
        for (Session session: sessions) {
            output.writeLong(session.getStartDate().getTime());
            output.writeInt(session.getNumberOfWorkCredits());
            output.writeUTF(session.getDepartment());
            output.writeUTF(session.getNumber());
        }
    }
    finally {
        output.close();
    }
}

```

CourseCatalog

You'll need to create a getter method in Session to return the number of credits:

```

public int getNumberOfWorkCredits() {
    return numberOfWorkCredits;
}

```

The `load` method creates a `DataInputStream` by wrapping a `FileInputStream`. It reads the count of sessions to determine how many sessions are stored in the file. Using counts in this manner is preferred to the alternative, which is to anticipate an end-of-file exception with each read operation against the file.

The `load` method assumes that the session being read is a `CourseSession`, not a `SummerCourseSession` or other `Session` subclass. If the `CourseCatalog` needed to support more than one type, you would need to store the type of `Session`. The type information would allow code in the `load` method to know what class to instantiate when reading each object.

```

public void load(String filename) throws IOException {
    DataInputStream input = null;
    try {
        input = new DataInputStream(new FileInputStream(filename));
        int numberOfSessions = input.readInt();
        for (int i = 0; i < numberOfSessions; i++) {
            Date startDate = new Date(input.readLong());
            int credits = input.readInt();
            String department = input.readUTF();
            String number = input.readUTF();
            Course course = new Course(department, number);
            Session session =
                CourseSession.create(course, startDate);
            session.setNumberOfWorkCredits(credits);
            sessions.add(session);
        }
    }
    finally {
        input.close();
    }
}

```

Both `load` and `store` methods ensure that the associated data streams are closed by use of a `finally` block.

Advanced Streams

Piped Streams

You use piped streams for a safe I/O-based data communication channel between different threads. Piped streams work in pairs: Data written to a piped output stream is read from a piped input stream to which it is attached. The piped stream implementations are `PipedInputStream`, `PipedOutputStream`, `PipedReader`, and `PipedWriter`. Refer to Lesson 13 for more information on multithreading.

Object Streams

SequenceInputStream

You can use a `SequenceInputStream` to allow a collection of input sources to act as a single input stream. The collection of sources is ordered; when one source is fully read, it is closed and the next stream in the collection is opened for reading.

Pushback Streams

The primary use of pushback streams (`PushbackInputStream` and `PushbackReader`) is for lexical analysis programs such as a tokenizer for a compiler. They allow data to be put back onto a stream as if it had not yet been read by the stream.

StreamTokenizer

The primary use of `StreamTokenizer` is also in parsing applications. It acts similarly to `StringTokenizer`. Instead of returning only strings from the underlying stream, however, a `StreamTokenizer` returns the type of the token in addition to the value. A token type may be a word, a number, an end-of-line marker, or an end-of-file marker.

Object Streams

Java provides the capability to directly read and write objects from and to streams. Java can write an object to an object output stream by virtue of *serializing* it. Java serializes an object by converting it into a sequence of bytes.

Object Streams

The ability to serialize objects is the basis for Java's RMI (Remote Method Invocation) technology. RMI allows objects to communicate with other objects on remote systems as if they were local. RMI in turn provides the basis for Java's EJB (Enterprise Java Bean) technology for component-based computing.

You write and read objects using the classes `ObjectOutputStream` and `ObjectInputStream`. As a quick demonstration, the following code is a rewrite of the methods `store` and `load` in the `CourseCatalog` class. The modified code uses object streams instead of data streams.

```
public void store(String filename) throws IOException {
    ObjectOutputStream output = null;
    try {
        output =
            new ObjectOutputStream(new FileOutputStream(filename));
        output.writeObject(sessions);
    }
    finally {
        output.close();
    }
}

public void load(String filename)
    throws IOException, ClassNotFoundException {
    ObjectInputStream input = null;
    try {
        input = new ObjectInputStream(new FileInputStream(filename));
        sessions = (List<Session>)input.readObject();
    }
    finally {
        input.close();
    }
}
```

The test remains largely unchanged.

```
public void testStoreAndLoad() throws Exception {
    final String filename = "CourseCatalogTest.testAdd.txt";
    catalog.store(filename);
    catalog.clearAll();
    assertEquals(0, catalog.getSessions().size());
    catalog.load(filename);

    List<Session> sessions = catalog.getSessions();
    assertEquals(2, sessions.size());
    assertSession(session1, sessions.get(0));
    assertSession(session2, sessions.get(1));
}
```

The throws clause on the test method signature must change, since the `load` method now throws a `ClassNotFoundException`. Within the context of this example, it doesn't seem possible for a `ClassNotFoundException` to be generated. You store a `List` of `Session` objects and immediately read it back, and both `java.util.List` and `Session` are known to your code. An exception could be thrown, however, if another application with no access to your `Session` class were to read the objects from the file.

When you run the test, you should receive an exception:

```
java.io.NotSerializableException: studentinfo.CourseSession
```

Object Streams

In order to write an object to an object stream, its class must be *serializable*. You mark a class as serializable by having it implement the interface `java.io.Serializable`. Most of the classes in the Java system class library that you would expect to be serializable are already marked as such. This includes the `String` and `Date` classes as well as all collection classes (`HashMap`, `ArrayList`, and so on). But you will need to mark your own application classes:

```
abstract public class Session
    implements Comparable<Session>,
    Iterable<Student>,
    java.io.Serializable { ...}
```

Don't forget the `Course` class, since `Session` encapsulates it:

```
public class Course implements java.io.Serializable { ...}
```

When you mark the abstract superclass as serializable, all its subclasses will also be serializable. The `Serializable` interface contains no method definitions, so you need not do anything else to the `Session` class.

An interface that declares no methods is known as a *marker interface*. You create marker interfaces to allow a developer to explicitly mark a class for a specific use. You must positively designate a class as capable of being serialized. The `Serializable` marker is intended as a safety mechanism—you may want to prevent certain objects from being serialized for security reasons.

Transient

`Course` sessions allow for enrollment of students. However, you don't want the course catalog to be cluttered with student objects. Suppose, though, that a student has enrolled early, before the catalog was created. The `setUp` method in `CourseCatalogTest` enrolls a student as an example:

```

protected void setUp() {
    catalog = new CourseCatalog();
    course1 = new Course("a", "1");
    course2 = new Course("a", "1");

    session1 =
        CourseSession.create(
            course1, DateUtil.createDate(1, 15, 2005));
    session1.setNumberOfCredits(3);

    session2 =
        CourseSession.create(
            course2, DateUtil.createDate(1, 17, 2005));
    session2.setNumberOfCredits(5);
    session2.enroll(new Student("a"));

    catalog.add(session1);
    catalog.add(session2);
}

```

Object Streams

If you run your tests, you again receive a `NotSerializableException`. The course session now refers to a `Student` object that must be serialized. But the `Student` class does not implement `java.io.Serializable`.

Instead of changing `Student`, you can indicate that the list of students in `Session` is to be skipped during serialization by marking them with the `transient` modifier.

```

abstract public class Session
    implements Comparable<Session>,
    Iterable<Student>,
    java.io.Serializable {

    ...
    private transient List<Student> students = new ArrayList<Student>();
}

```

The list of students will not be serialized in this example. Your tests will now pass.

Serialization and Change

Serialization makes it easy to persist objects. Too easy, perhaps. There are many implications to declaring a class as `Serializable`. The most significant issue is that when you persist a serialized object, you export with it a definition of the class as it currently exists. If you subsequently change the class definition, then attempt to read the serialized object, you will get an exception.

To demonstrate, add a `name` field to `Session.java`. This creates a new version of the `Session` class. Don't worry about a test; this is a temporary "spike," or experiment. You will delete this line of code in short time. Also, *do not run any tests*—doing so will ruin the experiment.

```
abstract public class Session
    implements Comparable<Session>,
    Iterable<Student>,
    java.io.Serializable {
    private String name;
    ...
}
```

Your last execution of tests persisted an object stream to the file named `CourseCatalogTest.testAdd.txt`. The object stream stored in this file contains `Session` objects created using the older definition of `Session` without the `name` field.

Then create an entirely new test class, `studentinfo.SerializationTest`:

```
package sis.studentinfo;

import junit.framework.*;

public class SerializationTest extends TestCase {
    public void testLoadToNewVersion() throws Exception {
        CourseCatalog catalog = new CourseCatalog();
        catalog.load("CourseCatalogTest.testAdd.txt");
        assertEquals(2, catalog.getSessions().size());
    }
}
```

Object Streams

The test tries to load the persisted object stream. *Execute only this test.* Do not execute your `AllTests` suite. You should receive an exception that looks something like:

```
testLoadToNewVersion(studentinfo.SerializationTest)
java.io.InvalidClassException: studentinfo.Session;
local class incompatible:
stream classdesc serialVersionUID = 5771972560035839399,
local class serialVersionUID = 156127782215802147
```

Java determines compatibility between the objects stored in the output stream and the existing (local) class definition. It considers the class name, the interfaces implemented by the class, its fields, and its methods. Changing any of these will result in incompatibility.

Transient fields are ignored, however. If you change the declaration of the `name` field in `Session` to `transient`:

```
private transient String name;
```

`SerializationTest` will then pass.

Serial Version UID

The `InvalidClassException` you received referred to a `serialVersionUID` for both the stream's class definition and the local (current Java) class definition. In order to determine whether the definition of a class has changed, Java generates the `serialVersionUID` based on the class name, interfaces implemented, fields, and methods. The `serialVersionUID`, a 64-bit `long` value, is known as a *stream unique identifier*.

Object Streams

You can choose to define your own `serialVersionUID` instead of using the one Java generates. This may give you some ability to better control version management. You can obtain an initial `serialVersionUID` by using the command-line utility `serialver` or you can assign an arbitrary value to it. An example execution of `serialver`:

```
serialver -classpath classes studentinfo.Session
```

You optionally specify the classpath, followed by the list of classes for which you wish to generate a `serialVersionUID`.

Remove the `name` field from `Session`. Rebuild and rerun your entire test suite. Add a `serialVersionUID` definition to `Session`. At the same time, add back the `name` field.

```
abstract public class Session
    implements Comparable<Session>,
    Iterable<Student>,
    java.io.Serializable {
    public static final long serialVersionUID = 1L;
    private String name;
    ...
```

Then run only `SerializationTest`. Even though you've added a new field, the version ID is the same. Java will initialize the `name` field to its default value of `null`. If you change the `serialVersionUID` to `2L` and rerun the test, you will cause the stream version (1) to be out of synch with the local class version (2).

Creating a Custom Serialized Form

Your class may contain information that can be reconstructed based on other data in the class. When you model a class, you define its attributes to represent the logical state of every object of that class. In addition to those attributes, you may have data structures or other fields that cache dynamically computed information. Persisting this dynamically calculated data may be slow and/or a grossly inefficient use of space.



Suppose you need to persist not only the course sessions but also the students enrolled in each session. Students carry a large amount

of additional data, and they are already being persisted elsewhere. You can traverse the collection of course sessions and persist only the unique identifier for each student to the object stream.¹ When you load this compacted collection, you can execute a lookup to retrieve the complete student object and store it in the course session.

To accomplish this, you will define two methods in `Session`, `writeObject` and `readObject`. These methods are hooks that the serialization mechanism calls when reading and writing each object to the object stream. If you don't supply anything for these hooks, default serialization and deserialization takes place.

First, change the test in `CourseCatalogTest` to ensure that the enrolled student was properly persisted and restored.

```
public void testStoreAndLoad() throws Exception {
    final String filename = "CourseCatalogTest.testAdd.txt";
    catalog.store(filename);
    catalog.clearAll();
    assertEquals(0, catalog.getSessions().size());
    catalog.load(filename);

    List<Session> sessions = catalog.getSessions();
    assertEquals(2, sessions.size());
    assertSession(session1, sessions.get(0));
    assertSession(session2, sessions.get(1));

    Session session = sessions.get(1);
    assertSession(session2, session);
    Student student = session.getAllStudents().get(0);
    assertEquals("a", student.getLastname());
}
```

Make sure that the `students` field in `Session` is marked as transient. Then code the `writeObject` definition for `Session`:

```
private void writeObject(ObjectOutputStream output)
    throws IOException {
    output.defaultWriteObject();
    output.writeInt(students.size());
    for (Student student: students)
        output.writeObject(student.getLastname());
}
```

¹We have a small school. We don't admit anyone with the same last name as another student, so you can use that as your unique identifier.

The first line of `writeObject` calls the method `defaultWriteObject` on the stream. This will write every nontransient field to the stream normally. Subsequently, the code in `writeObject` first writes the number of students to the stream, then loops through the list of students, writing each student's last name to the stream.

```
private void readObject(ObjectInputStream input)
    throws Exception {
    input.defaultReadObject();
    students = new ArrayList<Student>();
    int size = input.readInt();
    for (int i = 0; i < size; i++) {
        String lastName = (String)input.readObject();
        students.add(Student.findByName(lastName));
    }
}
```

Object Streams

On the opposite end, `readObject` first calls `defaultReadObject` to load all nontransient fields from the stream. It initializes the transient field `students` to a new `ArrayList` of students. It reads the number of students into `size` and iterates `size` times. Each iteration extracts a student's last name from the stream. The code looks up and retrieves a `Student` object using this last name and stores the `Student` in the `students` collection.

In real life, the `findByName` method might involve sending a message to a student directory object, which in turn retrieves the appropriate student from a database or another serialization file. For demonstration purposes, you can provide a simple implementation that will pass the test:

```
public static Student findByName(String lastName) {
    return new Student(lastName);
}
```

Serialization Approaches

For classes whose definitions are likely to change, dealing with serialization version incompatibility issues can be a major headache. While it is possible to load serialized objects from an older version, it is difficult. Your best tactics include:

- minimizing use of serialization
- maximizing the number of transient fields
- identifying versions with `serialVersionUID`
- defining a custom serialization version

When you serialize an object, you are exporting its interface. Just as you should keep interfaces as abstract and unlikely to change as possible, you should do the same with serializable classes.

Random Access Files

Instead of loading and storing the complete course catalog each time you execute the application, you can dynamically interact with the catalog by implementing it as a *random access file*. A random access file allows you to quickly seek to specific positions in the file and either read from or write to that position.

It would be possible to create a fully featured object database in Java using random access files. The forthcoming example code is a starting point.

In Lesson 9, you created a `StudentDirectory` class. You implemented the class to store student objects in a `HashMap`, using the student's ID as a key.



Now you need to ensure that your student directory supports the tens of thousands of students enrolling at the university. Further, you must persist the directory to the file system in order to secure the data. It is imperative that retrieving students from the directory executes rapidly. A retrieval must execute in constant time—the amount of time to access a student should not vary depending on where the student appears in the file.

You will implement the student directory using a simple indexed file system. You will store student records in a data file and unique ids (identifiers) for each student in an index file. As you insert a serialized student into the data file, you will record its id, position, and length in the index file. The index file will be small compared to the data file size. It can be quickly loaded into memory and written out when the data file is closed.²

The example code is the most involved in this book so far. If you take it a test and method at a time, you shouldn't have much trouble understanding it. Building the tests and code from scratch is a bit more of a challenge. The UML is shown in Figure 11.1.

And here is the code. I'll explain the interesting parts of it after the code listing for each class.

²There is a bit of risk in not persisting the indexes as you add Students. You could mitigate this risk by writing the data length within the data file itself as well. Doing so would allow you to re-create the index file by traversing through the data file.

Random Access
Files

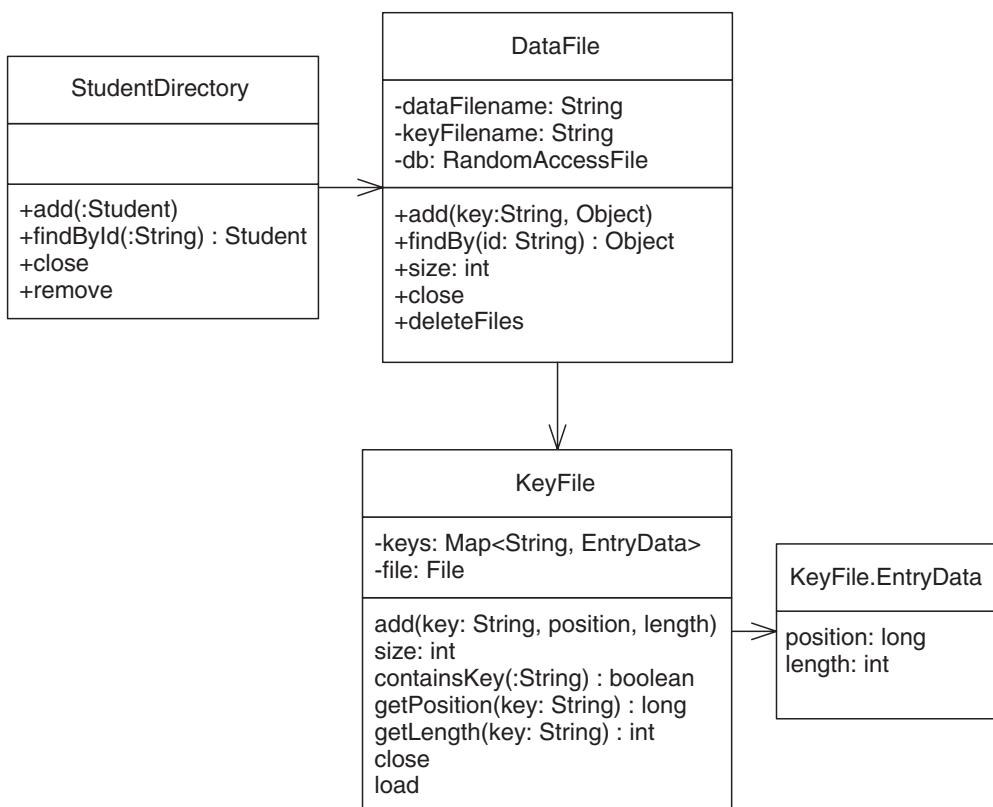


Figure 11.1 *The Student Directory*

The Student Directory

```

package sis.studentinfo;

import junit.framework.*;
import java.io.*;

public class StudentDirectoryTest extends TestCase {
    private StudentDirectory dir;

    protected void setUp() throws IOException {
        dir = new StudentDirectory();
    }

    protected void tearDown() throws IOException {
        dir.close();
        dir.remove();
    }
}
  
```

```

public void testRandomAccess() throws IOException {
    final int numberofStudents = 10;
    for (int i = 0; i < numberofStudents; i++)
        addStudent(dir, i);
    dir.close();

    dir = new StudentDirectory();
    for (int i = 0; i < numberofStudents; i++)
        verifyStudentLookup(dir, i);
}

void addStudent(StudentDirectory directory, int i)
    throws IOException {
    String id = "" + i;
    Student student = new Student(id);
    student.setId(id);
    student.addCredits(i);
    directory.add(student);
}

void verifyStudentLookup(StudentDirectory directory, int i)
    throws IOException {
    String id = "" + i;
    Student student = dir.findById(id);
    assertEquals(id, student.getLastName());
    assertEquals(id, student.getId());
    assertEquals(i, student.getCredits());
}
}

```

The Student
Directory

The most significant new addition to `StudentDirectoryTest` appears in `testRandomAccess`. When it adds the students to the directory, the test closes it. It then creates a new directory instance to be used for the student lookups. By doing this, the test demonstrates at least some notion of persistence.

An additional performance test might be worthwhile to demonstrate that a lookup into the directory takes the same amount of time regardless of where it appears in the file. Additions of students to the directory should also execute in constant time.

```

package sis.studentinfo;

import java.io.*;
import sis.db.*;

public class StudentDirectory {
    private static final String DIR_BASENAME = "studentDir";
    private DataFile db;

    public StudentDirectory() throws IOException {
        db = DataFile.open(DIR_BASENAME);
    }
}

```

```

public void add(Student student) throws IOException {
    db.add(student.getId(), student);
}

public Student findById(String id) throws IOException {
    return (Student)db.findById(id);
}

public void close() throws IOException {
    db.close();
}

public void remove() {
    db.deleteFiles();
}
}

```

**sis.db.DataFile-
Test**

In contrast, most of the StudentDirectory class has changed. The StudentDirectory class now encapsulates a DataFile instance to supply directory functionality. It provides a few additional specifics, including the key field to use (the student id) and the base filename for the data and key files. Beyond that, the class merely delegates messages to the DataFile object.

sis.db.DataFileTest

```

package sis.db;

import junit.framework.*;
import java.io.*;
import sis.util.*;

public class DataFileTest extends TestCase {
    private static final String ID1 = "12345";
    private static final String ID2 = "23456";
    private static final String FILEBASE = "DataFileTest";

    private DataFile db;
    private TestData testData1;
    private TestData testData2;

    protected void setUp() throws IOException {
        db = DataFile.create(FILEBASE);
        assertEquals(0, db.size());
    }

    testData1 = new TestData(ID1, "datum1a", 1);
    testData2 = new TestData(ID2, "datum2a", 2);
}

```

```
protected void tearDown() throws IOException {
    db.close();
    db.deleteFiles();
}

public void testAdd() throws IOException {
    db.add(ID1, testData1);
    assertEquals(1, db.size());

    db.add(ID2, testData2);
    assertEquals(2, db.size());

    assertTestDataEquals(testData1, (TestData)db.findById(ID1));
    assertTestDataEquals(testData2, (TestData)db.findById(ID2));
}

public void testPersistence() throws IOException {
    db.add(ID1, testData1);
    db.add(ID2, testData2);
    db.close();

    db = DataFile.open(FILEBASE);
    assertEquals(2, db.size());

    assertTestDataEquals(testData1, (TestData)db.findById(ID1));
    assertTestDataEquals(testData2, (TestData)db.findById(ID2));

    db = DataFile.create(FILEBASE);
    assertEquals(0, db.size());
}

public void testKeyNotFound() throws IOException {
    assertNull(db.findById(ID2));
}

private void assertTestDataEquals(
    TestData expected, TestData actual) {
    assertEquals(expected.id, actual.id);
    assertEquals(expected.field1, actual.field1);
    assertEquals(expected.field2, actual.field2);
}

static class TestData implements Serializable {
    String id;
    String field1;
    int field2;
    TestData(String id, String field1, int field2) {
        this.id = id;
        this.field1 = field1;
        this.field2 = field2;
    }
}
```

sis.db.DataFile-
Test

DataFileTest shows that you create a new DataFile using the static factory method `create`. The `create` method takes the name of the DataFile as its parameter.

The test also shows that you insert objects into a DataFile by using the `add` method, which takes a unique key and associated object as parameters. To retrieve objects, you send the message `findBy`, passing with it the unique id of the object to be retrieved. If the object is not available, DataFile returns `null`.

Persistence is tested by closing a DataFile and creating a new instance using the static factory method `open`.³ The distinction between `open` and `create` is that `create` will delete the data file if it already exists, while `open` will reuse an existing data file or create a new one if necessary.

Note that the object returned by the `findBy` method requires a cast! This suggests that DataFile is a candidate for implementing as a parameterized type. (See Lesson 14 for more information on parameterized types.)

Static Nested Classes and Inner Classes

DataFileTest needs to show that DataFile can persist an object and retrieve that object at a later time. In order to write such a test, it's best to use a class that you can guarantee no one else will change. You could use a Java system library class such as `String`, but they change with new releases of Java. Instead, the better solution is to create a test class solely for use by DataFileTest.

The class `TestData` is defined as a *nested* class of `DataFileTest`; that is, it is completely contained within `DataFileTest`. There are two kinds of nested classes: inner classes and static nested classes. The chief distinction is that inner classes have access to the instance variables defined within the enclosing class. Static nested classes do not.

Another distinction is that inner classes are completely encapsulated by the enclosing class. Since inner classes can refer to instance variables of the enclosing class, it makes no sense for other code to be able to create instances of an instance inner class. While Java technically allows you to refer to an

³Technically the test does not *prove* disk persistence. You could have implemented a solution that stored objects in a static-side collection. However, the point of the test is not to protect you from being dishonest and coding a foolish solution. The test instead demonstrates the expected behavior. If you're not convinced, however, there's nothing that prohibits you from writing a test to ensure that the object is actually stored in a disk file. It's just a lot more complex, and probably unnecessary.

Class Names for Nested Classes

When you compile a class or interface, the compilation unit is named by taking the class name and appending a .class file extension. For example, the Course class compiles to Course.class.

When compiling a class containing a nested class, Java names the compilation unit for the inner class by first using the containing class name, followed by the dollar sign (\$), then the nested class name.

For example, the DataFileTest class creates two compilation units: DataFileTest.class and DataFileTest\$TestData.class.

When you copy class files or put them in a JAR file (see Additional Lesson III) for distribution, don't forget to include all the nested classes.

sis.db.DataFile

inner class from external code, it's a bit of trickery that I'm not going to show you here—don't do it!

Static nested classes, on the other hand, can be used by external code as long as the access specifier is not private. You have already used the static nested class Entry in order to iterate key-value pairs in a Map object. You referred to this class as Map.Entry, since you used it in a code context other than Map.

So the first reason to declare a nested class as static is to allow other classes to use it. You could declare the class as a top-level (i.e., non-nested) class, but you may want to tightly couple it to the containing class. For example, Map.Entry is tightly bound to Map, since it makes no sense for the Entry class to exist in the absence of Map.

The second reason to declare a nested class as static is to allow it to be serialized. You cannot serialize inner class objects, since they must have access to the instance variables of the enclosing class. In order to make it work, the serialization mechanism would have to capture the fields of the enclosing class. Yuck.

Since you need to persist TestData objects, you must make the class serializable. If TestData is to be a nested class, you must declare it as static.

sis.db.DataFile

```
package sis.db;

import java.util.*;
import java.io.*;
import sis.util.*;
```

```
public class DataFile {
    public static final String DATA_EXT = ".db";
    public static final String KEY_EXT = ".idx";

    private String datafilename;
    private String keyfilename;

    private RandomAccessFile db;
    private KeyFile keys;

    public static DataFile create(String filebase) throws IOException {
        return new DataFile(filebase, true);
    }

    public static DataFile open(String filebase) throws IOException {
        return new DataFile(filebase, false);
    }

    private DataFile(String filebase, boolean deleteFiles)
        throws IOException {
        datafilename = filebase + DATA_EXT;
        keyfilename = filebase + KEY_EXT;

        if (deleteFiles)
            deleteFiles();
        openFiles();
    }

    public void add(String key, Object object) throws IOException {
        long position = db.length();

        byte[] bytes = getBytes(object);
        db.seek(position);
        db.write(bytes, 0, bytes.length);

        keys.add(key, position, bytes.length);
    }

    public Object findBy(String id) throws IOException {
        if (!keys.containsKey(id))
            return null;

        long position = keys.getPosition(id);
        db.seek(position);

        int length = keys.getLength(id);
        return read(length);
    }

    public int size() {
        return keys.size();
    }
}
```

```
public void close() throws IOException {
    keys.close();
    db.close();
}

public void deleteFiles() {
    IOUtil.delete(dataFilename, keyFilename);
}

private Object read(int length) throws IOException {
    byte[] bytes = new byte[length];
    db.readFully(bytes);
    return readObject(bytes);
}

private Object readObject(byte[] bytes) throws IOException {
    ObjectInputStream input =
        new ObjectInputStream(new ByteArrayInputStream(bytes));
    try {
        try {
            return input.readObject();
        }
        catch (ClassNotFoundException unlikely) {
            // but write a test for it if you must
            return null;
        }
    }
    finally {
        input.close();
    }
}

private void openFiles() throws IOException {
    keys = new KeyFile(keyFilename);
    db = new RandomAccessFile(new File(dataFilename), "rw");
}

private byte[] getBytes(Object object) throws IOException {
    ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
    ObjectOutputStream outputStream =
        new ObjectOutputStream(byteStream);
    outputStream.writeObject(object);
    outputStream.flush();
    return byteStream.toByteArray();
}
```

sis.db.DataFile

The DataFile class is the core part of the solution. It demonstrates use of a RandomAccessFile object, which you store in the instance variable `db` (short for database, one of the small number of abbreviations I use). You create a RandomAccessFile object by passing in a File object and a String declaring the mode in which you wish to access the RandomAccessFile.

RandomAccessFile provides four modes: "r" (read-only), "rw" (read and write access), "rws" (read-write with synchronous data/metadata updates), and "rwd" (read-write with synchronous data updates). The synchronous updates ensure that updates to the file are safely written to the underlying storage device. Without using them, you could conceivably lose data in a system crash. The "rws" option ensures persistence of both content and metadata (information such as the last modified timestamp for the file), while the "rwd" option only ensures updates of the content.

Here we chose the "rw" option, since you want to be able to both read data from and write data to the data file. The other read-write options incur additional overhead but may be necessary to maintain data integrity.

You use the seek method to rapidly move an internal file pointer to any location within the underlying file. The getFilePointer method (not used here) returns the current position of the file pointer. The length method returns the total number of bytes in the file. Like other IO classes, RandomAccessFile supplies many read and write methods to extract and store data.

RandomAccessFile does not directly support storing objects. In order to persist an object to the RandomAccessFile, you must first convert it to a byte array. To convert an object to bytes, the getBytes method wraps a ByteArrayOutputStream in an ObjectOutputStream. This means that any objects written to the ObjectOutputStream are piped to the underlying ByteArrayOutputStream. You can extract the bytes from an ByteArrayOutputStream by sending it the toByteArray message.

To read an object from the RandomAccessFile, you must do the opposite. Create a byte array of the appropriate length and use the method readFully to populate the bytes from the RandomAccessFile. Wrap the populated byte array in a ByteArrayInputStream, which you then wrap in an ObjectInputStream. Reading from the ObjectInputStream will reconstitute the persisted object using the underlying bytes.

sis.db.KeyFileTest

```
package sis.db;

import junit.framework.*;
import java.io.*;
import java.util.*;
import sis.util.*;
```

```

public class KeyFileTest extends TestCase {
    private static final String FILENAME = "keyfiletest.idx";
    private static final String KEY = "key";
    private static final long POSITION = 1;
    private static final int LENGTH = 100;

    private KeyFile keyFile;

    protected void setUp() throws IOException {
        TestUtil.delete(FILENAME);
        keyFile = new KeyFile(FILENAME);
    }

    protected void tearDown() throws IOException {
        TestUtil.delete(FILENAME);
        keyFile.close();
    }

    public void testCreate() {
        assertEquals(0, keyFile.size());
    }

    public void testAddEntry() {
        keyFile.add(KEY, POSITION, LENGTH);

        assertEquals(1, keyFile.size());
        assertTrue(keyFile.containsKey(KEY));
        assertEquals(POSITION, keyFile.getPosition(KEY));
        assertEquals(LENGTH, keyFile.getLength(KEY));
    }

    public void testReopen() throws IOException {
        keyFile.add(KEY, POSITION, LENGTH);
        keyFile.close();

        keyFile = new KeyFile(FILENAME);
        assertEquals(1, keyFile.size());
        assertEquals(POSITION, keyFile.getPosition(KEY));
        assertEquals(LENGTH, keyFile.getLength(KEY));
    }
}

```

sis.db.KeyFileTest



KeyFileTest demonstrates the ability to add keys (unique ids) to a KeyFile. A key is stored with the position of the associated data in the DataFile object as well as with the length of that data. The data position and length can be retrieved from the KeyFile using the unique key.

The third test, testReopen, ensures that you can create a new KeyFile object using the name of an existing key file. The KeyFile object must load the already-persisted key data.

sis.db.KeyFile

sis.db.KeyFile

```
package sis.db;

import java.util.*;
import java.io.*;

class KeyFile {
    private Map<String, EntryData> keys =
        new HashMap<String, EntryData>();
    private File file;

    KeyFile(String filename) throws IOException {
        file = new File(filename);
        if (file.exists())
            load();
    }

    void add(String key, long position, int length) {
        keys.put(key, new EntryData(position, length));
    }

    int size() {
        return keys.size();
    }

    boolean containsKey(String key) {
        return keys.containsKey(key);
    }

    long getPosition(String key) {
        return keys.get(key).getPosition();
    }

    int getLength(String key) {
        return keys.get(key).getLength();
    }

    void close() throws IOException {
        ObjectOutputStream stream =
            new ObjectOutputStream(new FileOutputStream(file));
        stream.writeObject(keys);
        stream.close();
    }

    void load() throws IOException {
        ObjectInputStream input = null;
        try {
            input = new ObjectInputStream(new FileInputStream(file));
            try {

```

```

        keys = (Map<String, EntryData>)input.readObject();
    }
    catch (ClassNotFoundException e) {
    }
}
finally {
    input.close();
}
}

static class EntryData implements Serializable {
    private long position;
    private int length;

    EntryData(long position, int length) {
        this.position = position;
        this.length = length;
    }

    private long getPosition() {
        return position;
    }

    private int getLength() {
        return length;
    }
}
}

```

sis.util.IOUtil-Test

KeyFile stores the key information using a Map named keys. This Map object maps the key to a serializable static nested class, EntryData, which contains the data position and length. When closed, the KeyFile writes the entire Map to the file by using an ObjectOutputStream. It loads the entire Map when opened.

sis.util.IOUtilTest

```

package sis.util;

import junit.framework.*;
import java.io.*;

public class IOUtilTest extends TestCase {
    static final String FILENAME1 = "IOUtilTest1.txt";
    static final String FILENAME2 = "IOUtilTest2.txt";

    public void testDeleteSingleFile() throws IOException {
        create(FILENAME1);
    }
}

```

```

        assertTrue(IOUtil.delete(FILENAME1));
        TestUtil.assertGone(FILENAME1);
    }

    public void testDeleteMultipleFiles() throws IOException {
        create(FILENAME1, FILENAME2);
        assertTrue(IOUtil.delete(FILENAME1, FILENAME2));
        TestUtil.assertGone(FILENAME1, FILENAME2);
    }

    public void testDeleteNoFile() throws IOException {
        TestUtil.delete(FILENAME1);
        assertFalse(IOUtil.delete(FILENAME1));
    }

    public void testDeletePartiallySuccessful() throws IOException {
        create(FILENAME1);
        TestUtil.delete(FILENAME2);
        assertFalse(IOUtil.delete(FILENAME1, FILENAME2));
        TestUtil.assertGone(FILENAME1);
    }

    private void create(String... filenames) throws IOException {
        for (String filename: filenames) {
            TestUtil.delete(filename);
            new File(filename).createNewFile();
        }
    }
}

```

The most interesting aspect of IOUtilTest is that it contains four test methods, each testing the same IOUtil method `delete`. Each test proves a typical scenario. There are probably many more tests possible. It's up to you to decide whether you have enough tests to give you the confidence you need in your code.



Err in the direction of too many tests instead of too few.

sis.util.IOUtil

```

package sis.util;

import java.io.*;

public class IOUtil {
    public static boolean delete(String... filenames) {
        boolean deletedAll = true;

```

```

        for (String filename: filenames)
            if (!new File(filename).delete())
                deletedAll = false;
        return deletedAll;
    }
}

```

The `delete` method uses varargs to allow you to delete multiple files in a single method call. It returns true only if all files were successfully deleted.

sis.util.TestUtil

sis.util.TestUtil

```

package sis.util;

import junit.framework.*;
import java.io.*;

public class TestUtil {
    public static void assertGone(String... filenames) {
        for (String filename: filenames)
            Assert.assertFalse(new File(filename).exists());
    }

    public static void delete(String filename) {
        File file = new File(filename);
        if (file.exists())
            Assert.assertTrue(file.delete());
    }
}

```

Only tests use code in the `TestUtil` class. Since `TestUtil` does not extend from `junit.framework.TestCase`, it does not have instance access to any `assert` methods. However, the class `junit.framework.Assert` defines the `assert` methods as class methods, allowing you to access them anywhere.

Other things you will need to do:

- Make the `GradingStrategy` implementation types and `Student` serializable.
- Add an `id` field and associated getter/setter methods to the `Student` class.
- Update the `AllTests` suite classes.⁴

⁴Having to remember to add the classes is asking for trouble. In Lesson 12, you'll learn how to use Java's reflections capabilities to dynamically build test suites.

Exercises

Developing the Solution

In this example, I started by trying to implement `StudentDirectory` through the single test you see in `StudentDirectoryTest`, `testRandomAccess`. As often happens, the task of building functionality for `StudentDirectory` represented a large step. As I stubbed out the methods in `StudentDirectory`, it led me to the task of building the `DataFile` class. This meant my work on `StudentDirectoryTest` was temporarily put on hold. Similarly, after I started to work on `DataFile`, I found it easier to encapsulate the key functionality to a class named `KeyFile`.

As I progressed in developing the solution, I was constantly reevaluating the design at hand. My general strategy is to sketch an initial design going from the “outside” and moving in. In other words, I determine how client code wants to be able to use a class and base its design on that interface. If necessary, I sketch out the high-level structure and work on some of the “inner” details. No matter how I work, I find that the design is usually in a state of flux. Implementing inner details sometimes impacts the surrounding classes, and vice versa.

Changes to the design are normal. The outermost interface remains fairly stable, but the details change frequently. Few of the design changes are major—it’s more like pushing small bits of code from here to there, constantly honing and improving the design.

Doing an initial design can still be very valuable. Just don’t invest much time in the details. The important parts of the design to get right are the interface points—where does your code have to interact with other code in the system? Beyond that, the pieces in the middle are under your control. You’ll craft a better system by letting the implementation guide your design.

Exercises

1. Create a test to write the text of this exercise to the file system. The test should read the file back in and make assertions about the content. Ensure that you can run the test multiple times and have it pass. Finally, make sure that there are no leftover files when the test finishes, even if an exception is thrown.
2. (hard) Create a timing test to prove that using a `Buffered` class is important for performance. The test can loop through various sizes of

file, creating character data in sizes growing by a factor of 10, calling a method coded to write using the basic character-at-a-time methods, then wrapping the writer in a buffered output stream and writing it again until a 5x performance gain is reached. What is the threshold at which a buffered writer is a significant performance gain?

3. Create and test a utility class called `MyFile`. This class should wrap a `File` object, taking a string filename as its constructor argument. It should have methods for retrieving the content of the file as a `String` or as a `List` of lines. It should also have a method for writing either a `String` or a `List` of `Strings`. Read and write operations should encapsulate opening and closing the file—clients should not have to close the file themselves.

Ensure that the read methods fail with a specific unchecked exception type if the file doesn't exist. Similarly, the write methods should fail if the file does exist. Provide `delete` and `overwrite` methods, and you will have built a well-tested utility class that you can place in your toolbox.

4. Further adventures in utility classes: Create a `Dir` class that encapsulates a `File` object that in turn represents an actual file system directory. Design the class so that it is functional only when mapped to an existing directory. Provide a method named `ensureExists` to create the directory if it does not exist. The constructor should throw an exception if the directory name is the same as an existing file. Finally, provide a method that returns a list of `MyFile` objects in the directory and throws an exception if the directory has not been created yet.
5. Code a test that shows the use of a `ByteArrayOutputStream` to capture an exception and dump the stack trace into a string. Code it with and without an `OutputStreamWriter`. In both the character version and the byte version, use buffered readers and writers.
6. Modify the chess application to allow you to save the board positions to a text file and read them back in. Provide two choices—a serialized object of type `Board` or a textual representation as shown in the earlier exercises.
7. In Additional Lesson III, you will learn the preferred Java technique for cloning, or creating a copy of, an object. Until you learn this technique, you can implement cloning using object serialization and deserialization to duplicate objects. Your implementation will be a “poor man's” clone that you should not use in production systems.
8. Create an instance inner class for `Dir`, named `Attributes`, that encapsulates two directory attributes: Is the directory read-only and is it hidden? The `Dir` class should return an instance of this object if

Exercises

Exercises

requested. Demonstrate (through failure to compile) that the test cannot instantiate this object.

9. Change the Dir.Attributes inner class to a static nested class and change the code and test so they still work. What are the implications? Show that the test can instantiate a Dir.Attributes class. Does this design make sense?
10. In the exercises for Lesson 10, you wrote code to programmatically determine the size of each primitive integral type. Now, write code to determine the base size of *all* primitive types by using a data stream.

Lesson 12

Reflection and Other Advanced Topics

Francis Katasani
2800703
Mock Objects Revisited

In terms of learning enough Java to be able to tackle most problems, you're just about there! This chapter presents some advanced Java topics as well as a few important odds and ends that haven't made their way into your student information system.

You will learn about:

- additional mocking techniques
- anonymous inner classes
- reflection
- adapters
- instance initializers
- the class `Class`
- the dynamic proxy mechanism

Mock Objects Revisited

In Lesson 10, you created a mock class to force the behavior of a random number generator. The mock class you built extended from the class `java.util.Random`. You built the mock class as a nested class within the test. Since the mock class was used solely by the test, it was beneficial to include the mock code directly within the test class. In this lesson, you'll learn an even more succinct technique: embedding the mock class completely within a test method.

You might need to use mock objects if your code must interact with an external API. You typically have little control over the external code or the results an API returns. Writing tests against such code often results in breakage due to changing results from the API. Also, the API might communicate with an external resource that is not always available. Such an API introduces a bad dependency that mocks can help you manage.¹



Ensure that you have a dependency issue before introducing mocks.



The student Account class needs to be able to handle transfers from an associated bank account. To do so, it must interact with ACH (automated clearing house) software from Jim Bob's ACH Shop. The ACH software publishes an API specification. Another bad dependency: We haven't licensed or installed the actual software yet! But you need to start coding now in order to be able to ship working software soon after the actual code is licensed and delivered.

The test itself is small.

```
package sis.studentinfo;

import java.math.BigDecimal;
import junit.framework.*;

public class AccountTest extends TestCase {
    static final String ABA = "102000012";
    static final String ACCOUNT_NUMBER = "194431518811";

    private Account account;

    protected void setUp() {
        account = new Account();
        account.setBankAba(ABA);
        account.setBankAccountNumber(ACCOUNT_NUMBER);
        account.setBankAccountType(Account.BankAccountType.CHECKING);
    }
    // ...
    public void testTransferFromBank() {
        account.setAch(new com.jimbob.ach.JimBobAch()); // uh-oh

        final BigDecimal amount = new BigDecimal("50.00");
        account.transferFromBank(amount);

        assertEquals(amount, account.getBalance());
    }
}
```

¹[Langr2003].



Debiting an account requires the bank's ABA (routing) number, the account number, and the type of account. You can populate this required account information using the `setUp` method.

The test method, `testTransferFromBank`, creates a new `JimBobAch` object and passes it into the `Account` object. Uh-oh; you don't have the `JimBobAch` class yet, so the code won't even compile. You'll solve this problem shortly with a mock class.

The remainder of the test sends the `transferFromBank` message to the account, then asserts that the account balance increased to the correct amount.

So how will you solve the problem of the nonexistent `JimBobAch` class? You *do* have just about everything else you need. Jim Bob ACH Company has supplied you with the API documentation.

The Jim Bob
ACH Interface

The Jim Bob ACH Interface

The API documentation, which has been supplied to you in PDF form, includes the interface definition that the class `JimBobAch` implements. It also includes the definition for the related data classes. You can take the Jim Bob ACH code and for the time being copy it into your system. Enter this code into the directory structure `./com/jimbob/ach` within your source directory. (You would probably be able to paste it directly from the PDF document.)

When you receive the actual API library from Jim Bob and company, ensure that it matches what you have before deleting your temporary copy.

```
// com.jimbob.ach.Ach
package com.jimbob.ach;

public interface Ach {
    public AchResponse issueDebit(AchCredentials credentials, AchTransactionData data);
    public AchResponse markTransactionAsNSF(AchCredentials credentials, AchTransactionData data,
String traceCode);
    public AchResponse refundTransaction(AchCredentials credentials, AchTransactionData data,
String traceCode);
    public AchResponse issueCredit(AchCredentials credentials, AchTransactionData data);
    public AchResponse voidSameDayTransaction(AchCredentials credentials, AchTransactionData
data, String traceCode);
    public AchResponse queryTransactionStatus(AchCredentials credentials, AchTransactionData
data, String traceCode);
}

// com.jimbob.ach.AchCredentials
package com.jimbob.ach;
```

**The Jim Bob
ACH Interface**

```

public class AchCredentials {
    public String merchantId;
    public String userName;
    public String password;
}

// com.jimbob.ach.AchTransactionData
package com.jimbob.ach;

import java.math.BigDecimal;

public class AchTransactionData {
    public String description;
    public BigDecimal amount;
    public String aba;
    public String account;
    public String accountType;
}

// com.jimbob.ach.AchResponse
package com.jimbob.ach;

import java.util.*;

public class AchResponse {
    public Date timestamp;
    public String traceCode;
    public AchStatus status;
    public List<String> errorMessages;
}

// com.jimbob.ach.AchStatus
package com.jimbob.ach;

public enum AchStatus {
    SUCCESS, FAILURE;
}

```

For now, you are interested only in the Ach interface method named `issueDebit`. You must send the `issueDebit` message to a `JimBobAch` object in order to withdraw money from the bank account. The message takes an `AchCredentials` object and an `AchTransactionData` object as parameters. Note the questionable practice of exposing the instance variables directly in these two classes. Unfortunately, when you deal with third-party vendor software, you get what you get.

The `issueDebit` method returns an `AchResponse` object upon completion. This response data indicates whether or not the debit succeeded by using an `AchStatus` enum value.

The Mock Class

You don't have the actual JimBobAch class, but you can create a mock class that implements the same Ach interface as the JimBobAch class. Your test will work with this mock object.

The MockAch class below implements the Ach interface. The only ACH service that concerns you for now is the issueDebit method; you implement the other methods as stubs, returning null to satisfy the compiler. You code the issueDebit method to ensure that the parameters passed to it are as expected. If the parameters are valid,² you construct an AchResponse object, populating it with the data that the test expects to see.

```
package sis.studentinfo;

import java.util.*;
import com.jimbob.ach.*;
import junit.framework.Assert;

class MockAch implements Ach {
    public AchResponse issueDebit(
        AchCredentials credentials, AchTransactionData data) {
        Assert.assertTrue(
            data.account.equals(AccountTest.ACCOUNT_NUMBER));
        Assert.assertTrue(data.aba.equals(AccountTest.ABA));

        AchResponse response = new AchResponse();
        response.timestamp = new Date();
        response.traceCode = "1";
        response.status = AchStatus.SUCCESS;
        return response;
    }

    public AchResponse markTransactionAsNSF(AchCredentials credentials,
        AchTransactionData data,
        String traceCode) {
        return null;
    }
    public AchResponse refundTransaction(AchCredentials credentials,
        AchTransactionData data,
        String traceCode) {
        return null;
    }
    public AchResponse issueCredit(AchCredentials credentials,
        AchTransactionData data) {
        return null;
    }
}
```

The Mock Class

²You should also ensure that the AchCredentials object is as expected; I'm omitting the check here because of space considerations.

```

    }
    public AchResponse voidSameDayTransaction(
        AchCredentials credentials,
        AchTransactionData data,
        String traceCode) {
        return null;
    }
    public AchResponse queryTransactionStatus(AchCredentials credentials,
        AchTransactionData data, String traceCode) {
        return null;
    }
}

```

The Mock Class

Note that this solution creates a two-way dependency, as shown in Figure 12.1. AccountTest class creates MockAch, meaning the test class is dependent upon MockAch. MockAch is dependent upon AccountTest since MockAch refers to the ACCOUNT_NUMBER and ABA class constants from AccountTest. Two-way dependencies are generally not a good thing, but in some cases such a tight coupling is acceptable. The test requires the mock, and the mock is useful only in close concert with the test.

Now back to `testTransferFromBank`. In it, you originally wrote code to instantiate the nonexistent JimBobAch class. Instead, you can now create a MockAch instance and pass that into the Account object:

```

public void testTransferFromBank() {
    // account.setAch(new com.jimbob.ach.JimBobAch());
    account.setAch(new MockAch());

    final BigDecimal amount = new BigDecimal("50.00");
    account.transferFromBank(amount);

    assertEquals(amount, account.getBalance());
}

```

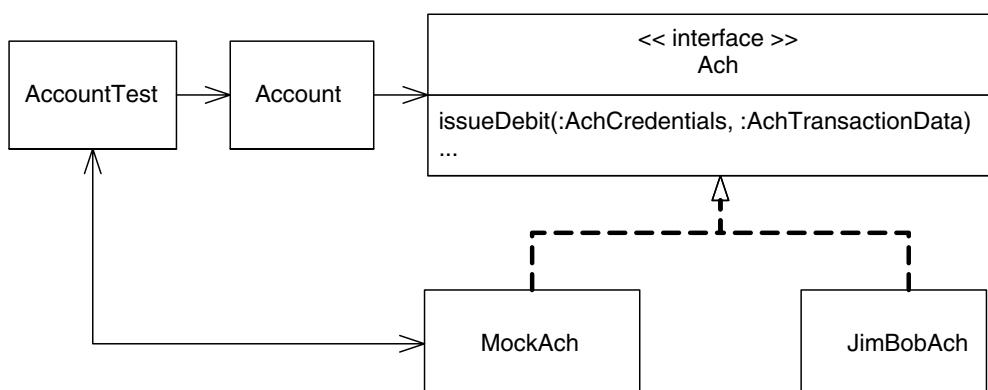


Figure 12.1 Mocking the Ach Interface

The Account Class Implementation

Here is the updated Account class implementation:

```
package sis.studentinfo;

import java.math.BigDecimal;
import com.jimbob.ach.*;

public class Account {
    private BigDecimal balance = new BigDecimal("0.00");
    private int transactionCount = 0;
    private String bankAba;
    private String bankAccountNumber;
    private BankAccountType bankAccountType;
    private Ach ach;

    public enum BankAccountType {
        CHECKING("ck"), SAVINGS("sv");
        private String value;
        private BankAccountType(String value) {
            this.value = value;
        }
        @Override
        public String toString() {
            return value;
        }
    }

    public void credit(BigDecimal amount) {
        balance = balance.add(amount);
        transactionCount++;
    }

    public BigDecimal getBalance() {
        return balance;
    }

    public BigDecimal transactionAverage() {
        return balance.divide(
            new BigDecimal(transactionCount), BigDecimal.ROUND_HALF_UP);
    }

    public void setBankAba(String bankAba) {
        this.bankAba = bankAba;
    }

    public void setBankAccountNumber(String bankAccountNumber) {
        this.bankAccountNumber = bankAccountNumber;
    }
}
```

The Account
Class
Implementation

The Account
Class
Implementation

```

public void setBankAccountType(
    Account.BankAccountType bankAccountType) {
    this.bankAccountType = bankAccountType;
}

public void transferFromBank(BigDecimal amount) {
    AchCredentials credentials = createCredentials();

    AchTransactionData data = createData(amount);

    Ach ach = getAch();
    AchResponse achResponse = ach.issueDebit(credentials, data);

    credit(amount);
}

private AchCredentials createCredentials() {
    AchCredentials credentials = new AchCredentials();
    credentials.merchantId = "12355";
    credentials.userName = "sismerc1920";
    credentials.password = "pitseleh411";
    return credentials;
}

private AchTransactionData createData(BigDecimal amount) {
    AchTransactionData data = new AchTransactionData();
    data.description = "transfer from bank";
    data.amount = amount;
    data.aba = bankAba;
    data.account = bankAccountNumber;
    data.accountType = bankAccountType.toString();
    return data;
}

private Ach getAch() {
    return ach;
}

void setAch(Ach ach) {
    this.ach = ach;
}
}

```

Account allows clients to pass it an Ach reference that it then stores. In the production system, your client code that constructs an Account will pass the Account a JimBobAch object. In the test, your code passes a MockAch object to the Account. In either case, Account doesn't know and doesn't care which concrete type it is sending the issueDebit message to.

While testTransferFromBank now passes, it is difficult to see exactly why. The mock class is in another source file, so to figure out what's going on, you must flip between the two class definitions. This isn't totally unacceptable,

but you can improve upon things. One solution would be to include the mock definition as a nested class. Another would be to set up the mock directly within the test method, using an anonymous inner class.

Anonymous Inner Classes

Lesson 11 discussed the distinction between static nested classes and inner classes. A third type of nested class is the *anonymous inner class*. An anonymous inner class allows you to dynamically define unnamed class implementations within a method body.

Understanding the syntax and nuances of anonymous inner classes can be daunting. But once you understand anonymous inner classes, your use of them can make your code more concise and easy to understand.

Take the body of the MockAch class (i.e., all the method definitions) and insert it as the first line of code in `testTransferFromBank`. Immediately preceding all that code, add the line:

```
Ach mockAch = new Ach() {
```

Then terminate the mock Ach method definitions with a closing brace and semicolon. The transformed test:³

```
public void testTransferFromBank() {
    Ach mockAch = new Ach() {
        public AchResponse issueDebit(
            AchCredentials credentials, AchTransactionData data) {
            Assert.assertTrue(
                data.account.equals(AccountTest.ACCOUNT_NUMBER));
            Assert.assertTrue(data.aba.equals(AccountTest.ABA));

            AchResponse response = new AchResponse();
            response.timestamp = new Date();
            response.traceCode = "1";
            response.status = AchStatus.SUCCESS;
            return response;
        }
        public AchResponse markTransactionAsNSF(
            AchCredentials credentials,
            AchTransactionData data,
            String traceCode) {
            return null;
        }
        public AchResponse refundTransaction(AchCredentials credentials,
            AchTransactionData data,
```

**Anonymous
Inner Classes**

³You'll need to import com.jimbob.ach and java.util to get this to compile.

Instance Initializers

In Lesson 4 you learned about static initialization blocks. Java allows you to use instance initialization blocks, which are also known as instance initializers.

Within an anonymous inner class, you cannot create a constructor. Since the anonymous inner class has no name, you wouldn't be able to supply a name for its constructor! Instead, you can use an instance initialization block.

```
Expirable t = new Expirable() {
    private long then;
    {
        long now = System.currentTimeMillis();
        then = now + 86400000;
    }

    public boolean isExpired(Date date) {
        return date.getTime() > then;
    }
};
```

Instance initializers are rarely used within top-level classes. Prefer either constructor or field-level initialization. However, you might use instance initializers to eliminate duplication if you have multiple constructors that must execute common setup code.

```
String traceCode) {
    return null;
}
public AchResponse issueCredit(AchCredentials credentials,
    AchTransactionData data) {
    return null;
}
public AchResponse voidSameDayTransaction(
    AchCredentials credentials,
    AchTransactionData data,
    String traceCode) {
    return null;
}
public AchResponse queryTransactionStatus (
    AchCredentials credentials,
    AchTransactionData data, String traceCode) {
    return null;
}
};

account.setAch(mockAch);

final BigDecimal amount = new BigDecimal("50.00");
account.transferFromBank(amount);

assertEquals(amount, account.getBalance());
}
```

The line of code:

```
Ach mockAch = new Ach() {
```

creates a reference named `mockAch` of the interface type `Ach`. The right-hand side instantiates an `Ach` object using `new`. But `Ach` is an interface—how can you instantiate an interface?

Java is allowing you to dynamically provide an implementation for the `Ach` interface and at the same time create an instance with that implementation. You provide the implementation in a block of code immediately after the constructor call (`new Ach()`) but before the semicolon that terminates the statement.

The implementation supplied in the block is of `Ach` type, but it has no specific class name. It is *anonymous*. You can send messages to this object, store it, or pass it as a parameter, just as you might with an object of a named `Ach` implementation such as `MockAch`.

Make sure that your tests pass, but don't delete `MockAch` just yet.

As you've seen, anonymous inner classes can be useful when building mocks for testing purposes. Swing applications (see Additional Lesson I) and multithreading (see Lesson 13) often make heavy use of anonymous inner classes.

Admittedly, it's still a bit difficult to follow what's going on in `testTransferFromBank`. The test method is now long and cluttered. The next section on adapters shows how to improve on things.

Adapters

Adapters

Overusing anonymous inner classes can actually make code harder to follow. Your goal should be to keep methods short, even test methods. The long anonymous inner class implementation in `testTransferFromBank` means that you will probably need to scroll in order to get the big picture of what the method is doing.

You can create an *interface adapter class* that provides an empty implementation for the `Ach` interface. Modify the `MockAch` class to supply an empty definition for the `issueDebit` method.

```
package sis.studentinfo;  
  
import java.util.*;  
import com.jimbob.ach.*;  
import junit.framework.Assert;
```

Adapters

```

class MockAch implements Ach {
    public AchResponse issueDebit(
        AchCredentials credentials, AchTransactionData data) {
        return null;
    }
    public AchResponse markTransactionAsNSF(AchCredentials credentials,
        AchTransactionData data,
        String traceCode) {
        return null;
    }
    public AchResponse refundTransaction(AchCredentials credentials,
        AchTransactionData data,
        String traceCode) {
        return null;
    }
    public AchResponse issueCredit(AchCredentials credentials,
        AchTransactionData data) {
        return null;
    }
    public AchResponse voidSameDayTransaction(
        AchCredentials credentials,
        AchTransactionData data,
        String traceCode) {
        return null;
    }
    public AchResponse queryTransactionStatus(AchCredentials credentials,
        AchTransactionData data, String traceCode) {
        return null;
    }
}

```

Back in AccountTest, change the instantiation from new Ach() to new MockAch(). Remove the method implementations for everything but issueDebit.

```

public void testTransferFromBank() {
    Ach mockAch = new MockAch() {
        public AchResponse issueDebit(
            AchCredentials credentials, AchTransactionData data) {
            Assert.assertTrue(
                data.account.equals(AccountTest.ACCOUNT_NUMBER));
            Assert.assertTrue(data.aba.equals(AccountTest.ABA));

            AchResponse response = new AchResponse();
            response.timestamp = new Date();
            response.traceCode = "1";
            response.status = AchStatus.SUCCESS;
            return response;
        }
    };
    account.setAch(mockAch);
}

```

```

final BigDecimal amount = new BigDecimal("50.00");
account.transferFromBank(amount);

assertEquals(amount, account.getBalance());
}

```

You are still creating an anonymous inner class. An anonymous inner class can either implement an interface or subclass another class. This code demonstrates the latter. Here, you use the MockAch class as an adapter to hide the methods that your mock doesn't need to worry about. Now you can include the relevant mock code in a reasonably brief test method.

**Accessing
Variables from
the Enclosing
Class**

Accessing Variables from the Enclosing Class

 You need an additional test, `testFailedTransferFromBank`, to define what happens when the bank rejects the debit request. The test and mock are going to be very similar to `testTransferFromBank`. The student's balance should not change on a failed transfer, so the test assertion must change. The status returned from the mock in the `AchResponseData` object must be `AchStatus.FAILURE`.

Because the mocks will be almost exactly the same, you want a way to create both of them without introducing duplicate code. You want to be able to phrase the new test as follows:⁴

```

public void testFailedTransferFromBank() {
    account.setAch(createMockAch(AchStatus.FAILURE));
    final BigDecimal amount = new BigDecimal("50.00");
    account.transferFromBank(amount);
    assertEquals(new BigDecimal("0.00"), account.getBalance());
}

```

The existing test, `testTransferFromBank`, can use the same construct.

```

public void testTransferFromBank() {
    account.setAch(createMockAch(AchStatus.SUCCESS));
    final BigDecimal amount = new BigDecimal("50.00");
    account.transferFromBank(amount);
    assertEquals(amount, account.getBalance());
}

```

Anonymous inner class objects are just objects. They can be created in a separate method and returned like any other object.

⁴You may also want the Account code to throw an exception if the ACH debit fails. I avoided the exception here for simplicity reasons.

```
// THIS WON'T COMPILE!
private Ach createMockAch(AchStatus status) {
    return new MockAch() {
        public AchResponse issueDebit(
            AchCredentials credentials, AchTransactionData data) {
            Assert.assertTrue(
                data.account.equals(AccountTest.ACCOUNT_NUMBER));
            Assert.assertTrue(data.aba.equals(AccountTest.ABA));

            AchResponse response = new AchResponse();
            response.timestamp = new Date();
            response.traceCode = "1";
            response.status = status;
            return response;
        }
    };
}
```

Accessing Variables from the Enclosing Class

Class Names for Anonymous Inner Classes

When it compiles a class containing an anonymous inner class, Java uses a similar rule for naming the compilation unit as its rule for anonymous nested classes that are not anonymous. Java uses the dollar sign (\$) to separate the class names.

A slight problem: An anonymous inner class has no name! Java provides a simple solution, numbering anonymous inner classes using a counter starting at 1. It then uses this number as the nested inner class name portion of the compilation unit name.

For example, the AccountTest class creates two compilation units: AccountTest.class and AccountTest\$1.class.

I'll repeat my warning from last chapter: When you copy class files or put them in a JAR file for distribution, don't forget to include all the nested classes.

However, this code won't compile. You will see the message:

```
local variable status is accessed from within inner class; needs to be declared final
```

Anonymous inner classes are inner classes. By the definition you learned in Lesson 11, an inner classes has access to the instance variables (and methods) of the enclosing class.

An anonymous inner class does *not* have access to local variables. The status parameter is analogous to a local variable because it is accessible only to code in the createMockAch method.

According to the compiler error, however, you can get around this limitation by declaring the status parameter as final.

```
private Ach createMockAch(final AchStatus status)
```

After you do this, your code should compile but your tests will not pass.

Why must you define the variable as `final`? Remember that by declaring a variable as `final`, you are declaring that its value cannot be changed after it has been set.

An instance of an anonymous inner class can exist outside of the method in which it is declared. In our example, the `createMockAch` method instantiates a new anonymous inner class and returns it from the method it will be used by the calling test methods.

Local variables do not exist once a method has finished executing! Nor do parameters—all parameters to methods are copies of the arguments passed by the calling code.⁵ If an anonymous inner class was able to access local variables, the value of the local variable could be destroyed by the time the code in the anonymous inner class actually executed. This would, of course, be a bad thing.

To get your tests to pass, insert the code in `Account` so that you credit only successful transfers.

```
public void transferFromBank(BigDecimal amount) {
    AchCredentials credentials = createCredentials();
    AchTransactionData data = createData(amount);
    Ach ach = getAch();
    AchResponse achResponse = ach.issueDebit(credentials, data);
    if (achResponse.status == AchStatus.SUCCESS)
        credit(amount);
}
```

Tradeoffs

With a bit of refactoring, this method simplifies to:

```
public void transferFromBank(BigDecimal amount) {
    AchResponse achResponse =
        getAch().issueDebit(createCredentials(), createData(amount));
    if (achResponse.status == AchStatus.SUCCESS)
        credit(amount);
}
```

Tradeoffs

Earlier, I mentioned that it is a good thing to be able to see the mock behavior right in the test itself. Then I had you extract the mock definition to a separate method in order to eliminate the duplication that was impending with the addition of another test method that needed a similar mock. This was at a cost of some expressiveness, although it's reasonably clear what the code

⁵For a further discussion, see Additional Lesson III, which talks about call by value.

```
account.setAch(createMockAch(AchStatus.SUCCESS));
```

is doing.

Which is worse? Duplication or code that is hard to follow? The answer isn't always clear cut. The question often engenders debate. From my perspective, duplicate code is usually more troublesome than difficult-to-read code. You don't often need to choose one over the other, but when you do, eliminating duplication is a safer route. By eliminating duplicate code, you reduce the cost of maintenance efforts—you avoid having to make changes in more than one place. You also minimize the risk of making a change in one place but forgetting to make it in another place.

Reflection

Reflection

You've seen how the use of the `Object` method `getClass` allows you to determine the type of an object at runtime. This is known as reflective capability—the ability of code to examine information about the code itself; that is, to *reflect* on itself. Another term used for reflection capabilities is *metadata*.

JUnit uses reflection heavily to gather the information it needs to be able to run your tests. You supply a test class file to JUnit to be tested. JUnit uses reflection to determine whether or not your class file inherits from `junit.framework.TestCase`, rejecting it if it does not. JUnit also uses reflection to obtain a list of methods in a test class. It then iterates these methods, executing only those that pass its criteria for being a test method:

- The method name must start with the word “test,” in lowercase letters.
- The return type of the method must be `void`.
- The method must not take any parameters.

So far, you have been telling JUnit which classes you want tested. This allows you to run only the tests that you specify. A problem with this approach is that it is easy to forget to add your test class to a suite. You might end up skipping critical tests!

Another approach is to have Java scan through all the classes in your classpath, gather the classes that are legitimate test classes, and execute only those. The benefit is that you don't have to maintain the suites in code. No tests will be left behind. The only downside is that there may be tests, such as performance tests, that you don't want run all the time. However, you can write additional code to bypass such tests if necessary.

Using JUnit Code

Fortunately, JUnit supplies a mechanism that will help you gather the test classes on the classpath. How would you know that? First, you might have run JUnit and observed that it has the capability to gather a list of all test classes. Following up on that, you could have perused the source distributed with JUnit and seen that it has a class named `junit.runner.ClassPathTestCollector`.

You will create a class named `SuiteBuilder` to gather all test classes. A simple first test for `SuiteBuilder` would ensure that `SuiteBuilderTest` itself was returned in the list of test classes found on the classpath.

```
package sis.testing;

import junit.framework.*;
import java.util.*;

public class SuiteBuilderTest extends TestCase {
    public void testGatherTestClassNames() {
        SuiteBuilder builder = new SuiteBuilder();
        List<String> classes = builder.gatherTestClassNames();
        assertTrue(classes.contains("testing.SuiteBuilderTest"));
    }
}
```

**Using JUnit
Code**

The interesting part of the test is that it binds the list to the `String` type—not the `Class` type. When you look at the source for `ClassPathTestCollector`, you will note that it returns an `Enumeration` of `Strings`, each representing the class name. We'll go along with that for now and see if it causes any problems later.

```
package sis.testing;

import java.util.*;
import junit.runner.*;
import junit.framework.*;

public class SuiteBuilder {
    public List<String> gatherTestClassNames() {
        TestCollector collector = new ClassPathTestCollector() {
            public boolean isTestClass(String className) {
                return super.isTestClass(className);
            }
        };
        return Collections.list(collector.collectTests());
    }
}
```

The ClassPathTestCollector is declared as `abstract`, so you *must* extend it in order to instantiate it. Interestingly, there are no abstract methods contained within, meaning you are not required to override any of ClassPathTestCollector's methods. You *will* want to override the method `isTestClass`:

```
protected boolean isTestClass(String classFileName) {  
    return  
        classFileName.endsWith(".class") &&  
        classFileName.indexOf('$') < 0 &&  
        classFileName.indexOf("Test") > 0;  
}
```

The Class Class

The query method `isTestClass` concerns itself only with the class filenames, not with the structure of the classes themselves. The conditional states that if a filename ends in a `.class` extension, does not contain a `$` (i.e., it does not represent a nested class), and contains the text “Test,” then it is a test class.

You might recognize that this definition could cause problems: What if a file with a `.class` extension is not a valid class file? What if the class name contains the text “Test” but is not a test class? The last condition is highly likely. Your own code for the student information system contains the class `util.TestUtil`, which is not a `TestCase` subclass.

You’re not going to worry about these problems (yet) since your test doesn’t (yet) concern itself with any of these cases. For now, your implementation code in `getTestClassNames` overrides `isTestClass` and simply calls the superclass method to use its existing behavior. This is unnecessary, but it acts as a placeholder and reminder when you are ready to take care of the other concerns.

The last line of `gatherTestClassNames` sends the message `collectTests` to the ClassPathTestCollector. Somewhere along the way, `collectTests` indirectly calls `isTestClass`. The return type of `collectTests` is `java.util.Enumeration`. You can use the `java.util.Collections` utility method `list` to transform the `Enumeration` into an `ArrayList`. You will get an unchecked warning, since ClassPathTestCollector uses raw collections—collections not bound to a specific type.

The Class Class

Let’s tackle the likely concern that a class with the word “Test” in its name is not a `junit.framework.TestCase` subclass. A test for this case would need to specify a class that the `SuiteBuilder` can attempt to gather and then reject. You might consider using an existing class that you have already coded for the student information system, such as `TestUtil`.

However, you do not want to depend on the existence or stability of existing classes. Someone making a valid change to `TestUtil`, or deleting it, would unwittingly break `SuiteBuilderTest` tests. You instead will create dummy test classes expressly for the use of `SuiteBuilderTest`. Start by creating the class `NotATestClass` in the new package `sis.testing.testclasses`.

```
package sis.testing.testclasses;
public class NotATestClass {}
```

`NotATestClass` does not extend `junit.framework.TestCase`, so it should not be recognized as a test class. Add an assertion to your test method.

```
public void testGatherTestClassNames() {
    SuiteBuilder builder = new SuiteBuilder();
    List<String> classes = builder.gatherTestClassNames();
    assertTrue(classes.contains("testing.SuiteBuilderTest"));
    assertFalse(classes.contains("testing.testclasses.NotATestClass"));
}
```

The Class Class

The relevant code in `SuiteBuilder`:⁶

```
public List<String> gatherTestClassNames() {
    TestCollector collector = new ClassPathTestCollector() {
        public boolean isTestClass(String className) {
            if (!super.isTestClass(className))
                return false;
            String className = classNameFromFile(className); // 1
            Class klass = createClass(className); // 2
            return TestCase.class.isAssignableFrom(klass); // 3
        }
    };
    return Collections.list(collector.collectTests());
}

private Class createClass(String name) {
    try {
        return Class.forName(name);
    }
    catch (ClassNotFoundException e) {
        return null;
    }
}
```

The additional constraint implemented in `SuiteBuilder` is that a class filename must represent a compilation unit that extends from `TestCase`. There

⁶Note use of the variable name `klass` to represent a `Class` object. You can't use `class`, since Java reserves it for defining classes only. Some developers use the single character `c` (not recommended), others `clazz`.

are three steps involved. The following descriptions correspond to the commented lines of code in `gatherTestclassNames`.

1. Translate the directory-based filename into a class name, for example from “testing/testclasses/NotATestClass” to “testing.testclasses.NotATestClass.” The `ClassPathTestCollector` method `classNameFromFile` accomplishes this task.
2. Create a `Class` object from the class name. In `createClass`, the static method call of `forName`, defined on `Class`, does this. The `forName` method throws a `ClassNotFoundException` if the `String` object you pass to it does not represent a class that can be loaded by Java. For now, you can return `null` from `createClass` if this occurs, but you will need to write a test to cover this possibility.
3. Determine whether the class is a subclass of `TestCase`. To do so, you will use a method on the class `java.lang.Class`, which provides metadata about a class definition. The `Class` method `isAssignableFrom` takes as a parameter a `Class` object and returns `true` if it is possible to assign an instance of the parameter (`klass`, in the above code) to a reference of the receiving class type (`junit.framework.TestCase`).

Take a brief look at the Java API documentation for `java.lang.Class`. It contains methods such as `getMethods`, `getConstructors`, and `getFields` that you can use to derive most of the information you need about a compiled Java class.

Building the Suite

Building the Suite

The next step is to construct a `TestSuite` object to be passed to the Swing test runner. The code to do this is simple: iterate through the results of `gatherTestclassNames`, create the corresponding `Class` object for each test class name, and add it to the suite. The test seems to be straightforward: Ensure that the suite contains the expected test classes.

```
public void testCreateSuite() {
    SuiteBuilder builder = new SuiteBuilder() {
        public List<String> gatherTestclassNames() {
            List<String> classNames = new ArrayList<String>();
            classNames.add("testing.SuiteBuilderTest");
            return classNames;
        }
    };

    TestSuite suite = builder.suite();
    assertEquals(1, suite.testCount());
    assertTrue(contains(suite, testing.SuiteBuilderTest.class));
}
```

I used a mock instead of letting the suite method gather all classes normally. Otherwise, `gatherTestclassNames` would return the entire list of classes on the classpath. This list would include all other tests in the student information system. You would have no definitive way to prove that the suite method was doing its job.

Determining whether or not a class is included in a `TestSuite` is not as trivial as you might expect. A test suite can either contain test case classes or other test suites. Figure 12.2 shows this design, known as a composite.⁷ The filled-in diamond represents a *composition* relationship between `TestSuite` and `Test`. A `TestSuite` is composed of `Test` objects.

To determine if a class is contained somewhere within a suite, you must traverse the entire hierarchy of suites. The `contains` method here uses recursion, since `contains` calls itself when encountering a suite within the current suite.

```
public boolean contains(TestSuite suite, Class testClass) {
    List testClasses = Collections.list(suite.tests());
    for (Object object: testClasses) {
        if (object.getClass() == TestSuite.class)
            if (contains((TestSuite)object, testClass))
                return true;
        if (object.getClass() == testClass)
            return true;
    }
    return false;
}
```

Building the Suite

All that is left is for you to create a `TestRunner` class. `TestRunner` will use the `SuiteBuilder` to construct a suite. It will then execute the Swing test runner using this suite. There are no tests—you'll exercise the code as part of

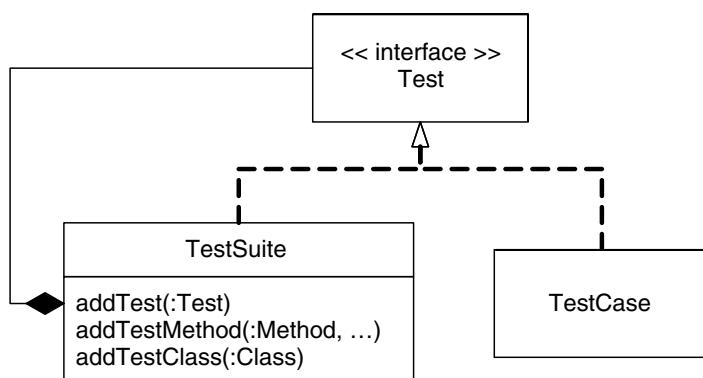


Figure 12.2 JUnit Composite Design

⁷[Gamma1995].

continually running JUnit. Writing tests for this code would be possible but difficult. The way you constructed SuiteBuilder, however, means that the TestRunner code is so small, it's almost breakproof.

```
package sis.testing;

public class TestRunner {
    public static void main(String[] args) {
        new junit.swingui.TestRunner().run(TestRunner.class);
    }

    public static junit.framework.Test suite() {
        return new SuiteBuilder().suite();
    }
}
```

Class Modifiers

You'll want to add a target to your build script that executes testing.TestRunner:

```
<target name="runAllTests" depends="build">
    <java classname="testing.TestRunner" fork="yes">
        <classpath refid="classpath" />
    </java>
</target>
```

When you run all the tests, however, you get about half a dozen failures.

Class Modifiers

The problem is that SessionTest, an abstract class, contains a number of definitions for test methods. The methods are intended to be executed as part of SessionTest subclasses; thus, you did not previously add SessionTest to a suite. You need to modify SuiteBuilder to ignore abstract classes.

Create a test class that is abstract and extends from TestCase.

```
package sis.testing.testclasses;

abstract public class AbstractTestClass
    extends junit.framework.TestCase {
    public void testMethod() {}
}
```

Then modify the test to ensure that it isn't collected.

```
public void testGatherTestClassNames() {
    SuiteBuilder builder = new SuiteBuilder();
    List<String> classes = builder.gatherTestClassNames();
    assertTrue(classes.contains("testing.SuiteBuilderTest"));
```

```

assertFalse(classes.contains("testing.testclasses.NotATestClass"));
assertFalse(
    classes.contains("testing.testclasses.AbstractTestClass"));
}

```

Run the test and ensure that it fails. Only then should you go on to correct the code in SuiteBuilder. You are following a testing pattern: If you release code that fails either in production or in an acceptance test (a customer-defined test that exercises code from an end user's standpoint), it means that it was inadequately unit-tested. Your job is to add the missing test code and ensure that it fails for the same reason that the acceptance test failed⁸.



Never depend solely on one level of testing. Ensure you have a layer above unit testing to ferret out “holes” in your unit tests.

Now that you've seen the test fail, you can add code to SuiteBuilder that corrects the defect:

```

public List<String> gatherTestclassNames() {
    TestCollector collector = new ClassPathTestCollector() {
        public boolean isTestClass(String classFileName) {
            if (!super.isTestClass(classFileName))
                return false;
            String className = classNameFromFile(classFileName);
            Class klass = createClass(className);
            return
                TestCase.class.isAssignableFrom(klass) &&
                isConcrete(klass);
        }
    };
    return Collections.list(collector.collectTests());
}

private boolean isConcrete(Class klass) {
    if (klass.isInterface())
        return false;
    int modifiers = klass.getModifiers();
    return !Modifier.isAbstract(modifiers);
}

```

Class Modifiers

If a type (Class represents both interface and class types) is an interface, it is not concrete. You can determine that using the Class method `isInterface`.

The `isConcrete` method also uses the Class method `getModifiers`. The method `getModifiers` returns an `int` that embeds a list of flags, each corresponding to a possible class modifier.⁹ Some of the modifiers you have seen include `abstract`, `static`, `private`, `protected`, and `public`. A utility class `java.lang.reflect.Modifier`

⁸[Jeffries2001], p. 163.

⁹Methods and fields also have modifiers and use the same int-flag construct.

(you'll need to add an appropriate `import` statement to `SuiteBuilder`) contains a number of static query methods to help you determine the modifiers that are set within the modifiers `int`.

You can now delete all of your `AllTests` classes!

Dynamic Proxy

J2SE version 1.3 introduced a new class, `java.lang.reflect.Proxy`, that allows you to construct *dynamic proxy classes*. A dynamic proxy class allows you to dynamically—at runtime—implement one or more interfaces.

The *proxy pattern* is one of the more useful patterns identified in the book *Design Patterns*.¹⁰ A proxy is a stand-in; a proxy object stands in for a real object. In a proxy pattern implementation, client objects think they are interacting with the real object but are actually interacting with a proxy.

Figure 12.3 shows the proxy pattern in the context of distributed object communication. A `ServiceImplementation` class resides on a separate machine in a separate process space from the `Client` class. In order for a `Client` to interact with a `ServiceImplementation`, some low-level communications must take place. Suppose `Client` must send the message `submitOrder` to a `ServiceImplementation`. This Java message send must be translated to a data stream that can be transmitted over the wire. On the server side, the data

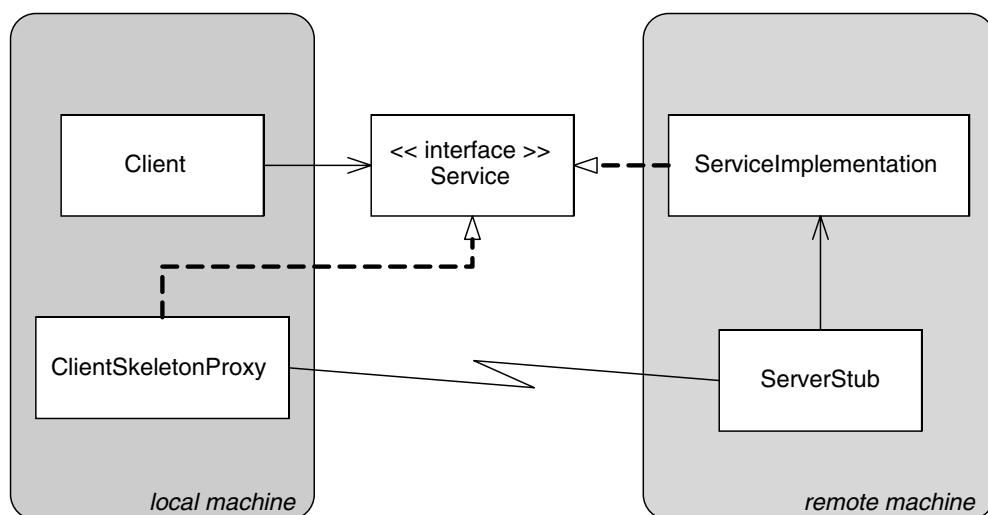


Figure 12.3 *Distributed Communication Using a Proxy*

¹⁰[Gamma1995].

stream must be reconstituted into a Java message to be sent to the ServiceImplementation object.

You want the code in Client to be able to work with the code in ServiceImplementation as if it were executing in the same local Java VM process. Neither the code in Client nor the code in ServiceImplementation should have to be concerned with the nuts and bolts of the remote communication.

The solution is to have the ServiceImplementation class implement a Service interface that a client-side class, ClientSkeletonProxy, also implements. A Client object can then interact with a ClientSkeletonProxy through this interface, thinking that it is interacting with the real ServiceImplementation. Thus, the ClientSkeletonProxy is a stand-in, or proxy, for the real ServiceImplementation. The ClientSkeletonProxy takes incoming requests and collaborates with a ServerStub on the remote machine to do the low-level communication. The ServerStub takes incoming data transmissions and translates them into calls against the ServiceImplementation. (You might recognize that implementing this solution in Java is heavily dependent upon the use of reflection.)

In Java, this proxy scheme is the basis of a technology known as Remote Method Invocation (RMI). RMI, in turn, is the basis of the Enterprise Java Beans (EJB) technology for component-based distributed computing.¹¹

Proxies have many other uses, including techniques known as lazy load, copy on write, pooling, caching, and transactional markup. You will code a transparent security mechanism using a proxy.

In a Java proxy implementation, the proxy implements the same interface as the real object. The proxy intercepts messages sent to a client; after processing a message, the proxy often delegates it to the real, or *target*, object. You can imagine that maintaining proxies can become tedious if a proxy class must implement the same interface as the target class. For every method you add to an interface, you must provide both the real implementation and the proxy implementation. Fortunately, the dynamic proxy in Java eliminates the need for you to explicitly implement every interface method within the proxy class.

A Secure
Account Class

A Secure Account Class



You need to be able to restrict access to certain Account methods based on the permissions that the client has. A client should have either read-only or update access. A client with update access can use

¹¹EJB is a part of the J2EE (Java 2 Enterprise Edition) platform.

any method defined on Account. A client with read-only access can use only methods that don't change an account's state.

You could add code to the Account class, passing in a user's classification when the account is created. To each restricted method, you would add code that would check the user classification and throw an exception if the user did not have the proper access rights. Security-related code would quickly clutter your Account class, obscuring its business logic. You would be violating the Single-Responsibility Principle!¹²

Instead, you will externalize the security restrictions into a proxy class. The Account class itself will remain almost completely untouched! We are going for the open-closed principle¹³ here—building new functionality by adding new code, not by modifying existing code.

The use of proxies often calls for factories. The UML diagram in Figure 12.4 shows the complete solution for the security proxy, including the use of the class AccountFactory to return an instance of a SecureProxy. The client can think that it is interacting with a real Account, but it is instead interacting with a proxy that is able to respond to the same messages.

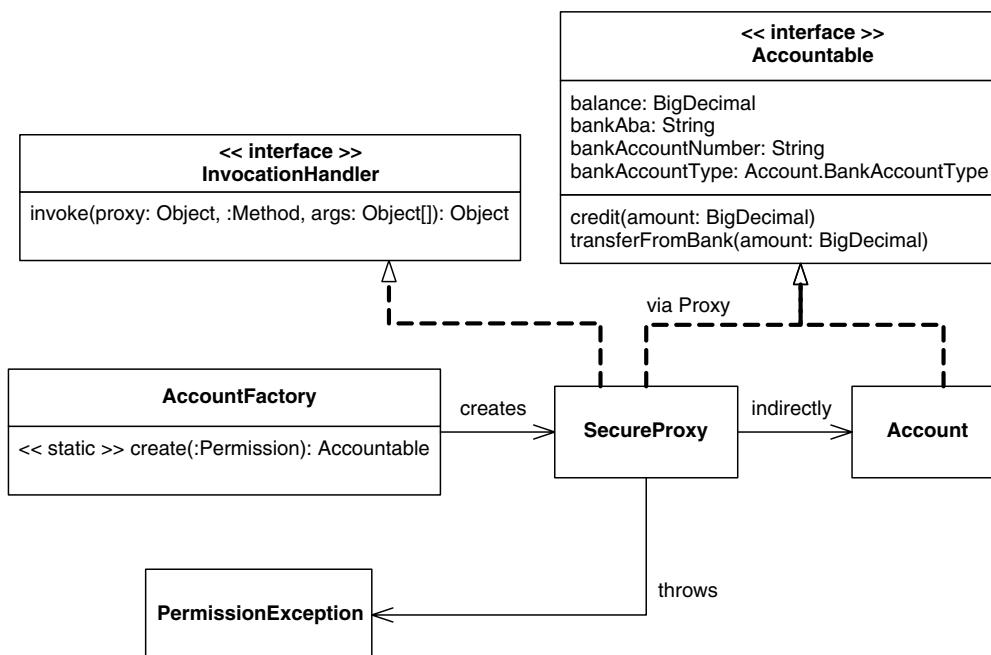


Figure 12.4 Security Proxy

¹²[Martin2003], p. 95.

¹³[Martin2003], p. 99.

The AccountFactory class uses the dynamic proxy class Proxy to create the SecureProxy object. SecureProxy does not directly implement Accountable; instead, the Proxy class sets SecureProxy up to capture all incoming messages and redirect each to the InvocationHandler interface method invoke.

Building the Secure Account Solution

The starting point is AccountFactoryTest, a test class that exercises the various combinations of permissions and methods. The test class contains two test methods: testReadOnlyAccess ensures that all account methods can be called normally by a user with update permission, and testUpdateAccess ensures that exceptions are thrown when a user with read-only access attempts to execute a secure method.

**Building the
Secure Account
Solution**

```
package sis.studentinfo;

import java.math.*;
import java.util.*;
import java.lang.reflect.*;
import junit.framework.*;

import sis.security.*;

public class AccountFactoryTest extends TestCase {
    private List<Method> updateMethods;
    private List<Method> readOnlyMethods;

    protected void setUp() throws Exception {
        updateMethods = new ArrayList<Method>();
        addUpdateMethod("setBankAba", String.class);
        addUpdateMethod("setBankAccountNumber", String.class);
        addUpdateMethod("setBankAccountType",
                        Account.BankAccountType.class);
        addUpdateMethod("transferFromBank", BigDecimal.class);
        addUpdateMethod("credit", BigDecimal.class);

        readOnlyMethods = new ArrayList<Method>();
        addReadOnlyMethod("getBalance");
        addReadOnlyMethod("transactionAverage");
    }

    private void addUpdateMethod(String name, Class parmClass)
        throws Exception {
        updateMethods.add(
            Accountable.class.getDeclaredMethod(name, parmClass));
    }
}
```

**Building the
Secure Account
Solution**

```

private void addReadOnlyMethod(String name) throws Exception {
    Class[] noParms = new Class[] {};
    readOnlyMethods.add(
        Accountable.class.getDeclaredMethod(name, noParms));
}

public void testUpdateAccess() throws Exception {
    Accountable account = AccountFactory.create(Permission.UPDATE);
    for (Method method: readOnlyMethods)
        verifyNoException(method, account);
    for (Method method: updateMethods)
        verifyNoException(method, account);
}

public void testReadOnlyAccess() throws Exception {
    Accountable account = AccountFactory.create(Permission.READ_ONLY);
    for (Method method: updateMethods)
        verifyException(PermissionException.class, method, account);
    for (Method method: readOnlyMethods)
        verifyNoException(method, account);
}

private void verifyException(
    Class exceptionType, Method method, Object object)
    throws Exception {
    try {
        method.invoke(object, nullParmsFor(method));
        fail("expected exception");
    }
    catch (InvocationTargetException e) {
        assertEquals("expected exception",
            exceptionType, e.getCause().getClass());
    }
}

private void verifyNoException(Method method, Object object)
    throws Exception {
    try {
        method.invoke(object, nullParmsFor(method));
    }
    catch (InvocationTargetException e) {
        assertFalse(
            "unexpected permission exception",
            PermissionException.class == e.getCause().getClass());
    }
}

private Object[] nullParmsFor(Method method) {
    return new Object[method.getParameterTypes().length];
}
}

```

The test demonstrates some additional features of the Java reflection API.

In the test, you create two collections, one for read-only methods and one for update methods. You populate both read-only and update collections in the `setUp` method with `java.lang.reflect.Method` objects. Method objects represent the individual methods defined within a Java class and contain information including the name, the parameters, the return type, and the modifiers (e.g., static and final) of the method. Most important, once you have a `Method` object and an object of the class on which the method is defined, you can dynamically execute that method.

You obtain `Method` objects by sending various messages to a `Class` object. One way is to get an array of all methods by sending `getDeclaredMethods` to a `Class`. The class will return all methods directly *declared* within the class. You can also attempt to retrieve a specific method by sending the `Class` the message `getDeclaredMethod`, passing along with it the name and the list of parameter types. In the above code, the line:

```
Accountable.class.getDeclaredMethod(name, parmClass)
```

**Building the
Secure Account
Solution**

demonstrates use of `getDeclaredMethod`.

The test `testReadOnlyAccess` first uses the `AccountFactory` to create an object—the proxy—that implements the `Accountable` interface. The parameter to the `create` method is a `Permission` enum:

```
package sis.security;

public enum Permission {
    UPDATE, READ_ONLY
}
```

You derive the `Accountable` interface by extracting all public method signatures from the `Account` class:

```
package sis.studentinfo;

import java.math.*;

public interface Accountable {
    public void credit(BigDecimal amount);
    public BigDecimal getBalance();
    public BigDecimal transactionAverage();
    public void setBankAba(String bankAba);
    public void setBankAccountNumber(String bankAccountNumber);
    public void setBankAccountType(
        Account.BankAccountType bankAccountType);
    public void transferFromBank(BigDecimal amount);
}
```

You'll need to modify the `Account` class definition to declare that it implements this interface:

```
public class Account implements Accountable {
```

Which it already does—no further changes to Account are required.

Once testReadOnlyAccess has an Accountable reference, it loops through the lists of update methods and read-only methods.

```
for (Method method: updateMethods)
    verifyException(PermissionException.class, method, account);
for (Method method: readOnlyMethods)
    verifyNoException(method, account);
```

For each update method, testReadOnlyAccess calls verifyException to ensure that an exception is thrown when method is invoked on account. The test also ensures that each read-only method can be called without generating security exceptions.

The method verifyException is responsible for invoking a method and ensuring that a SecurityException was thrown. The line of code that actually calls the method:

```
method.invoke(object, nullParmsFor(method));
```

To execute a method, you send a Method object the invoke message, passing along with it the object on which to invoke the method plus an Object array of parameters. Here, you use the utility method nullParmsFor to construct an array of all null values. It doesn't matter what you pass the methods. Even if a method generates a NullPointerException or some other sort of exception, the test concerns itself only with PermissionException:

```
package sis.security;

public class PermissionException extends RuntimeException {
```

The method verifyException expects the invoke message send to generate an InvocationTargetException, not a PermissionException. If the underlying method called by invoke throws an exception, invoke wraps it in an InvocationTargetException. You must call getCause on the InvocationTargetException to extract the original wrapped exception object.

The use of reflection in the test was not necessary. If this was not a lesson on reflection, I might have had you implement the test in some other nondynamic manner.

The job of the AccountFactory class is to create instances of classes that implement the Accountable interface.

```
package sis.studentinfo;

import java.lang.reflect.*;
```

```

import sis.security.*;

public class AccountFactory {
    public static Accountable create(Permission permission) {
        switch (permission) {
            case UPDATE:
                return new Account();
            case READ_ONLY:
                return createSecuredAccount();
        }
        return null;
    }

    private static Accountable createSecuredAccount() {
        SecureProxy secureAccount =
            new SecureProxy(new Account(),
                           "credit",
                           "setBankAba",
                           "setBankAccountNumber",
                           "setBankAccountType",
                           "transferFromBank");

        return (Accountable)Proxy.newProxyInstance(
            Accountable.class.getClassLoader(),
            new Class[] { Accountable.class },
            secureAccount);
    }
}

```

**Building the
Secure Account
Solution**

The use of a factory means that the client is isolated from the fact that a security proxy even exists. The client can request an account and, based on the `Permission` enum passed, `AccountFactory` creates either a real `Account` object (`Permission.UPDATE`) or a dynamic proxy object (`Permission.READ_ONLY`). The work of creating the dynamic proxy appears in the method `createSecuredAccount`.

The class `SecureProxy` is the dynamic proxy object that you will construct. It could act as a secure proxy for any target class. To construct a `SecureProxy`, you pass it the target object and a list of the methods that must be secured. (The list of methods could easily come from a database lookup. A security administrator could populate the contents of the database through another part of the SIS application.)

The second statement in `createSecuredAccount` is the way that you set up the `SecureProxy` object to act as a dynamic proxy. It's an ugly bit of code, so I'll repeat it here with reference numbers:

```

return (Accountable)Proxy.newProxyInstance( // 1
    Accountable.class.getClassLoader(), // 2
    new Class[] { Accountable.class }, // 3
    secureAccount); // 4

```

Line 1 invokes the Proxy factory method `newProxyInstance`. This method is not parameterized, thus it returns an `Object`. You must cast the return value to an `Accountable` interface reference.

The first parameter to `newProxyInstance` (line 2) requires the *class loader* for the interface class. A class loader reads a stream of bytes representing a Java compilation unit from some source location. Java contains a default class loader, which reads class files from a disc file, but you can create custom class loaders that read class files directly from a database or from a remote source, such as the internet. In most cases you will want to call the method `getClassLoader` on the `Class` object itself; this method returns the class loader that originally loaded the class.

The SecureProxy Class

The second parameter (line 3) is an array of interface types for which you want to create a dynamic proxy. Behind the scenes, Java will use this list to dynamically construct an object that implements all of the interfaces.

The type of the final parameter (line 4) is `InvocationHandler`, an interface that contains a single method that your dynamic proxy class must implement in order to intercept incoming messages. You pass your proxy object as this third parameter.

The SecureProxy Class

`SecureProxyTest:`

```
package sis.security;

import java.lang.reflect.*;
import junit.framework.*;

public class SecureProxyTest extends TestCase {
    private static final String secureMethodName = "secure";
    private static final String insecureMethodName = "insecure";
    private Object object;
    private SecureProxy proxy;
    private boolean secureMethodCalled;
    private boolean insecureMethodCalled;

    protected void setUp() {
        object = new Object() {
            public void secure() {
                secureMethodCalled = true;
            }
            public void insecure() {
                insecureMethodCalled = true;
            }
        };
    }
}
```

```

    };
    proxy = new SecureProxy(object, secureMethodName);
}

public void testSecureMethod() throws Throwable {
    Method secureMethod =
        object.getClass().getDeclaredMethod(
            secureMethodName, new Class[]{});
    try {
        proxy.invoke(proxy, secureMethod, new Object[]{});
        fail("expected PermissionException");
    }
    catch (PermissionException expected) {
        assertFalse(secureMethodCalled);
    }
}

public void testInsecureMethod() throws Throwable {
    Method insecureMethod =
        object.getClass().getDeclaredMethod(
            insecureMethodName, new Class[]{});
    proxy.invoke(proxy, insecureMethod, new Object[]{});
    assertTrue(insecureMethodCalled);
}
}
}

```

The
SecureProxy
Class

Some of the code in `SecureProxyTest` is similar to that of `AccountFactoryTest`. Some refactoring of common test code is probably a good idea. The main distinction between the two tests is that `AccountFactoryTest` needs to ensure that all methods defined on the `Accountable` interface are covered properly. `SecureProxyTest` instead works with a pure test-class definition.

The `setUp` method uses an anonymous inner class construct to define a new unnamed class with two methods, `secure` and `insecure`. All that these methods do is set a corresponding `boolean` instance variable if they are called. The second statement in the `setUp` method creates a `SecureProxy` object by passing in its target object—the anonymous inner class instance—along with the name of the method to be secured for testing purposes.

The test `testSecureMethod` first looks up the correct `Method` object on the anonymous class. The test then passes this `Method` object, along with the proxy instance and an empty parameter list, to the `invoke` method on the proxy object. Normally you would never call the `invoke` method directly, but there's no reason you can't do so for purposes of testing.

```
proxy.invoke(proxy, secureMethod, new Object[]{});
```

The test `testSecureMethod` uses the expected exception testing idiom, looking for a `PermissionException`. It also ensures that the secure method was never called. The test `testInsecureMethod` ensures that the insecure method *was* called.

**The
SecureProxy
Class**

```
package sis.security;

import java.lang.reflect.*;
import java.util.*;

public class SecureProxy implements InvocationHandler {
    private List<String> secureMethods;
    private Object target;

    public SecureProxy(Object target, String... secureMethods) {
        this.target = target;
        this.secureMethods = Arrays.asList(secureMethods);
    }

    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        try {
            if (isSecure(method))
                throw new PermissionException();
            return method.invoke(target, args);
        }
        catch (InvocationTargetException e) {
            throw e.getTargetException();
        }
    }

    private boolean isSecure(Method method) {
        return secureMethods.contains(method.getName());
    }
}
```

SecureProxy stores the target object and the list of methods to be secured in its constructor. It implements the sole method in InvocationHandler, `invoke`. Any messages sent to the proxy get routed to the `invoke` method. As you saw in the test, the `invoke` method takes three parameters: the proxy object itself, the method being called, and the parameters to the method.¹⁴ It's up to you to interpret the incoming information.

In SecureProxy, your interpretation is to look up the method name to see if it is in the list of secure methods. If so, you throw a `PermissionException`. If not, you delegate the message to the target by calling `invoke` on the incoming `Method` object.

That's it!! Your tests should pass.

¹⁴If it has not been apparent until now, primitive parameters are autoboxed into object references.

Problems With Reflection

As you've seen, you can do some very cool things using reflection. Reflection is almost essential for some applications. JUnit currently depends heavily upon reflection. Many other Java technologies require the use of reflection, including EJBs and JavaBeans. Reflection can allow you to accomplish tasks that would otherwise be virtually impossible to do in Java. However, you should approach the use of reflection with caution.

First, code using reflection can be hard to decipher and debug. Modern IDEs, such as Eclipse, go a long way toward helping you easily navigate a system. Eclipse, for example, allows you to find all references to a class or a method. It cannot, however, as effectively find code that uses reflection to create the class or method call you are interested in. Reflection introduces a hole in your code's traceability!

Code that uses reflection executes considerably slower than equivalent code that does not use it. When you introduce reflection code that will be used frequently or on a large scale, profile it to ensure that it executes in a timely fashion.

Finally, when you use reflection, your code can exhibit defects that you might have otherwise caught at compile time. You must ensure that your code defensively, for example by dealing with the `ClassNotFoundException` that could be thrown by `Class.forName`.



Don't use reflection for the sake of using reflection.

This is Jeff's Rule of Statics (from Lesson 4) applied to reflection.

Exercises

1. Use an anonymous inner class of type `Comparable` to sort the chess-board positions that you created in an earlier exercise.
2. When you debug, you will often find that a `toString` method doesn't provide you with enough information. At times like this, it may be handy to have a utility method to expose, or "dump," information that is more internal.

Create an object-dumper utility class that takes an object and lists every field name on the object and the current dump of those objects, recursively, using a hierarchical output format. Do not traverse into

Exercises

classes from `java` or `javax` packages. The utility should be able to display private fields. Mark static fields as such. For simplicity's sake, ignore fields from any superclass.

3. Yet another clone exercise: Implement another version of the poor man's clone, this time using reflection operations. Obtain a Constructor object from the source object's class. Call `newInstance` to create the new object and copy each field's values from the source object to the new object. The class being cloned will need to provide a no-argument constructor. Build support for a shallow copy only.
4. Create a Proxy class that delegates to the original object for every method it defines, except for `toString`. When the proxy receives the `toString` message, it delegates to the object dumper instead. The target object will need to implement an interface type that defines `toString`, since proxies work off an interface.

Exercises

Lesson 13

Multithreading

This chapter presents the most difficult core Java technology to understand and master: multithreading. So far, you have been writing code to run in a single *thread* of execution. It runs serially, start to finish. You may have the need, however, to *multithread*, or execute multiple passages of code simultaneously.

You will learn about:

Multithreading

- suspending execution in a thread
- creating and running threads by extending Thread
- creating and running threads by implementing Runnable
- cooperative and preemptive multitasking
- synchronization
- BlockingQueue
- stopping threads
- `wait` and `notify` methods
- locks and conditions
- thread priorities
- deadlocks
- ThreadLocal
- the timer classes
- basic design principles for multithreading

Multithreading

Many multithreading needs are related to building responsive user interfaces. For example, most word processing applications contain an “autosave” function. You can configure the word processor to save your documents every x minutes. The autosave executes automatically on schedule without needing your intervention. A save may take several seconds to execute, yet you can continue work without interruption.

The word processor code manages at least two threads. One thread, often referred to as the foreground thread, is managing your direct interaction with the word processor. Any typing, for example, is captured by code executing in the foreground thread. Meanwhile, a second, background thread is checking the clock from time to time. Once the configured number of minutes has elapsed, the second thread executes the code in the save function.

On a multiple-processor machine, multiple threads can actually run simultaneously, each thread on a separate processor. On a single-processor machine, threads each get a little slice of time from the processor (which can usually only execute one thing at a time), making the threads appear to execute simultaneously.

Learning to write code that runs in a separate thread is fairly simple. The challenge is when multiple threads need to share the same resource. If you are not careful, multithreaded solutions can generate incorrect answers or freeze up your application. Testing multithreaded solutions is likewise complex.

Search Server

Search Server

A server class may need to handle a large number of incoming requests. If it takes more than a few milliseconds to handle each request, clients who make the request may have to wait longer than an acceptable period of time while the server works on other incoming requests. A better solution is to have the server store each incoming request as a search on a *queue*. Another thread can then take searches from the queue in the order they arrived and process them one at a time. This is known as the active object pattern;¹ it decouples the method request from the actual execution of the method. See Figure 13.1. All relevant search information is translated into a *command object* for later execution.

¹[Lavender1996].

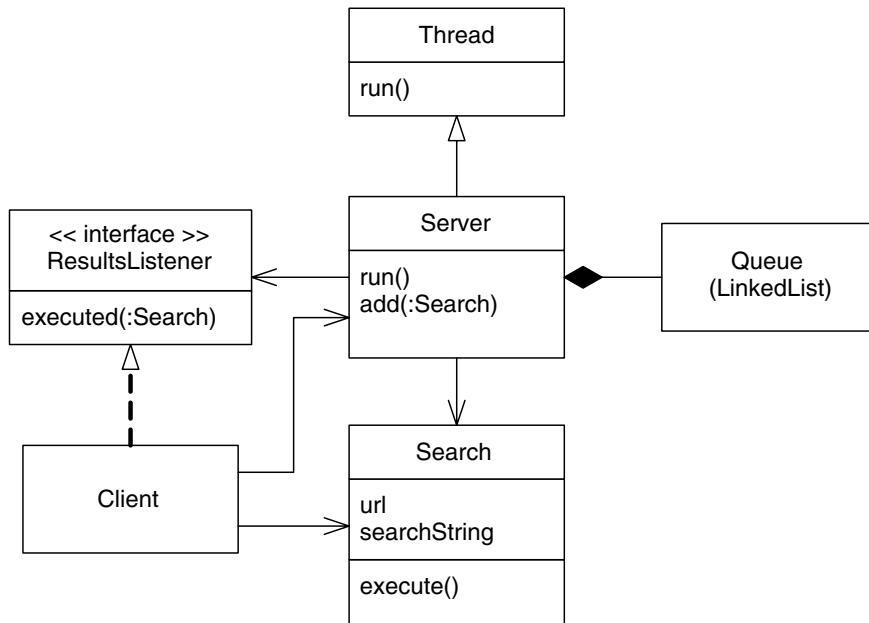


Figure 13.1 Active Object

The Search Class

 You will build a simple web search server. The server will take a search object consisting of a URL and a search string as a request. As it searches the URL, the server will populate the search object with the number of times the string was found in the document at the URL.

Note: If you're not connected to the Internet, the search test will not execute properly. This is corrected in short time—the section entitled Less Dependent Testing shows how to modify the test to execute against local URLs.

The Search Class

The simplest approach to creating a search server is to follow the Single-Responsibility Principle and first design a class that supports a single search. Once it works, you can concern yourself with the multithreading needs. The benefit is that in keeping each search as a separate object, you have fewer concerns with respect to shared data.

The SearchTest class tests a few possible cases:

```

package sis.search;

import junit.framework.TestCase;
import java.io.*;
  
```

The Search Class

```
public class SearchTest extends TestCase {
    private static final String URL = "http://www.langrsoft.com";
    public void testCreate() throws IOException {
        Search search = new Search(URL, "x");
        assertEquals(URL, search.getUrl());
        assertEquals("x", search.getText());
    }

    public void testPositiveSearch() throws IOException {
        Search search = new Search(URL, "Jeff Langr");
        search.execute();
        assertTrue(search.matches() >= 1);
        assertFalse(search.errorred());
    }

    public void testNegativeSearch() throws IOException {
        final String unlikelyText = "mama cass elliot";
        Search search = new Search(URL, unlikelyText);
        search.execute();
        assertEquals(0, search.matches());
        assertFalse(search.errorred());
    }

    public void testErroredSearch() throws IOException {
        final String badUrl = URL + "/z2468.html";
        Search search = new Search(badUrl, "whatever");
        search.execute();
        assertTrue(search.errorred());
        assertEquals(FileNotFoundException.class,
                    search.getError().getClass());
    }
}
```

The implementation a:

```
package sis.search;

import java.net.*;
import java.io.*;
import sis.util.*;

public class Search {
    private URL url;
    private String searchString;
    private int matches = 0;
    private Exception exception = null;

    public Search(String urlString, String searchString)
        throws IOException {
        this.url = new URL(urlString);
        this.searchString = searchString;
    }
```

```
public String getText() {
    return searchString;
}
public String getUrl() {
    return url.toString();
}
public int matches() {
    return matches;
}
public boolean errored() {
    return exception != null;
}
public Exception getError() {
    return exception;
}
}

public void execute() {
    try {
        searchUrl();
    }
    catch (IOException e) {
        exception = e;
    }
}
}

private void searchUrl() throws IOException {
    URLConnection connection = url.openConnection();
    InputStream input = connection.getInputStream();
    BufferedReader reader = null;
    try {
        reader = new BufferedReader(new InputStreamReader(input));
        String line;
        while ((line = reader.readLine()) != null)
            matches += StringUtil.occurrences(line, searchString);
    }
    finally {
        if (reader != null)
            reader.close();
    }
}
```

The Search
Class

The method `searchUrl` in `Search` uses the class `java.net.URL` to obtain a `java.netURLConnection` object. A `URLConnection` is used to establish communication between the client and the actual URL. Once you send the `openConnection` message to a `URLConnection`, you can send the message `getInputStream` to obtain an `InputStream` reference. The remainder of the method uses Java IO code (see Lesson 11) to search the file.

Currently, `searchUrl` invokes the `StringUtil` method `occurrences` to count the number of matches within each line read from the input stream. Here are `StringUtilTest` and `StringUtil`.

```
// StringUtilTest.java
package sis.util;

import junit.framework.*;

public class StringUtilTest extends TestCase {
    private static final String TEXT = "this is it, isn't it";
    public void testOccurrencesOne() {
        assertEquals(1, StringUtil.occurrences(TEXT, "his"));
    }
    public void testOccurrencesNone() {
        assertEquals(0, StringUtil.occurrences(TEXT, "smelt"));
    }
    public void testOccurrencesMany() {
        assertEquals(3, StringUtil.occurrences(TEXT, "is"));
        assertEquals(2, StringUtil.occurrences(TEXT, "it"));
    }
    public void testOccurrences searchStringTooLarge() {
        assertEquals(0, StringUtil.occurrences(TEXT, TEXT + "sdfas"));
    }
}

// StringUtil.java
package sis.util;

public class StringUtil {
    static public int occurrences(String string, String substring) {
        int occurrences = 0;
        int length = substring.length();
        final boolean ignoreCase = true;
        for (int i = 0; i < string.length() - substring.length() + 1; i++)
            if (string.regionMatches(ignoreCase, i, substring, 0, length))
                occurrences++;
        return occurrences;
    }
}
```

Less Dependent Testing

Another solution would be to use Java's regular expression (regex) API. See Additional Lesson III for a discussion of regex.

Less Dependent Testing

If you're unfortunate enough to not be connected to the Internet on your computer, you're probably grousing about your inability to execute `ServerTest` at all. You will now rectify this situation by changing the search

URLs to file URLs.² This also means you can write out test HTML files to control the content of the “web” pages searched.

```
package sis.search;

import junit.framework.TestCase;
import java.io.*;
import java.util.*;
import sis.util.*;

public class SearchTest extends TestCase {
    public static final String[] TEST_HTML = {
        "<html>",
        "<body>",
        "Book: Agile Java, by Jeff Langr<br />",
        "Synopsis: Mr Langr teaches you<br />",
        "Java via test-driven development.<br />",
        "</body></html>"};

    public static final String FILE = "/temp/testFileSearch.html";
    public static final String URL = "file:" + FILE;

    protected void setUp() throws IOException {
        TestUtil.delete(FILE);
        LineWriter.write(FILE, TEST_HTML);
    }

    protected void tearDown() throws IOException {
        TestUtil.delete(FILE);
    }
    // ...
}
```

Less Dependent Testing

You will need the LineWriter utility; here is the test and production source.

```
// LineWriterTest.java
package sis.util;

import junit.framework.*;
import java.io.*;

public class LineWriterTest extends TestCase {
    public void testMultipleRecords() throws IOException {
        final String file = "LineWriterTest.testCreate.txt";
        try {
            LineWriter.write(file, new String[] {"a", "b"});

            BufferedReader reader = null;
            try {
                reader = new BufferedReader(new FileReader(file));
                assertEquals("a", reader.readLine());
                assertEquals("b", reader.readLine());
            } finally {
                if (reader != null) reader.close();
            }
        } finally {
            if (file != null) TestUtil.delete(file);
        }
    }
}
```

²Another solution would involve installing a local web server, such as Tomcat.

```

        assertNull(reader.readLine());
    }
    finally {
        if (reader != null)
            reader.close();
    }
}
finally {
    TestUtil.delete(file);
}
}

// LineWriter.java
package sis.util;

import java.io.*;

public class LineWriter {
    public static void write(String filename, String[] records)
        throws IOException {
        BufferedWriter writer = null;
        try {
            writer = new BufferedWriter(new FileWriter(filename));
            for (int i = 0; i < records.length; i++) {
                writer.write(records[i]);
                writer.newLine();
            }
        }
        finally {
            if (writer != null)
                writer.close();
        }
    }
}

```

Less Dependent Testing

The good part is that the changes to SearchTest are not very invasive. In fact, none of the test methods need change. You are adding setup and teardown methods to write HTML to a local file and subsequently delete that file.

Initialization Expressions

The method `searchUrl` contains the code:

```
String line;
while ((line = reader.readLine()) != null)
```

The parentheses should help you understand this code. First, the result of `reader.readLine()` is assigned to the `line` reference variable. The value of the `line` reference is compared to `null` to determine whether or not the `while` loop should terminate.

each time a test completes. You are also changing the URL to use the file protocol instead of the http protocol.

The bad part is that the Search class must change. To obtain an InputStream from a URL with a file protocol, you must extract the path information from the URL and use it to open the stream as a FileInputStream. It's not a significant change. The minor downside is that you now have a bit of code in your production system that will probably only be used by the test.

```
private void searchUrl() throws IOException {
    InputStream input = getInputStream(url);
    BufferedReader reader = null;
    try {
        reader = new BufferedReader(new InputStreamReader(input));
        String line;
        while ((line = reader.readLine()) != null)
            matches += StringUtil.occurrences(line, searchString);
    }
    finally {
        if (reader != null)
            reader.close();
    }
}

private InputStream getInputStream(URL url) throws IOException {
    if (url.getProtocol().startsWith("http")) {
        URLConnection connection = url.openConnection();
        return connection.getInputStream();
    }
    else if (url.getProtocol().equals("file")) {
        return new FileInputStream(url.getPath());
    }
    return null;
}
```

Less Dependent Testing

One other thing you want to consider is that you are no longer exercising the portion of the code that deals with an http URL. The best approach is to ensure that you have adequate coverage at an *acceptance test*³ level. Acceptance tests provide a level of testing above unit testing—they test the system from an end user's standpoint. They are executed against a live system as possible and should not use mocks. For the search application, you would certainly execute acceptance tests that used live web URLs.

³There are many different nomenclatures for different types of testing. Acceptance tests are the tests that signify that the system meets the acceptance criteria of the customer to whom it's delivered. You may hear these tests referred to as customer tests.

The Server

ServerTest:

```
package sis.search;

import junit.framework.*;
import sis.util.*;

public class ServerTest extends TestCase {
    private int numberOfResults = 0;
    private Server server;
    private static final long TIMEOUT = 3000L;
    private static final String[] URLs = {
        SearchTest.URL, SearchTest.URL, SearchTest.URL }; // 1

    protected void setUp() throws Exception {
        TestUtil.delete(SearchTest.FILE);
        LineWriter.write(SearchTest.FILE, SearchTest.TEST_HTML);

        ResultsListener listener = new ResultsListener() { // 2
            public void executed(Search search) {
                numberOfResults++;
            }
        };

        server = new Server(listener);
    }

    protected void tearDown() throws Exception {
        TestUtil.delete(SearchTest.FILE);
    }

    public void testSearch() throws Exception {
        long start = System.currentTimeMillis();
        for (String url: URLs) // 3
            server.add(new Search(url, "xxx"));
        long elapsed = System.currentTimeMillis() - start;
        long averageLatency = elapsed / URLs.length;
        assertTrue(averageLatency < 20); // 4
        assertTrue(waitForResults()); // 5
    }

    private boolean waitForResults() {
        long start = System.currentTimeMillis();
        while (numberOfResults < URLs.length) {
            try { Thread.sleep(1); }
            catch (InterruptedException e) {}
            if (System.currentTimeMillis() - start > TIMEOUT)
                return false;
        }
        return true;
    }
}
```

The Server

The test first constructs a list of URL strings (line 1).

The setup method constructs a ResultsListener object (line 2). You pass this object as a parameter when constructing a new Server instance.

Your test, as a client, will only be to pass search requests to the server. In the interest of responsiveness, you are designing the server so the client will not have to wait for each request to complete. But the client will still need to know the results each time the server gets around to processing a search. A frequently used mechanism is a *callback*.

The term callback is a holdover from the language C, which allows you to create pointers to functions. Once you have a function pointer, you can pass this pointer to other functions like any other reference. The receiving code can then *call back* to the function located in the originating code, using the function pointer.

In Java, one effective way to implement a callback is to pass an anonymous inner class instance as a parameter to a method. The interface ResultsListener defines an executed method:

```
package sis.search;

public interface ResultsListener {
    public void executed(Search search);
}
```

The Server

In testSearch, you pass an anonymous inner class instance of ResultsListener to the constructor of Server. The Server object holds on to the ResultsListener reference and sends it the executed message when the search has completed execution.

Callbacks are frequently referred to as listeners in Java implementations. A listener interface defines methods that you want called when something happens. User interface classes frequently use listeners to notify other code when an event occurs. For example, you can configure a Java Swing class with a event listener that lets you know when a user clicks on the button to close a window. This allows you to tie up any loose ends before the window actually closes.

In the test, you use the ResultsListener instance simply to track the total number of search executions that complete.

Once you create a Server object, you use a loop (line 3) to iterate through the list of URLs. You use each URL to construct a Search object (the search text is irrelevant). In the interest of demonstrating that Java rapidly dispatches each request, you track the elapsed execution time to create and add a search to the server. You use a subsequent assertion (line 4) to show that the average latency (delay in response time) is an arbitrarily small 20 milliseconds or less.

Since the server will be executing multiple searches in a separate thread, the likelihood is that the code in the JUnit test will complete executing before the searches do. You need a mechanism to hold up processing until the searches are complete—until the number of searches executed is the same as the number of URLs searched.

Waiting in the Test

The assertion at line 5 in the `ServerTest` listing calls the `waitForResults` method, which will suspend execution of the current thread (i.e., the thread in which the test is executing) until all search results have been retrieved. The `timeout` value provides an arbitrary elapsed time after which `waitForResults` should return `false` and cause the assertion to fail.

Waiting in the Test

```
private boolean waitForResults() {
    long start = System.currentTimeMillis();
    while (numberOfResults < URLs.length) {
        try { Thread.sleep(1); }
        catch (InterruptedException e) {}
        if (System.currentTimeMillis() - start > TIMEOUT)
            return false;
    }
    return true;
}
```

The `waitForResults` method executes a simple loop. Each time through, the body of the loop pauses for a millisecond, then makes a quick calculation of elapsed time to see if it's passed the timeout limit. The pause is effected by a call to the static `Thread` method `sleep`. The `sleep` method takes a number of milliseconds and idles the currently executing thread for that amount of time.⁴

Since `sleep` can throw a checked exception of type `InterruptedException`, you can choose to enclose it in a `try-catch` block. It is possible for one thread to interrupt another, which is what would generate the exception. But thread interruptions are usually only by design, so the code here shows one of the rare cases where it's acceptable to ignore the exception and provide an empty `catch` block.

The looping mechanism used in `waitForResults` is adequate at best. The `wait/notify` technique, discussed later in this lesson (see Wait/Notify), provides the best general-purpose mechanism of waiting for a condition to occur.

⁴The thread scheduler will wait *at least* the specified amount of time and possibly a bit longer.

Creating and Running Threads

The Server class needs to accept incoming searches and simultaneously process any requests that have not been executed. The code to do the actual searches will be executed in a separate thread. The main thread of Server, in which the remainder of the code executes, must spawn the second thread.

Java supplies two ways for you to initiate a separate thread. The first requires you to extend the class `java.lang.Thread` and provide an implementation for the `run` method. Subsequently, you can call the `start` method to kick things off. At that point, code in the `run` method begins execution in a separate thread.

A second technique for spawning threads is to create an instance of a class that implements the interface `java.lang.Runnable`:

```
public interface Runnable {
    void run();
}
```

**Creating and
Running
Threads**

You then construct a `Thread` object, passing the `Runnable` object as a parameter. Sending the message `start` to the `Thread` object initiates code in the `run` method. This second technique is often done using anonymous inner class implementations of `Runnable`.

It is important that you distinguish between a *thread*, or thread of execution, and a *Thread* object. A thread is a control flow that the thread scheduler manages. A `Thread` is an object that manages information about a thread of execution. The existence of a `Thread` object does not imply the existence of a thread: A thread doesn't exist until a `Thread` object starts one, and a `Thread` object can exist long after a thread has terminated.

For now, you'll use the first technique to create a thread and have the `Server` class extend from `Thread`. `Thread` itself implements the `Runnable` interface. If you extend `Thread`, you will need to override the `run` method in order for anything useful to happen.

```
package sis.search;

import java.util.*;

public class Server extends Thread {
    private List<Search> queue = new LinkedList<Search>(); // flaw!
    private ResultsListener listener;

    public Server(ResultsListener listener) {
        this.listener = listener;
        start();
    }
}
```

```

public void run() {
    while (true) {
        if (!queue.isEmpty())
            execute(queue.remove(0));
        Thread.yield();
    }
}

public void add(Search search) {
    queue.add(search);
}

private void execute(Search search) {
    search.execute();
    listener.executed(search);
}
}

```

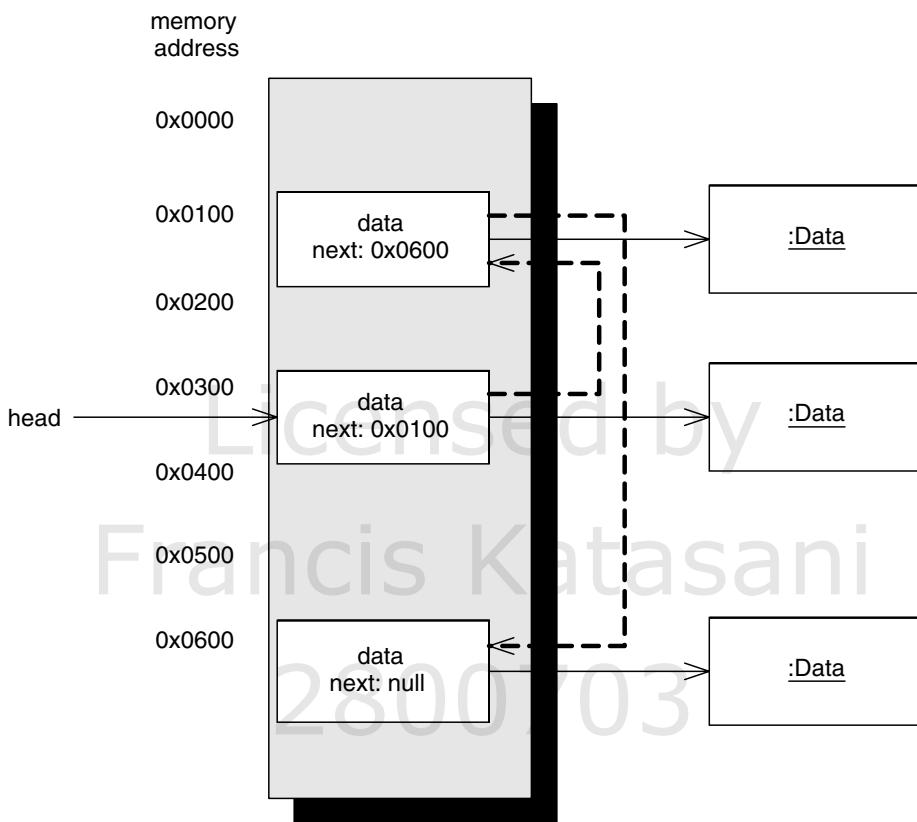
Creating and Running Threads

The `Server` class defines two fields: a `ResultsListener` reference and a `LinkedList` of `Search` objects, named `queue`. A flaw exists in the declaration of the `queue` reference—it is not “thread safe”! For now, your test will likely execute successfully, but the flaw could cause your application to fail. In the section `Synchronized Collections` in this chapter, you will learn about this thread safety issue and how to correct it.

The `java.util.LinkedList` class implements the `List` interface. Instead of storing elements in a contiguous block of memory, as in an `ArrayList`, a linked list allocates a new block of memory for each element you add to it. The individual blocks of memory thus will be scattered in memory space; each block contains a link to the next block in the list. See Figure 13.2 for a conceptual memory picture of how a linked list works. For collections that require frequent removals or insertions at points other than the end of the list, a `LinkedList` will provide better performance characteristics than an `ArrayList`.

The `queue` reference stores the incoming search requests. The linked list in this case acts as a *queue* data structure. A queue is also known as a first-in, first-out (FIFO) list: When you ask a queue to remove elements, it removes the oldest item in the list first. The `add` method in `Server` takes a `Search` parameter and adds it to the *end* of the queue. So in order for the `LinkedList` to act as a queue, you must remove `Search` objects from the *beginning* of the list.

The constructor of `Server` kicks off the second thread (often called a *worker* or *background* thread) by calling `start`. The `run` method is invoked at this point. The `run` method in `Server` is an infinite loop—it will keep executing until some other code explicitly terminates the thread (see `Stopping Threads`



Creating and
Running
Threads

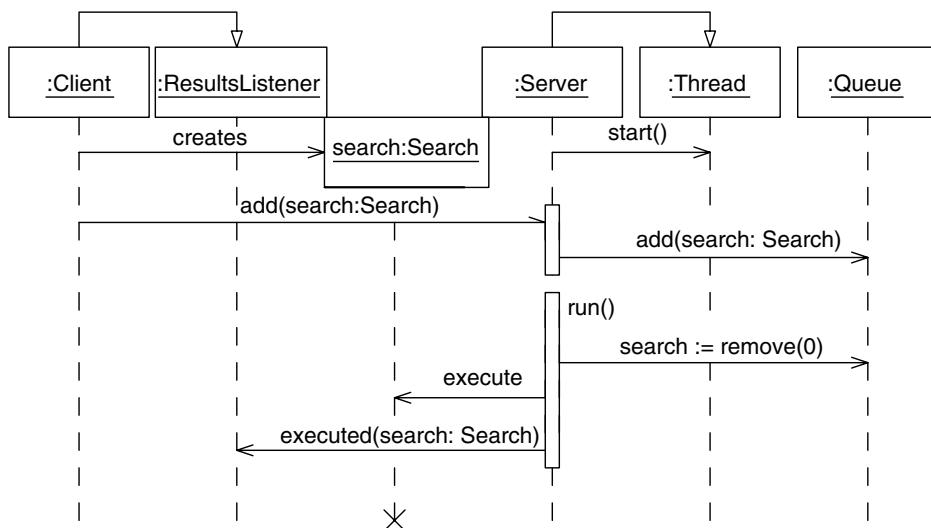
Figure 13.2 A Conceptual View of a Linked List

in this chapter) or until the application in which it is running terminates (see Shutting Down in this chapter). When you run `ServerTest` in JUnit, the background thread will keep executing even after all the JUnit tests have completed. It will stop only when you close the JUnit window.

The body of the loop first checks whether the queue is empty. If not, the code removes and executes the first element in the queue. Regardless, the code then calls the `Thread` method `yield`. The `yield` method potentially allows other threads the opportunity to get some time from the processor before the background thread picks up again. The next section on cooperative and preemptive multitasking discusses why the `yield` might be necessary.

The `execute` method delegates the actual search to the `Search` object itself. When the search completes, the `ResultsListener` reference is sent the message `executed` with the `Search` object as a parameter.

You can represent the flow of messages in a successful search using a UML sequence diagram (Figure 13.3). The sequence diagram is a dynamic view of your system in action. It shows the ordered message flow between objects. I



Creating and Running Threads

Figure 13.3 Sequence Diagram for the Active Object Search

find that sequence diagrams can be a very effective means of communicating how a system is wired together with respect to various uses.⁵

In a sequence diagram, you show object boxes instead of class boxes. The dashed line that emanates vertically downward from each box represents the object's lifeline. An "X" at the bottom of the object lifeline indicates its termination. The Search object in Figure 13.3 disappears when it is notified that a search has been completed; thus, its lifeline is capped off with an "X."

You represent a message send using a directed line from the sending object's lifeline to the receiver. Messages flow in order from top to bottom. In Figure 13.3, a Client object sends the first message, creates. This special message shows that the Client object is responsible for creation of a Search object. Note that the Search object starts farther down; it is not in existence until this point.

After the first message send, the Client sends add(Search) to the Server object. This message send is conceptually asynchronous—the Client need not wait for any return information from the Server object.⁶ You use a half-arrowhead to represent asynchronous message sends. When it receives this add message, the Server's add method passes the Search object off to the queue. You represent the execution lifetime of the add method at the Server object using an *activation*—a thin rectangle laid over the object lifeline.

⁵Sequence diagrams sometimes are not the best way of representing complex parallel activity. A UML model that is perhaps better suited for representing multithreaded behavior is the *activity diagram*.

⁶The way we've implemented it, the message send is synchronous, but since the operation is immediate and returns no information you can represent it as asynchronous.

Meanwhile, the Server thread has sent the message `start` to its superclass (or, more correctly, to itself). The `run` method overridden in Server begins execution. Its lifetime is indicated by an activation on the Server object's lifeline. Code in the `run` method sends the message `remove(0)` to the Queue to obtain and remove its first Search element. Subsequently the `run` method sends the message `execute` to the Search object and notifies the ResultsListener (via `executed`) when `execute` completes.

Cooperative and Preemptive Multitasking

In a single-processor environment, individual threads each get a slice of time from the processor in which to execute. The question is, how much time does each thread get before another thread takes over? The answer depends upon many things, including which operating system you are using to run Java and your particular JVM implementation.

Most modern operating systems (including Unix variants and Windows) use *preemptive multitasking*, in which the operating system (OS) interrupts the currently executing thread. Information about the thread is stored, and the OS goes on to provide a slice of time to the next thread. In this environment, all threads will eventually get some attention from the thread scheduler.

**Cooperative
and Preemptive
Multitasking**

The other possibility is that the operating system manages threads using *cooperative multitasking*. Cooperative multitasking depends on thread code behaving well by yielding time to other threads frequently. In a cooperative threading model, one poorly written thread could hog the processor completely, preventing all other threads from doing any processing. Threads may yield time by explicitly calling the `yield` method, by sleeping, when blocking (waiting) on IO operations, and when they are suspended, to mention a few ways.

In the Server class, the `while` loop in the `run` method calls the `yield` method each time through the loop to allow other threads the opportunity to execute.⁷

```
public void run() {
    while (true) {
        if (!queue.isEmpty())
            execute(queue.remove(0));
        Thread.yield();
    }
}
```

⁷You may experience significant CPU usage when executing this code, depending on your environment. The `yield` method may effectively do nothing if there are no other threads executing. An alternative would be to introduce a sleep of a millisecond.

Synchronization

One of the biggest pitfalls with multithreaded development is dealing with the fact that threads run in *nondeterministic* order. Each time you execute a multithreaded application, the threads may run in a different order that is impossible to predict. The most challenging result of this nondeterminism is that you may have coded a defect that surfaces only once in a few thousand executions, or once every second full moon.⁸

Two threads that are executing code do not necessarily move through the code at the same rate. One thread may execute five lines of code before another has even processed one. The thread scheduler interleaves slices of code from each executing thread.

Also, threading is not at the level of Java statements, but at a lower level of the VM operations that the statements translate down to. Most statements you code in Java are *non-atomic*: The Java compiler might create several internal operations to correspond to a single statement, even something as simple as incrementing a counter or assigning a value to a reference. Suppose an assignment statement takes two internal operations. The second operation may not complete before the thread scheduler suspends the thread and executes a second thread.

All this code interleaving means that you may encounter *synchronization* issues. If two threads go after the same piece of data at the same time, the results may not be what you expect.

You will modify the sis.studentinfo.Account class to support withdrawing funds. To withdraw funds, the balance of the account must be at least as much as the amount being withdrawn. If the amount is too large, you should do nothing (for simplicity's and demonstration's sake). In any case, you never want the account balance to go below zero.

```
package sis.studentinfo;
// ...
public class AccountTest extends TestCase {
    // ...
    private Account account;

    protected void setUp() {
        account = new Account();
        // ...
    }
    // ...
```

⁸The subtle gravitational pull might cause electrostatic tides that slow the processor for a picosecond.

```

public void testWithdraw() throws Exception {
    account.credit(new BigDecimal("100.00"));
    account.withdraw(new BigDecimal("40.00"));
    assertEquals(new BigDecimal("60.00"), account.getBalance());
}

public void testWithdrawInsufficientFunds() {
    account.credit(new BigDecimal("100.00"));
    account.withdraw(new BigDecimal("140.00"));
    assertEquals(new BigDecimal("100.00"), account.getBalance());
}
// ...
}

```

The withdraw method:

```

package sis.studentinfo;
// ...
public class Account implements Accountable {
    private BigDecimal balance = new BigDecimal("0.00");
    // ...
    public void withdraw(BigDecimal amount) {
        if (amount.compareTo(balance) > 0)
            return;
        balance = balance.subtract(amount);
    }
    // ...
}

```

Synchronization

A fundamental problem exists with the withdraw method in a multithreaded environment. Suppose two threads attempt to withdraw \$80 from an account with a \$100 balance at the same time. One thread should succeed; the other thread should fail. The end balance should be \$0. However, the code execution could interleave as follows:

Thread 1	Thread 2	Bal
amount.compareTo(balance) > 0		100
	amount.compareTo(balance) > 0	100
balance = balance.subtract(amount)		20
	balance = balance.subtract(amount)	-60

The second thread tests the balance after the first thread has approved the balance but before the first thread has subtracted from the balance. Other execution sequences could cause the same problem.

You can write a short test to demonstrate this problem.

```

package sis.studentinfo;

import junit.framework.*;
import java.math.BigDecimal;

public class MultithreadedAccountTest extends TestCase {
    public void testConcurrency() throws Exception {
        final Account account = new Account();
        account.credit(new BigDecimal("100.00"));

        Thread t1 = new Thread(new Runnable() {
            public void run() {
                account.withdraw(new BigDecimal("80.00"));
            }
        });
        Thread t2 = new Thread(new Runnable() {
            public void run() {
                account.withdraw(new BigDecimal("80.00"));
            }
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        assertEquals(new BigDecimal("20.00"), account.getBalance());
    }
}

```

Creating Threads with Runnable

There's definitely some code redundancy in this method. After you understand what the method is doing, make sure you refactor it to eliminate the duplication.

Creating Threads with Runnable

The method `testConcurrency` introduces two new things. First, it shows how you can create a thread by passing it an instance of the `Runnable` interface. Remember that the `Runnable` interface defines the single method `run`. The test constructs an anonymous inner class instance of `Runnable` for each thread. Note that the code in the `run` methods does not execute until the `start` methods are called.

The second new thing in `testConcurrency` is the use of the `Thread` method `join`. When you send the message `join` to a thread, execution in the current thread halts until that thread completes. The code in `testConcurrency` first waits until `t1` completes, then `t2`, before the test can proceed.

When you execute JUnit, `testConcurrency` will in all likelihood pass. In a small method, where everything executes quickly, the thread scheduler will probably allot enough time to a thread to execute the entire method before moving on to the next thread. You can force the issue by inserting a pause in the `withdraw` method:

```
public void withdraw(BigDecimal amount) {
    if (amount.compareTo(balance) > 0)
        return;
    try { Thread.sleep(1); }
    catch (InterruptedException e) {}
    balance = balance.subtract(amount);
}
```

You should now see a red bar.

Synchronized

Synchronized

Semantically, you want all of the code in `withdraw` to execute atomically. A thread should be able to execute the entire `withdraw` method, start to finish, without any other thread interfering. You can accomplish this in Java through use of the `synchronized` method modifier.

```
public synchronized void withdraw(BigDecimal amount) {
    if (amount.compareTo(balance) > 0)
        return;
    balance = balance.subtract(amount);
}
```

This implementation of synchronization in Java is known as *mutual exclusion*. Another way you can refer to the code protected by mutual exclusion is as a *critical section*.⁹ In order to ensure that the `withdraw` code executes mutually exclusively, Java places a lock on the object in which the thread's code is executing. While one thread has a lock on an object, no other thread can obtain a lock on that object. Other threads that try to do so will block until the lock is released. The lock is released when the method completes execution.

Java uses a concept known as monitors to protect data. A monitor is associated with each object; this monitor protects the object's instance data. A monitor is associated with each class; it protects the class's static data. When you acquire a lock, you are acquiring the associated monitor; only one thread can acquire a lock at any given time.¹⁰

⁹[Sun2004].

¹⁰<http://www.artima.com/insidejvm/ed2/threadsynch2.html>.

You should always try to lock the smallest amount of code possible, otherwise you may experience performance problems while other threads wait to obtain a lock. If you are creating small, composed methods as I've repetitively recommended, you will find that locking at the method level suffices for most needs. However, you can lock at a smaller atomicity than an entire method by creating a synchronized block. You must also specify an object to use as a monitor by enclosing its reference in parentheses after the `synchronized` keyword.

The following implementation of `withdraw` is equivalent to the above implementation.

```
public void withdraw(BigDecimal amount) {
    synchronized(this) {
        if (amount.compareTo(balance) > 0)
            return;
        balance = balance.subtract(amount);
    }
}
```

Synchronized Collections

A synchronized block requires the use of braces, even if it contains only one statement.

You can declare a class method as `synchronized`. When the VM executes a class method, it will obtain a lock on the `Class` object for which the method is defined.

Synchronized Collections

As I hinted earlier, the `Server` class contains a flaw with respect to the declaration of the queue as a `LinkedList`.

```
public class Server extends Thread {
    private List<Search> queue = new LinkedList<Search>(); // flaw!
```

When you work with the Java 2 Collection Class Framework with classes such as `ArrayList`, `LinkedList`, and `HashMap`, you should be cognizant of the fact that they are not thread-safe—the methods in these classes are not synchronized. The older collection classes, `Vector` and `Hashtable`, are synchronized by default. However, I recommend you *do not* use them if you need thread-safe collections.

Instead, you can use utility methods in the class `java.util.Collections` to enclose a collection instance in what is known as a synchronization wrapper. The synchronization wrapper obtains locks where necessary on the target

collection and delegates all messages off to the collection for normal processing. Modify the Server class to wrap the queue reference in a synchronized list:

```
public class Server extends Thread {
    private List<Search> queue =
        Collections.synchronizedList(new LinkedList<Search>());
    // ...
    public void run() {
        while (true) {
            if (!queue.isEmpty())
                execute(queue.remove(0));
            Thread.yield();
        }
    }
    public void add(Search search) throws Exception {
        queue.add(search);
    }
    // ...
}
```

BlockingQueue

Adding the synchronization wrapper doesn't require you to change any code that uses the queue reference.

Using a bit of analysis, it doesn't seem possible for there to be a synchronization problem with respect to the queue. The `add` method inserts to the end of the list; the `run` loop removes from the beginning of the queue only if the queue is not empty. There are no other methods that operate on the queue. However, there is a slim possibility that portions of the `add` and `remove` operations will execute at the same time. This concurrency could corrupt the integrity of the `LinkedList` instance. Using the synchronization wrappers will eliminate the defect.

BlockingQueue

A promising API addition in J2SE 5.0 is the concurrency library of classes, located in the package `java.util.concurrent`. It provides utility classes to help you solve many of the issues surrounding multithreading. While you should prefer use of this library, its existence doesn't mean that you don't need to learn the fundamentals and concepts behind threading.¹¹

Since queues are such a useful construct in multithreaded applications, the concurrency library defines an interface, `BlockingQueue`, along with five

¹¹The importance of learning how multiplication works before always using a calculator to solve every problem is an analogy.

specialized queue classes that implement this interface. Blocking queues implement another new interface, `java.util.Queue`, which defines queue semantics for collections. Blocking queues add concurrency-related capabilities to queues. Examples include the ability to wait for elements to exist when retrieving the next in line and the ability to wait for space to exist when storing elements.

You can rework the `Server` class to use a `LinkedBlockingQueue` instead of a `LinkedList`.

```
package sis.search;

import java.util.concurrent.*;

public class Server extends Thread {
    private BlockingQueue<Search> queue =
        new LinkedBlockingQueue<Search>();
    private ResultsListener listener;

    public Server(ResultsListener listener) {
        this.listener = listener;
        start();
    }

    public void run() {
        while (true)
            try {
                execute(queue.take());
            }
            catch (InterruptedException e) {
            }
    }

    public void add(Search search) throws Exception {
        queue.put(search);
    }

    private void execute(Search search) {
        search.execute();
        listener.executed(search);
    }
}
```

BlockingQueue

The `LinkedBlockingQueue` method `put` adds an element to the end of the queue. The `take` method removes an element from the beginning of the queue. It also waits until an element is available in the queue. The code doesn't look very different from your original implementation of `Server`, but it is now thread-safe.

Stopping Threads

A testing flaw still remains in `ServerTest`. The thread in the `Server` class is executing an infinite loop. Normally, the loop would terminate when the `Server` itself terminated. When running your tests in JUnit, however, the thread keeps running until you close down JUnit itself—the `Server run` method keeps chugging along.

The Thread API contains a `stop` method that would appear to do the trick of stopping the thread. It doesn't take long to read in the detailed API documentation that the `stop` method has been deprecated and is "inherently unsafe." Fortunately, the documentation goes on to explain why and what you should do instead. (Read it.) The recommended technique is to simply have the thread die on its own based on some condition. One way is to use a boolean variable that you initialize to `true` and set to `false` when you want the thread to terminate. The conditional in the `while` loop in the `run` method can test the value of this variable.

You will need to modify the `ServerTest` method `tearDown` to do the shutdown and verify that it worked.

Stopping
Threads

```
protected void tearDown() throws Exception {  
    assertTrue(server.isAlive());  
    server.shutdown();  
    server.join(3000);  
    assertFalse(server.isAlive());  
    TestUtil.delete(SearchTest.FILE);  
}
```

Remember, the `tearDown` method executes upon completion of each and every test method, regardless of whether or not an exception was thrown by the test. In `tearDown`, you first ensure that the thread is "alive" and running. You then send the message `shutdown` to the server. You've chosen the method name `shutdown` to signal the server to stop.

You then wait on the server thread by using the `join` method, with an arbitrary timeout of 3 seconds. It will take some time for the thread to terminate. Without the `join` method, the rest of the code in `tearDown` will likely execute before the thread is completely dead. Finally, after you issue the `shutdown` command to the `Server`, you ensure that its thread is no longer alive. The `isAlive` method is defined on `Thread` and `Server` is a subclass of `Thread`, so you can send the message `isAlive` to the `Server` instance to get your answer.

Since the `Server` class now uses a `LinkedBlockingQueue`, you cannot use a boolean flag on the `run` method's `while` loop. The `LinkedBlockingQueue` take

method *blocks*—it waits until a new object is available on the queue. It would wait forever if you left JUnit running.

One way you could get the LinkedBlockingQueue to stop waiting would be to put a special Search object on the queue, one that signifies that you're done. Each time you take an object from the queue, you would need to check to see if it was the special Search object and terminate the *run* method if it was.

Another way to stop LinkedBlockingQueue from waiting is to interrupt the thread by sending it the *interrupt* message, which generates an *InterruptedException*. If you catch an *InterruptedException*, you can then break out of the infinite *while* loop. Here is the implementation of this approach:

```
public class Server extends Thread {
    // ...
    public void run() {
        while (true)
            try {
                execute(queue.take());
            }
            catch (InterruptedException e) {
                break;
            }
    }
    // ...
    public void shutDown() throws Exception {
        this.interrupt();
    }
}
```

Wait/Notify

Wait/Notify

Java provides a mechanism to help you coordinate actions between two or more threads. Often you need to have one thread wait until another thread has completed its work. Java provides a mechanism known as *wait/notify*. You can call a *wait* method from within a thread in order to idle it. Another thread can wake up this waiting thread by calling a *notify* method.

An example? Let's rock! A clock—tic-toc!

 In this example, you'll build a clock class suitable for use in a clock application. A user interface (UI) for the clock application could display either a digital readout or an analog representation, complete with hour, minute, and second hands. While the UI is busy creating output that the user sees, the actual Clock class can execute a sepa-

rate thread to constantly track the time. The Clock thread will loop indefinitely. Every second, it will wake up and notify the user interface that the time has changed.

The UI class implements the ClockListener interface. You pass a ClockListener reference to a Clock object; the clock can call back to this listener with the changed time. With this listener-based design, you could replace an analog clock interface with a digital clock interface and not have to change the clock class (see Figure 13.4).

The tough part is going to be writing a test. Here's a strategy:

- Create a mock ClockListener that stores any messages it receives.
- Create the Clock instance, passing in the mock object.
- Start the clock up and wait until five messages have been received by the mock.
- Ensure that the messages received by the ClockListener, which are date instances, are each separated by one second.

Wait/Notify

The test below performs each of these steps. The tricky bit is getting the test to wait until the mock gets all the messages it wants. I'll explain that part, but I'll let you figure out the rest of the test, including the verification of the tics captured by the test listener.

```
package sis.clock;

import java.util.*;
import junit.framework.*;
```

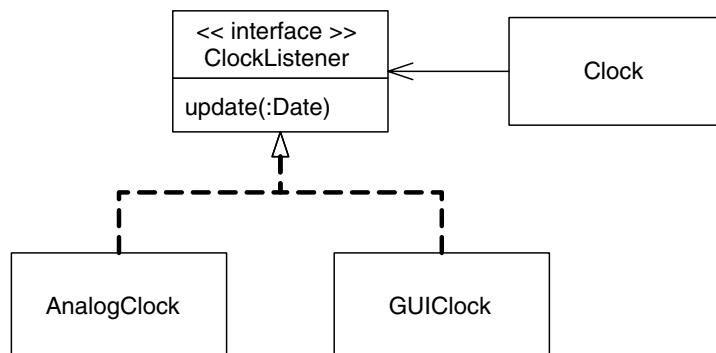


Figure 13.4 A Clock Listener

Wait/Notify

```

public class ClockTest extends TestCase {
    private Clock clock;
    private Object monitor = new Object(); // 1

    public void testClock() throws Exception {
        final int seconds = 5;
        final List<Date> tics = new ArrayList<Date>();
        ClockListener listener = new ClockListener() {
            private int count = 0;
            public void update(Date date) {
                tics.add(date);
                if (++count == seconds)
                    synchronized(monitor) { // 2
                        monitor.notifyAll(); // 3
                    }
            }
        };
        clock = new Clock(listener);
        synchronized(monitor) { // 4
            monitor.wait(); // 5
        }
        clock.stop();
        verify(tics, seconds);
    }

    private void verify(List<Date> tics, int seconds) {
        assertEquals(seconds, tics.size());
        for (int i = 1; i < seconds; i++)
            assertEquals(1, getSecondsFromLast(tics, i));
    }

    private long getSecondsFromLast(List<Date> tics, int i) {
        Calendar calendar = new GregorianCalendar();
        calendar.setTime(tics.get(i));
        int now = calendar.get(Calendar.SECOND);
        calendar.setTime(tics.get(i - 1));
        int then = calendar.get(Calendar.SECOND);
        if (now == 0)
            now = 60;
        return now - then;
    }
}

```

After the test creates the `Clock` instance, it waits (line 5). You can send the `wait` message to *any* object—`wait` is defined in the class `Object`. You must first obtain a lock on that same object. You do so using either a `synchronized` block (line 4) or `synchronized` method, as mentioned above. In the test, you use an arbitrary object stored in the `monitor` field (line 1) as the object to lock and wait on.

What does the test wait for? In the mock listener implementation, you can store a count variable that gets bumped up by 1 each time the `update(Date)` method is called. Once the desired number of messages is received, you can tell the `ClockTest` to stop waiting.

From an implementation standpoint, the test thread waits until the listener, which is executing in another thread, says to proceed. The listener says that things can proceed by sending the `notifyAll` message to *the same object the test is waiting on* (line 3). In order to call `notifyAll`, you must first obtain a lock against the monitor object, again using a synchronized block.

But . . . wait! The call to `wait` was in a synchronized block, meaning that no other code can obtain a lock—including the synchronized block wrapping the `notifyAll` method—until the synchronized block exits. Yet it won’t exit until the waiting is done, and the waiting isn’t done until `notifyAll` is called. This would seem to be a Catch-22 situation.

The trick: Behind the scenes, the `wait` method puts the current thread on what is called a “wait set” for the monitor object. It then *releases all locks* before idling the current thread. Thus when code in the other thread encounters the synchronized block wrapping the `notifyAll` call, there is no lock on the monitor. It can then obtain a lock (line 2). The `notifyAll` call requires the lock in order to be able to send its message to the monitor.

You may want to go over the discussion of wait/notify a few times until it sinks in.

Here is the `ClockListener` interface and the `Clock` class implementation:

```
// sis.clock.ClockListener
package sis.clock;

import java.util.*;

public interface ClockListener {
    public void update(Date date);
}

// sis.clock.Clock
package sis.clock;

import java.util.*;

public class Clock implements Runnable {
    private ClockListener listener;
    private boolean run = true;

    public Clock(ClockListener listener) {
        this.listener = listener;
        new Thread(this).start();
    }
}
```

Wait/Notify

```

    }

    public void stop() {
        run = false;
    }

    public void run() {
        while (run) {
            try { Thread.sleep(1000); }
            catch (InterruptedException e) {}
            listener.update(new Date());
        }
    }
}

```

The `run` method in `Clock` uses the simple technique of testing a boolean flag, `run`, each time through a `while` loop to determine when to stop. The `stop` method, called by the test after its waiting is complete, sets the `run` boolean to `false`.

There is a flaw in the `Clock` implementation. The `run` method sleeps for *at least* a full second. As I noted earlier, a `sleep` call may take a few additional nanoseconds or milliseconds. Plus, creating a new `Date` object takes time. This means that it's fairly likely that the clock could skip a second: If the last time posted was at 11:55:01.999, for example, and the `sleep` plus additional execution took 1,001 milliseconds instead of precisely 1,000 milliseconds, then the new time posted would be 11:55:03.000. The clock UI would show 11:55:01, then skip to 11:55:03. In an analog display, you'd see the second hand jump a little bit more—not that big a deal in most applications. If you run enough iterations in the test, it will almost certainly fail.

Here's an improved implementation:

```

public void run() {
    long lastTime = System.currentTimeMillis();
    while (run) {
        try { Thread.sleep(10); }
        catch (InterruptedException e) {}
        long now = System.currentTimeMillis();
        if ((now / 1000) - (lastTime / 1000) >= 1) {
            listener.update(new Date(now));
            lastTime = now;
        }
    }
}

```

Could you have written a test that would have uncovered this defect consistently? One option for doing so might include creating a utility class to return the current milliseconds. You could then mock the utility class to force a certain time.

Wait/Notify

The test for Clock is a nuisance. It adds 5 seconds to the execution time of your unit tests. How might you minimize this time? First, it probably doesn't matter if the test proves 5 tics or 2. Second, you might modify the clock class (and test accordingly) to support configurable intervals. Instead of 1 second, write the test to demonstrate support for hundredths of a second (like a stopwatch would require).

Additional Notes on wait and notify

The Object class overloads the `wait` method with versions that allow you to specify a timeout period. A thread you suspend using one of these versions of `wait` idles until another thread notifies it or until the timeout period has expired.

Under certain circumstances, it is possible for a *spurious wakeup* to occur—one that you did not trigger by an explicit notification! This is rare, but if you use `wait` and `notify` in production code, you need to guard against this condition. To do so, you can enclose the `wait` statement in a `while` loop that tests a condition indicating whether or not execution can proceed. The Java API documentation for the `wait` method shows how you might implement this.

In a test, you could create a `do-while` loop that called the `verify` method each time through. If the `verify` method were to then fail, you would execute `wait` again. I view this as an unnecessary complexity for a test. The worst case of not guarding against the unlikely spurious wakeup would be that the test would fail, at which time you could rerun it.

While `notifyAll` wakes up *all* threads that are waiting if there is more than one, the `notify` method chooses an arbitrary thread to be woken up. In most circumstances you will want to use `notifyAll`. However, in some situations you will want to wake up only a single thread. An example is a *thread pool*.

The search server currently uses a single thread to process incoming requests. Clients eventually get the results of their search in an asynchronous fashion. While this may be fine for a small number of requests, clients could end up waiting quite a while if a number of other clients have requested searches ahead of them.

You might consider spawning each search request off in a new, separate thread as it arrives. However, each Java thread you create and start carries with it significant overhead costs. Creating one thread for each of thousands of search requests would result in severe performance problems. The Java

Additional
Notes on wait
and notify

thread scheduler would spend more time context-switching between threads than it would processing each thread.

A thread pool collects an arbitrary, smaller number of worker Thread objects that it creates and starts running. As a search comes in, you pass it off to the pool. The pool adds the work to the end of a queue, then issues a `notify` message. Any worker thread that has completed its work checks the queue for any pending tasks. If no tasks are available, the worker thread sits in an idle state by blocking on a `wait` call. If there are idle threads, the `notify` message will wake up one of them. The awakened worker thread can then grab and process the next available piece of work.

Locks and Conditions

Locks and Conditions

J2SE 5.0 introduces a new, more flexible mechanism for obtaining locks on objects. The interface `java.util.concurrent.locks.Lock` allows you to associate one or more conditions with a lock. Different types of locks exist. For example, the `ReadWriteLock` implementation allows one thread to write to a shared resource, while allowing multiple threads to read from the same resource. You can also add fairness rules to a reentrant lock to do things like allow threads waiting the longest to obtain the lock first. Refer to the API documentation for details.

The listing of `ClockTest` shows how you would replace the use of synchronization with the new lock idiom.

```
package sis.clock;

import java.util.*;
import java.util.concurrent.locks.*;

import junit.framework.*;

public class ClockTest extends TestCase {
    private Clock clock;
    private Lock lock;
    private Condition receivedEnoughTics;

    protected void setUp() {
        lock = new ReentrantLock();
        receivedEnoughTics = lock.newCondition();
    }

    public void testClock() throws Exception {
        final int seconds = 2;
        final List<Date> tics = new ArrayList<Date>();
        ClockListener listener = createClockListener(tics, seconds);
    }
}
```

```

clock = new Clock(listener);
lock.lock();
try {
    receivedEnoughTics.await();
}
finally {
    lock.unlock();
}
clock.stop();
verify(tics, seconds);
}

private ClockListener createClockListener(
    final List<Date> tics, final int seconds) {
    return new ClockListener() {
        private int count = 0;
        public void update(Date date) {
            tics.add(date);
            if (++count == seconds) {
                lock.lock();
                try {
                    receivedEnoughTics.signalAll();
                }
                finally {
                    lock.unlock();
                }
            }
        }
    };
}
...
}

```

Locks and Conditions

The modified and slightly refactored implementation of `ClockTest` uses a `ReentrantLock` implementation of the `Lock` interface. A thread interested in accessing a shared resource sends the message `lock` to a `ReentrantLock` object. Once the thread obtains the lock, other threads attempt to lock the block until the first thread releases the lock. A thread releases the lock by sending `unlock` to a `Lock` object. You should always ensure an unlock occurs by using a `try-finally` block.

The shared resource in `ClockTest` is the `Condition` object. You obtain a `Condition` object from a `Lock` by sending it the message `newCondition` (see the `setUp` method). Once you have a `Condition` object, you can block on it by using `await`. As with the `wait` method, code in `await` releases the lock and suspends the current thread.¹² Code in another thread signals that the condition has been met by sending the `Condition` object the message `signal` or `signalAll`.

¹²Which suggests you may not need to enclose an `await` call in a `try-finally` block. Don't risk it—you can't guarantee that something won't go awry with the code in `await`.

Coupled with the use of Condition objects, the idiom shown in ClockTest effectively replaces the classic wait/notify scheme.

Thread Priorities

Threads can specify different priority levels. The scheduler uses a thread's priority as a suggestion for how often the thread should get attention. The main thread that executes has a default priority assigned to it, Thread.NORM_PRIORITY. A priority is an integer, bounded by Thread.MIN_PRIORITY on the low end and Thread.MAX_PRIORITY on the high end.

Thread Priorities

You might use a lower priority for a thread that executes in the background. In contrast, you might use a higher priority for user interface code that needs to be extremely responsive. Generally, you will want to deviate from NORM_PRIORITY only by small amounts, such as +1 and -1. Deviating by large amounts can create applications that perform poorly, with some threads dominating and others starving for attention.

Every thread (other than the main thread) starts with the same priority as the thread that spawned it. You can send a thread the message setPriority to specify a different priority.

The following snippet of code from the Clock class shows how you might set the priority for the clock to a slightly lower value.

```
public void run() {
    Thread.currentThread().setPriority(Thread.NORM_PRIORITY - 1);
    long lastTime = System.currentTimeMillis();
    while (run) {
        try { Thread.sleep(10); }
        catch (InterruptedException e) {}
        long now = System.currentTimeMillis();
        if ((now / 1000) - (lastTime / 1000) >= 1) {
            listener.update(new Date(now));
            lastTime = now;
        }
    }
}
```

Note use of the currentThread static method, which you can always call to obtain the Thread object that represents the thread that is currently executing.

Deadlocks

If you are not careful in coding multithreaded applications, you can encounter deadlocks, which will bring your system to a screeching halt. Suppose you have an object *alpha* running in one thread and an object *beta* running in another thread. *Alpha* executes a synchronized method (which by definition holds a lock on *alpha*) that calls a synchronized method defined on *beta*. If at the same time *beta* is executing a synchronized method, and this method in turn calls a synchronized method on *alpha*, each thread will wait for the other thread to relinquish its lock before proceeding. Deadlock!¹³

Solutions for resolving deadlock:

1. Order the objects to be locked and ensure the locks are acquired in this order.
2. Use a common object to lock.

ThreadLocal

You may have a need to store separate information along with each thread that executes. Java provides a class named ThreadLocal that manages creating and accessing a separate instance, per thread, of any type.

When you interact with a database through JDBC,¹⁴ you obtain a Connection object that manages communication between your application and the database. It is not safe for more than one thread to work with a single connection object, however. One classic use of ThreadLocal is to allow each thread to contain a separate Connection object.

However, you've not learned about JDBC yet. For your example, I'll define an alternate use for ThreadLocal.

In this example, you'll take the existing Server class and add logging capabilities to it. You want each thread to track when the search started and when it completed. You also want the start and stop message pairs for a search to appear together in the log. You could store a message for each event (start and stop) in a common thread-safe list stored in Server. But if several threads were to add messages to this common list, the messages would interleave. You'd probably see all the start log events first, then all the stop log events.

¹³[Arnold2000].

¹⁴The Java DataBase Connectivity API. See Additional Lesson III for further information.

ThreadLocal

To solve this problem, you can have each thread store its own log messages in a `ThreadLocal` variable. When the thread completes, you can obtain a lock and add all the messages at once to the complete log.

First, modify the `Server` class to create a new thread for each `Search` request. This should boost the performance in most environments, but you do want to be careful how many threads you let execute simultaneously. (You might experiment with the number of searches the test kicks off to see what the limits are for your environment.) You may want to consider a thread pool as mentioned in the section Additional Notes on `wait` and `notify`.

```
private void execute(final Search search) {
    new Thread(new Runnable() {
        public void run() {
            search.execute();
            listener.executed(search);
        }
    }).start();
}
```

Thread/Local

Ensure that your tests still run successfully. Next, let's refactor `ServerTest`—significantly. You want to add a new test that verifies the logged messages. The current code is a bit of a mess, and it has a fairly long and involved test. For the second test, you will want to reuse most of the nonassertion code from `testSearch`. The refactored code also simplifies a bit of the performance testing.

Here is the refactored test that includes a new test, `testLogs`.

```
package sis.search;

import junit.framework.*;
import java.util.*;
import sis.util.*;

public class ServerTest extends TestCase {
    // ...
    public void testSearch() throws Exception {
        long start = System.currentTimeMillis();
        executeSearches();
        long elapsed = System.currentTimeMillis() - start;
        assertTrue(elapsed < 20);
        waitForResults();
    }

    public void testLogs() throws Exception {
        executeSearches();
        waitForResults();
        verifyLogs();
    }
}
```

```

private void executeSearches() throws Exception {
    for (String url: URLs)
        server.add(new Search(url, "xxx"));
}

private void waitForResults() {
    long start = System.currentTimeMillis();
    while (numberOfResults < URLs.length) {
        try { Thread.sleep(1); }
        catch (InterruptedException e) {}
        if (System.currentTimeMillis() - start > TIMEOUT)
            fail("timeout");
    }
}

private void verifyLogs() {
    List<String> list = server.getLog();
    assertEquals(URLS.length * 2, list.size());
    for (int i = 0; i < URLs.length; i += 2)
        verifySameSearch(list.get(i), list.get(i + 1));
}

private void verifySameSearch(
    String startSearchMsg, String endSearchMsg) {
    String startSearch = substring(startSearchMsg, Server.START_MSG);
    String endSearch = substring(endSearchMsg, Server.END_MSG);
    assertEquals(startSearch, endSearch);
}

private String substring(String string, String upTo) {
    int endIndex = string.indexOf(upTo);
    assertTrue("didn't find " + upTo + " in " + string,
               endIndex != -1);
    return string.substring(0, endIndex);
}
}

```

ThreadLocal

The method `testLogs` executes the searches, waits for them to complete, and verifies the logs. To verify the logs, the test requests the complete log from the `Server` object, then loops through the log, extracting a pair of lines at a time. It verifies that the search string (the first part of the log message) is the same in each pair of lines.¹⁵

The modifications to the `Server` class:

```

package sis.search;

import java.util.concurrent.*;
import java.util.*;

```

¹⁵See Lesson 8 for an in-depth discussion of Java logging and a discussion of whether or not it is necessary to test logging code.

```

public class Server extends Thread {
    private BlockingQueue<Search> queue =
        new LinkedBlockingQueue<Search>();
    private ResultsListener listener;
    static final String START_MSG = "started";
    static final String END_MSG = "finished";

    private static ThreadLocal<List<String>> threadLog =
        new ThreadLocal<List<String>>() {
            protected List<String> initialValue() {
                return new ArrayList<String>();
            }
        };
    private List<String> completeLog =
        Collections.synchronizedList(new ArrayList<String>());
    // ...
    public List<String> getLog() {
        return completeLog;
    }

    private void execute(final Search search) {
        new Thread(new Runnable() {
            public void run() {
                log(START_MSG, search);
                search.execute();
                log(END_MSG, search);
                listener.executed(search);
                completeLog.addAll(threadLog.get());
            }
        }).start();
    }

    private void log(String message, Search search) {
        threadLog.get().add(
            search + " " + message + " at " + new Date());
    }
    // ...
}

```

ThreadLocal

You declare a `ThreadLocal` object by binding it to the type of object you want stored in each thread. Here, `threadLog` is bound to a `List` of `String` objects. The `threadLog` `ThreadLocal` instance will internally manage a `List` of `String` objects for each separate thread.

You can't simply assign an initial value to `threadLog`, since each of the `List` objects it manages need to be initialized. Instead, `ThreadLocal` provides an `initialValue` method that you can override. The first time each thread gets its `ThreadLocal` instance via `threadLog`, code in the `ThreadLocal` class calls the `initialValue` method. Overriding this method provides you with the opportunity to consistently initialize the list.

ThreadLocal defines three additional methods: `get`, `set`, and `remove`. The `get` and `set` methods allow you to access the current thread's ThreadLocal instance. The `remove` method allows you to remove the current thread's ThreadLocal instance, perhaps to save on memory space.

In the search thread's `run` method, you call the `log` method before and after executing the search. In the `log` method, you access the current thread's copy of the log list by calling the `get` method on `threadLog`. You then add a pertinent log string to the list.

Once the search thread completes, you want to add the thread's log to the complete log. Since you've instantiated `completeLog` as a synchronized collection, you can send it the `addAll` method to ensure that all lines in the thread log are added as a whole. Without synchronization, another thread could add its log lines, resulting in an interleaved complete log.

The Timer Class

The Timer Class



You want to continually monitor a web site over time to see if it contains the search terms you specify. You're interested in checking every several minutes or perhaps every several seconds.

The test `testRepeatedSearch` in `SearchSchedulerTest` shows the intent of how this search monitor should work: Create a scheduler and pass to it a search plus an interval representing how often to run the search. The test mechanics involve waiting for a set period of time once the scheduler has started and then verifying that the expected number of searches are executed within that time.

```
package sis.search;

import junit.framework.*;
import sis.util.*;

public class SearchSchedulerTest extends TestCase {
    private int actualResultsCount = 0;

    protected void setUp() throws Exception {
        TestUtil.delete(SearchTest.FILE);
        LineWriter.write(SearchTest.FILE, SearchTest.TEST_HTML);
    }

    protected void tearDown() throws Exception {
        TestUtil.delete(SearchTest.FILE);
    }

    public void testRepeatedSearch() throws Exception {
        final int searchInterval = 3000;
    }
}
```

```

Search search = new Search(SearchTest.URL, "xxx");

ResultsListener listener = new ResultsListener() {
    public void executed(Search search) {
        ++actualResultsCount;
    }
};

SearchScheduler scheduler = new SearchScheduler(listener);
scheduler.repeat(search, searchInterval);

final int expectedResultsCount = 3;
Thread.sleep((expectedResultsCount - 1) * searchInterval + 1000);

scheduler.stop();
assertEquals(expectedResultsCount, actualResultsCount);
}
}

```

The Timer Class

The `SearchScheduler` class, below, uses the `java.util.Timer` class as the basis for scheduling and executing searches. Once you have created a `Timer` instance, you can call one of a handful of methods to schedule `TimerTasks`. A `TimerTask` is an abstract class that implements the `Runnable` interface. You supply the details on what the task does by subclassing `TimerTask` and implementing the `run` method.

In `SearchScheduler`, you use the method `scheduleAtFixedRate`, which you pass a `TimerTask`, a delay before start of 0 milliseconds, and an interval (currently specified by the test) in milliseconds. Every *interval* milliseconds, the `Timer` object wakes up and sends the `run` message to the task. This is known as fixed-rate execution.

When you use fixed-rate execution,¹⁶ the `Timer` does its best to ensure that tasks are run every *interval* milliseconds. If a task takes longer than the interval, the `Timer` tries to execute tasks more tightly together to ensure that they finish before each interval is up. You can also use one of the delayed-rate execution methods, in which the case the interval from the end of one search to the next remains consistent.

You can cancel both timers and timer tasks via the `cancel` method.

```

package sis.search;

import java.util.*;

public class SearchScheduler {
    private ResultsListener listener;
    private Timer timer;

```

¹⁶[Sun2004b].

```

public SearchScheduler(ResultsListener listener) {
    this.listener = listener;
}

public void repeat(final Search search, long interval) {
    timer = new Timer();
    TimerTask task = new TimerTask() {
        public void run() {
            search.execute();
            listener.executed(search);
        }
    };
    timer.scheduleAtFixedRate(task, 0, interval);
}

public void stop() {
    timer.cancel();
}
}

```

Thread
Miscellany

Thread Miscellany

Atomic Variables and volatile

Java compilers try to optimize as much code as possible. For example, if you assign an initial value to a field outside of a loop:

```

private String prefix;
void show() throws Exception {
    prefix = ":";  

    for (int i = 0; i < 100; i++) {  

        System.out.print(prefix + i);  

        Thread.sleep(1);
    }
}

```

the Java compiler *may* figure that the value of the `prefix` field is fixed for the duration of the `show` method. It *may* optimize the code to treat the use of `prefix` within the loop as a constant. Java thus *may not* bother reading the value of `prefix` with each iteration through the loop. This is an appropriate assumption, as long as no other code (in a multithread execution) updates the value of `prefix` at the same time. But if another thread does modify `prefix`, code in the loop here may never see the changed value.¹⁷

¹⁷The actual behavior may depend on your compiler, JVM, and processor configuration.

You can force Java to always read a fresh value for a field by enclosing access to it in a `synchronized` block. Java uses this awareness that the code could be executing in multiple threads to ensure that it always gets the latest value for all fields contained within. Another way to get Java to read a fresh value is to declare the field as `volatile`. Doing so essentially tells the compiler that it cannot optimize access to the field.

When you use a shared `boolean` variable as the conditional for a thread's `run` loop, you may want to declare the `boolean` variable as `volatile`. This will ensure that the `while` loop reads the updated value of the `boolean` each time through the loop.

A related consideration is how Java treats common access to fields accessed from multiple threads. Java guarantees that reading or writing a variable—with the exception of a `long` or `double` variable—is *atomic*. This means that the smallest possible operation against the variable is to read or write its entire value. It is not possible for one thread to write a portion of the variable while another thread is writing another portion, which would corrupt the variable.

Thread Information

As mentioned earlier, you can obtain the `Thread` object for the currently executing thread by using the `Thread` static method `currentThread`. The `Thread` object contains a number of useful methods for obtaining information about the thread. Refer to the Java API documentation for a list of these getter and query methods. Additionally, using `setName` and `getName`, you can set and retrieve an arbitrary name for the thread.

Shutting Down

The main thread in an executing application is by default a *user thread*. As long as there is at least one active user thread, your application will continue to execute. In contrast, you can explicitly designate a thread as a *daemon thread*. If all user threads have terminated, the application will terminate as well, even if daemon threads continue to execute.

The `ThreadTest` example demonstrates this behavior with respect to user threads. The code executing in `main` runs in a user thread. Any threads spawned inherit the execution mode (user, or daemon) of the thread that spawned them, so thread `t` in the example is a user thread.

```
public class ThreadTest {
    public static void main(String[] args) throws Exception {
        Thread t = new Thread() {
```

```

        public void run() {
            while (true)
                System.out.print('.');
            }
        };
        t.start();
        Thread.sleep(100);
    }
}

```

When you execute ThreadTest, it will not stop until you force termination of the process (Ctrl-c usually works from the command line).

Setting a Thread object to execute as a daemon thread is as simple as sending it the message `setDaemon` with a parameter value of true. You must set the thread execution mode prior to starting the thread. Once a thread has started, you cannot change its execution mode.

```

public class ThreadTest {
    public static void main(String[] args) throws Exception {
        Thread t = new Thread() {
            public void run() {
                while (true)
                    System.out.print('.');
            }
        };
        t.setDaemon(true);
        t.start();
        Thread.sleep(100);
    }
}

```

**Thread
Miscellany**

You can forcibly exit an application, regardless of whether any user threads remain alive or not, by calling the `System` static method `exit`. (The `Runtime` class has an equivalent method.)

```

public class ThreadTest {
    public static void main(String[] args) throws Exception {
        Thread t = new Thread() {
            public void run() {
                while (true)
                    System.out.print('.');
            }
        };
        t.start();
        Thread.sleep(100);
        Runtime.getRuntime().exit(0);
        //System.exit(0); // this will also work
    }
}

```

The exit method requires you to pass it an int value. This value represents the application return code, which you can use to control shell scripts or batch files.

In many applications, such as Swing applications, you will not have control over other threads that are spawned. Swing itself executes code in threads. More often than not, threads end up being initiated as user threads. Accordingly, you may need to use the exit method to terminate a Swing application.

Managing Exceptions

When a run method throws an (unchecked¹⁸) exception, the thread in which it is running terminates. Additionally, the exception object itself disappears. You may wish to capture the exception, as ServerTest.testException suggests.

Thread Miscellany

```
public void testException() throws Exception {
    final String errorMessage = "problem";
    Search faultySearch = new Search(URLS[0], "") {
        public void execute() {
            throw new RuntimeException(errorMessage);
        }
    };
    server.add(faultySearch);
    waitForResults(1);
    List<String> log = server.getLog();
    assertTrue(log.get(0).indexOf(errorMessage) != -1);
}

private void waitForResults() {
    waitForResults(URLS.length);
}

private void waitForResults(int count) {
    long start = System.currentTimeMillis();
    while (numberOfResults < count) {
        try { Thread.sleep(1); }
        catch (InterruptedException e) {}
        if (System.currentTimeMillis() - start > TIMEOUT)
            fail("timeout");
    }
}
```

The test uses a mock override of the Search execute method to simulate a RuntimeException being thrown. Your expectation is that this exception will

¹⁸Since the Runnable run method signature declares no exceptions, you cannot override it to throw any checked exceptions.

be logged to the Server as a failed search. Here is the Server class implementation:

```
private void execute(final Search search) {
    Thread thread = new Thread(new Runnable() {
        public void run() {
            log(START_MSG, search);
            search.execute();
            log(END_MSG, search);
            listener.executed(search);
            completeLog.addAll(threadLog.get());
        }
    });
    thread.setUncaughtExceptionHandler(
        new Thread.UncaughtExceptionHandler() {
            public void uncaughtException(Thread th, Throwable thrown) {
                completeLog.add(search + " " + thrown.getMessage());
                listener.executed(search);
            }
        }
    );
    thread.start();
}
```

Thread
Miscellany

After creating the thread object, but before starting it, you can set an *uncaught exception handler* on the thread. You create an uncaught exception handler by implementing the method `uncaughtException`. If the thread throws an exception that is not caught, the `uncaughtException` method is called. Java passes `uncaughtException` both the thread and throwable objects.

Thread Groups

Thread groups provide a way to organize threads into arbitrary groups. A thread group can also contain other thread groups, allowing for a containment hierarchy. You can control all threads contained in a thread group as a unit. For example, you can send the message `interrupt` to a thread group, which will in turn send the `interrupt` message to all threads contained within the hierarchy.

In *Effective Java*, Joshua Bloch writes that thread groups “are best viewed as an unsuccessful experiment, and you may simply ignore their existence.”¹⁹ They were originally designed to facilitate security management in Java but never satisfactorily fulfilled this purpose.

Prior to J2SE 5.0, one useful purpose for thread groups was to monitor threads for uncaught exceptions. However, you can now fulfill this need by

¹⁹[Bloch 2001].

creating an `UncaughtExceptionHandler` for a thread (see the previous section, Managing Exceptions).

Atomic Wrappers

Even simple arithmetic operations are not atomic. As of J2SE 5.0, Sun supplies the package `java.util.concurrent.atomic`. This package includes a dozen Atomic wrapper classes. Each class wraps a primitive type (`int`, `long`, or `boolean`), a reference, or an array. By using an atomic wrapper, you guarantee that operations on the wrapped value are thread safe and work against an up-to-date value (as if you had declared the field `volatile`).

Here is an example use:

```
AtomicInteger i = new AtomicInteger(50);
assertEquals(55, i.addAndGet(5));
assertEquals(55, i.get());
```

Exercises

Refer to the Java API documentation for further details.

Summary: Basic Design Principles for Synchronization

- Avoid it. Introduce synchronization only when necessary.
- Isolate it. Cluster synchronization needs into a single-responsibility class that does as little as possible. Ensure that as few clients as possible interact with a synchronized class. Ensure that as few methods as possible interact with shared data.
- Shared, or “server,” classes should provide synchronization instead of depending on clients to synchronize properly.
- If a server class has not provided synchronization, consider writing a “wrapper” class that adds synchronization and delegates all messages to the server class.
- Use the synchronization class library located in `java.util.concurrent`.

Exercises

1. Create an `AlarmClock` class. A client can submit a single event and corresponding time. When the time passes, the `AlarmClock` sends an alarm to the client.

2. Eliminate the wait loop in `AlarmClockTest` (from the previous exercise) by using `wait` and `notify`.
3. Modify the alarm clock to support multiple alarms (if it doesn't already). Then add the ability to cancel an alarm by name. Write a test that demonstrates the ability to cancel the alarm. Then analyze your code and note the potential for synchronization issues. Introduce a pause in the production code to force the issue and get your test to fail. Finally, introduce synchronization blocks or modify methods to be synchronized as necessary to fix the problem.
4. Change the `run` method in `AlarmClock` to use a timer to check for alarms every half-second.

Exercises

This page intentionally left blank

Lesson 14

Generics

I introduced you to parameterized types in Lesson 2. This lesson covers parameterized types, or generics, in depth. You will learn the many rules behind creating parameterized types. Creating parameterized types is probably tied with multithreading as the most complex topic in core Java programming.

You will learn about:

- creating parameterized types
- multiple type parameters
- the erasure scheme
- upper bounds (extends)
- wildcards
- generic methods
- lower bounds (super)
- additional bounds
- raw types
- checked collections

Parameterized
Types

Parameterized Types

You generally develop a collection so that it can contain objects of any type. However, in most cases, collections are useful only if you constrain them to hold objects of a single (set of) types: a list of students, a words-to-definitions map, and so on. You could consider this to be the Single-Responsibility

Principle applied to collections: Collections should hold one kind of thing and one kind of thing only. You rarely want your student list to contain a stray professor. And if you included a word-to-picture entry in your map, it would create significant headaches for your dictionary application.

Sun originally developed the base collection classes in Java's class library to support storing and returning objects of any type (i.e., Object or any subclass). This meant that you could add a String object to a collection intended to store Student objects. While this sounds like an unlikely mistake, pulling an object of an unexpected type out of a collection is a frequent source of application defects. The reason is that code that stores objects into the collection is often distant from code that retrieves objects from the collection.

With the advent of J2SE 5.0, Sun introduced the concept of *parameterized types*, also known as generics. You can now associate, or *bind*, a collection instance to a specific type. You can specify that an ArrayList can only hold Student objects. Under earlier Java versions, you would receive a ClassCastException error at runtime when retrieving a String inadvertently stored in the ArrayList. Now, with parameterized types, you receive a compilation error at the point where code attempts to insert a String in the list.

Collection Framework

Collection Framework

Sun has parameterized all collection classes in the Collections Framework. A simple language-based test demonstrates a simple use of the parameterized type ArrayList:

```
final String name = "joe";
List<String> names = new ArrayList<String>(); // 1
names.add(name); // 2
String retrievedName = names.get(0); // 3
assertEquals(name, retrievedName);
```

Both the implementation class ArrayList and the interface List are bound to the String class (line 1). You can add String objects to name (2). When retrieving from name and assigning to a String reference (3), you need not cast to String.

If you attempt to insert an object of any other type:

```
names.add(new Date()); // this won't compile!
```

the compiler will present you with an error:

```
cannot find symbol
symbol : method add(java.util.Date)
```

```
location: interface java.util.List<java.lang.String>
    names.add(new Date()); // this won't compile!
    ^

```

Multiple Type Parameters

If you look at the API documentation for `java.util.List` and `java.util.ArrayList`, you will see that they are declared as `List<E>` and `ArrayList<E>`, respectively. The `<E>` is known as the *type parameter list*. The `List` interface and `ArrayList` class declarations each contain a single type parameter in the type parameter list. Therefore you must supply a single bind type when using `List` or `ArrayList` in your code.¹

The class `HashMap` represents a collection of key-value pairs. A key can be of one type and the value could be another. For example, you might store an appointment book in a `HashMap`, where the key is a `Date` and the value is a `String` description of an event occurring on that date.

```
final String event = "today";
final Date date = new Date();
Map<Date, String> events = new HashMap<Date, String>();
events.put(date, event);
String retrievedEvent = events.get(date);
assertEquals(event, retrievedEvent);
```

**Creating
Parameterized
Types**

The `Map` interface and `HashMap` class are declared as `Map<K,V>` and `HashMap<K,V>`, respectively. You must supply two bind types when using these—one for the key (`K`) and one for the value (`V`).

Creating Parameterized Types

You will develop a parameterized `MultiHashMap`. A `MultiHashMap` is similar to a `HashMap`, except that it allows you to associate multiple values with a given key. Let's expand on the calendar example: Some people lead particularly interesting lives, in which two events might occur on one day.

A few starter tests will get you off the ground. Remember that you should be developing incrementally, one test method and one assertion at a time.

¹You are not required to supply any bind types; however, this is not recommended. See the section in this lesson entitled Raw Types.

```

package sis.util;

import junit.framework.*;
import java.util.*;

public class MultiHashMapTest extends TestCase {
    private static final Date today = new Date();
    private static final Date tomorrow =
        new Date(today.getTime() + 86400000);
    private static final String eventA = "wake up";
    private static final String eventB = "eat";

    private MultiHashMap<Date, String> events;
    protected void setUp() {
        events = new MultiHashMap<Date, String>();
    }

    public void testCreate() {
        assertEquals(0, events.size());
    }

    public void testSingleEntry() {
        events.put(today, eventA);
        assertEquals(1, events.size());
        assertEquals(eventA, getSoleEvent(today));
    }

    public void testMultipleEntriesDifferentKey() {
        events.put(today, eventA);
        events.put(tomorrow, eventB);
        assertEquals(2, events.size());
        assertEquals(eventA, getSoleEvent(today));
        assertEquals(eventB, getSoleEvent(tomorrow));
    }

    public void testMultipleEntriesSameKey() {
        events.put(today, eventA);
        events.put(today, eventB);
        assertEquals(1, events.size());
        Collection<String> retrievedEvents = events.get(today);
        assertEquals(2, retrievedEvents.size());
        assertTrue(retrievedEvents.contains(eventA));
        assertTrue(retrievedEvents.contains(eventB));
    }

    private String getSoleEvent(Date date) {
        Collection<String> retrievedEvents = events.get(date);
        assertEquals(1, retrievedEvents.size());
        Iterator<String> it = retrievedEvents.iterator();
        return it.next();
    }
}

```

Creating Parameterized Types

The method `getSoleEvent` is of some interest. After retrieving the events collection stored at a date using the `Map` method `get`, you must ensure that it contains only one element. To retrieve the sole element, you can create an iterator and return the first element it points to. You must bind the `Iterator` object to the same type that you bound the collection to (`String` in this example).

From an implementation standpoint, there is more than one way to build a `MultiHashMap`. The easiest is to use a `HashMap` where each key is associated with a collection of values. For this example, you will define `MultiHashMap` to encapsulate and use a `HashMap`.

In order to support the existing tests, the implementation of `MultiHashMap` is simplistic. Each method `size`, `put`, and `get` will delegate to the encapsulated `HashMap`:

```
package sis.util;

import java.util.*;

public class MultiHashMap<K,V> {
    private Map<K,List<V>> map = new HashMap<K,List<V>>();

    public int size() {
        return map.size();
    }

    public void put(K key, V value) {
        List<V> values = map.get(key);
        if (values == null) {
            values = new ArrayList<V>();
            map.put(key, values);
        }
        values.add(value);
    }

    public List<V> get(K key) {
        return map.get(key);
    }
}
```

**Creating
Parameterized
Types**

The `put` method first extracts a list from `map` using the key passed in. If there is no entry at the key, the method constructs a new `ArrayList` bound to the value type `V` and puts this list into `map`. Regardless, `value` is added to the list.

Like `HashMap`, the type parameter list contains two type parameters, `K` and `V`. Throughout the definition for `MultiHashMap`, you will see these symbols where you might expect to see type names. For example, the `get` method returns `V` instead of, say, `Object`. Each use of a type parameter symbol within the type declaration is known as a *naked type variable*.

At compile time, each occurrence of a naked type variable is replaced with the appropriate type of the corresponding type parameter. You'll see how this actually works in the next section, Erasure.

In the definition of the `map` field (highlighted in bold), you construct a `HashMap` object and bind it to `<K,List<V>>`. The key for `map` is the same type and the one to which the key of `MultiHashMap` is bound (`K`). The value for `map` is a `List` bound to type `V`—the value type to which the `MultiHashMap` is bound.

Bind parameters correspond to type parameters. The test contains this instantiation of `MultiHashMap` in its `setUp` method:

```
events = new MultiHashMap<Date, String>();
```

The `Date` type corresponds to the type parameter `K`, and the `String` type corresponds to the type parameter `V`. Thus, the embedded `map` field would be a `HashMap<Date, List<String>>`—a hash map whose key is bound to a date and whose value is bound to a list of strings.

Erasure

Erasure

There is more than one way that Sun might have chosen to implement support for parameterized types. One possible way would have been to create a brand-new type definition for each type to which you bind a parameterized class. In binding to a type, each occurrence of a naked type variable in the source would be replaced with the bind type. This technique is used by C++.

For example, were Java to use this scheme, binding `MultiHashMap` to `<Date, String>` would result in the following code being created behind the scenes:

```
// THIS ISN'T HOW JAVA TRANSLATES GENERICS:
package sis.util;

import java.util.*;

public class MultiHashMap<Date, String> {
    private Map<Date, List<String>> map =
        new HashMap<Date, List<String>>();

    public int size() {
        return map.size();
    }

    public void put(Date key, String value) {
        List<String> values = map.get(key);
        if (values == null)
            values = new ArrayList<String>();
        values.add(value);
        map.put(key, values);
    }
}
```

```

        if (values == null) {
            values = new ArrayList<String>();
            map.put(key, values);
        }
        values.add(value);
    }

    public List<String> get(Date key) {
        return map.get(key);
    }
}

```

Binding MultiHashMap to another pair of types, such as <String, String>, would result in the compiler creating another version of MultiHashMap. Using this scheme could potentially result in the compiler creating dozens of class variants for MultiHashMap.

Java uses a different scheme called *erasure*. Instead of creating separate type definitions, Java erases the parameterized type information to create a single equivalent type. Each type parameter is associated with a constraint known as its *upper bound*, which is java.lang.Object by default. Client bind information is erased and replaced with casts as appropriate. The MultiHashMap class would translate to:

```

package sis.util;

import java.util.*;

public class MultiHashMap {
    private Map map = new HashMap();

    public int size() {
        return map.size();
    }

    public void put(Object key, Object value) {
        List values = (List)map.get(key);
        if (values == null) {
            values = new ArrayList();
            map.put(key, values);
        }
        values.add(value);
    }

    public List get(Object key) {
        return (List)map.get(key);
    }
}

```

Erasure

Knowing how parameterized types work behind the scenes is critical to being able to understand and use them effectively. Java contains a number of

restrictions on the use of parameterized types that exist because of the erasure scheme. I'll discuss these limitations throughout this chapter. Each possible scheme for implementing generics has its downsides. Sun chose the erasure scheme for its ability to provide ultimate backward compatibility.

Upper Bounds

As mentioned, every type parameter has a default upper bound of Object. You can constrain a type parameter to a different upper bound. For example, you might want to supply an EventMap class, where the key must be bound to a Date type—either java.util.Date or java.sql.Date (which is a subclass of java.util.Date). A simple test:

```
package sis.util;

import junit.framework.*;
import java.util.*;

public class EventMapTest extends TestCase {
    public void testSingleElement() {
        EventMap<java.sql.Date, String> map =
            new EventMap<java.sql.Date, String>();
        final java.sql.Date date =
            new java.sql.Date(new java.util.Date().getTime());
        final String value = "abc";
        map.put(date, value);

        List<String> values = map.get(date);
        assertEquals(value, values.get(0));
    }
}
```

Upper Bounds

The EventMap class itself has no different behavior, only additional constraints on the type parameter K:

```
package sis.util;

public class EventMap<K extends java.util.Date, V>
    extends MultiHashMap<K, V> {
}
```

You use the `extends` keyword to specify the upper bound for a type parameter. In this example, the K type parameter for EventMap has an upper bound of `java.util.Date`. Code using an EventMap must bind the key to a `java.util.Date` or a subclass of `java.util.Date` (such as `java.sql.Date`). If you attempt to do otherwise:

```
EventMap<String, String> map = new EventMap<String, String>();
```

you will receive compile-time errors:

```
type parameter java.lang.String is not within its bound
EventMap<String, String> map = new EventMap<String, String>();
^
type parameter java.lang.String is not within its bound
EventMap<String, String> map = new EventMap<String, String>();
^
```

The compiler replaces naked type variables in generated code with the upper bound type. This gives you the ability to send more specific messages to naked type objects within the generic class.

You want the ability to extract all event descriptions from the EventMap for events where the date has passed. Code the following test in EventMapTest.

```
public void testGetPastEvents() {
    EventMap<Date, String> events = new EventMap<Date, String>();
    final Date today = new java.util.Date();
    final Date yesterday =
        new Date(today.getTime() - 86400000);
    events.put(today, "sleep");
    final String descriptionA = "birthday";
    final String descriptionB = "drink";
    events.put(yesterday, descriptionA);
    events.put(yesterday, descriptionB);
    List<String> descriptions = events.getPastEvents();
    assertTrue(descriptions.contains(descriptionA));
    assertTrue(descriptions.contains(descriptionB));
}
```

Upper Bounds

Within EventMap, you can presume that objects of type K are Date objects:

```
package sis.util;

import java.util.*;

public class EventMap<K extends Date, V>
    extends MultiHashMap<K, V> {
    public List<V> getPastEvents() {
        List<V> events = new ArrayList<V>();
        for (Map.Entry<K, List<V>> entry: entrySet()) {
            K date = entry.getKey();
            if (hasPassed(date))
                events.addAll(entry.getValue());
        }
        return events;
    }

    private boolean hasPassed(K date) {
        Calendar when = new GregorianCalendar();
```

```

        when.setTime(date);
        Calendar today = new GregorianCalendar();
        if (when.get(Calendar.YEAR) != today.get(Calendar.YEAR))
            return when.get(Calendar.YEAR) < today.get(Calendar.YEAR);
        return when.get(Calendar.DAY_OF_YEAR) <
            today.get(Calendar.DAY_OF_YEAR);
    }
}

```

In order to accomplish this, you'll have to add a method to Multi-HashMap to return the entrySet from the encapsulated map:

```

protected Set<Map.Entry<K,List<V>>> entrySet() {
    return map.entrySet();
}

```

The `entrySet` method returns a set bound to the `Map.Entry` type. Each `Map.Entry` object in the Set is in turn bound to the key type (`K`) and a list of the value type (`List<V>`).

Wildcards

Wildcards

Sometimes you'll write a method where you don't care about the type to which a parameter is bound. Suppose you need a utility method that creates a single string by concatenating elements in a list, separating the printable representation of each element with a new line. The `StringUtilTest` method `testConcatenateList` demonstrates this need:

```

package sis.util;

import junit.framework.*;
import java.util.*;

public class StringUtilTest extends TestCase {
    ...
    public void testConcatenateList() {
        List<String> list = new ArrayList<String>();
        list.add("a");
        list.add("b");

        String output = StringUtil.concatenate(list);

        assertEquals(String.format("a%nb%nb"), output);
    }
}

```

In the `StringUtil` method `concatenate`, you will append each list element's string representation to a `StringBuilder`. You can get the string representation of any object, provided by the `toString` method, without knowing or caring about its type. Thus, you want to be able to pass a `List` that is bound to any type as the argument to `concatenate`.

You might think that you can bind the `list` parameter to `Object`:

```
// this won't work
public static String concatenate(List<Object> list) {
    StringBuilder builder = new StringBuilder();
    for (Object element: list)
        builder.append(String.format("%s%n", element));
    return builder.toString();
}
```

By binding `list` to `Object`, you constrain it to hold objects only of type `Object`—and not of any `Object` subclasses. *You cannot assign a `List<String>` reference to a `List<Object>` reference.* If you could, client code could add an `Object` to `list` using the `List<Object>` reference. Code that then attempted to extract from `list` using the `List<String>` reference would unexpectedly retrieve an `Object`.

Instead, Java allows you to use a *wildcard* character (?) to represent any possible type:

```
package sis.util;

import java.util.*;

public class StringUtil {
    ...
    public static String concatenate(List<?> list) {
        StringBuilder builder = new StringBuilder();
        for (Object element: list)
            builder.append(String.format("%s%n", element));
        return builder.toString();
    }
}
```

Wildcards

Within the `concatenate` method body, you cannot use the `?` directly as a naked type variable. But since `list` can contain any type of object, you can assign each of its elements to an `Object` reference in the `for-each` loop.

Additionally, you can constrain a wildcard to an upper bound using the `extends` clause.



For a second string utility method, you need to be able to concatenate a list of numerics, whether they are `BigDecimal` objects or `Integer` objects. Several tests drive out the minor differences between decimal output and integral output:

```

public void testConcatenateFormattedDecimals() {
    List<BigDecimal> list = new ArrayList<BigDecimal>();
    list.add(new BigDecimal("3.1416"));
    list.add(new BigDecimal("-1.4142"));

    String output = StringUtil.concatenateNumbers(list, 3);
    assertEquals(String.format("3.14%n-1.414%n"), output);
}

public void testConcatenateFormattedIntegers() {
    List<Integer> list = new ArrayList<Integer>();
    list.add(12);
    list.add(17);

    String output = StringUtil.concatenateNumbers(list, 0);
    assertEquals(String.format("12%n17%n"), output);
}

```

The implementation in StringUtil:

```

public static String concatenateNumbers(
    List<? extends Number> list, int decimalPlaces) {
    String decimalFormat = "%" + decimalPlaces + "f";
    StringBuilder builder = new StringBuilder();
    for (Number element: list) {
        double value = element.doubleValue();
        builder.append(String.format(decimalFormat + "%n", value));
    }
    return builder.toString();
}

```

Implications of Using Wildcards

You'll need to import `java.math.*` to get the above code to compile.

The declaration of the `list` parameter in `concatenateNumbers` specifies that it is a `List` bound to either `Number` or a subclass of `Number`. The code in `concatenateNumbers` can then assign each element in `list` to a `Number` reference.

Implications of Using Wildcards

The downside of using a wildcard on a reference is that you cannot call methods on it that take an object of the type parameter. For example, suppose you create a `pad` utility method that can add an element n times to the end of a list.

```

package sis.util;

import java.util.*;
import junit.framework.*;

```

```
public class ListUtilTest extends TestCase {
    public void testPad() {
        final int count = 5;
        List<Date> list = new ArrayList<Date>();
        final Date element = new Date();
        ListUtil.pad(list, element, count);
        assertEquals(count, list.size());
        for (int i = 0; i < count; i++)
            assertEquals("unexpected element at " + i,
                         element, list.get(i));
    }
}
```

The `pad` method declares the `list` parameter to be a `List` that can be bound to any type:

```
package sis.util;

import java.util.*;

public class ListUtil {
    public static void pad(List<?> list, Object object, int count) {
        for (int i = 0; i < count; i++)
            list.add(object);
    }
}
```

Implications of Using Wildcards

The error you receive indicates that the compiler doesn't recognize an appropriate `add` method.

```
cannot find symbol
symbol  : method add(java.lang.Object)
location: interface java.util.List<?>
    list.add(object);
           ^
```

The problem is that the wildcard `?` designates an unknown type. Suppose that the `List` is bound to a `Date`:

```
List<Date> list = new ArrayList<Date>();
```

Then you attempt to pass a `String` to the `pad` method:

```
ListUtil.pad(list, "abc", count);
```

The `pad` method doesn't know anything about the specific type of the object being passed in, nor does it know anything about the type to which the list is bound. It can't guarantee that a client isn't trying to subvert the type safety of the list. Java just won't allow you to do this.

There is still a problem even if you specify an upper bound on the wildcard.

```
public static void pad(
    List<? extends Date> list, Date date, int count) {
    for (int i = 0; i < count; i++)
        list.add(date); // this won't compile
}
```

The problem is that a client could bind the list to `java.sql.Date`:

```
List<java.sql.Date> list = new ArrayList<java.sql.Date>();
```

Since `java.util.Date` is a superclass of `java.sql.Date`, you cannot add it to a list that is constrained to hold only objects of type `java.sql.Date` or a subclass of `java.sql.Date`. Due to erasure, Java has no way of figuring out whether or not a given operation is safe, so it must prohibit them all. The general rule of thumb is that you can use bounded wildcards only when reading from a data structure.

Wildcard Capture

Generic Methods

How do you solve the above problem? You can declare the `pad` method as a *generic method*. Just as you can specify type parameters for a class, you can specify type parameters that exist for the scope of a method:

```
public static <T> void pad(List<T> list, T object, int count) {
    for (int i = 0; i < count; i++)
        list.add(object);
}
```

The compiler can extract, or infer, the type for `T` based on the arguments passed to `pad`. It uses the most specific type that it can infer from the arguments. Your test should now pass.

Generic method type parameters can also have upper bounds.

You will need to use generic methods when you have a dependency between an argument to a method and either another argument or the return type. Otherwise, you should prefer the use of wildcards. In the `pad` method declaration, the `object` parameter's type is dependent upon the type of the `list` parameter.

Wildcard Capture

The technique of introducing a generic method to solve the above problem is known as wildcard capture. Another example is a simple method that swaps the elements of a list from front to back.

```

public void testWildcardCapture() {
    List<String> names = new ArrayList<String>();
    names.add("alpha");
    names.add("beta");
    inPlaceReverse(names);
    assertEquals("beta", names.get(0));
    assertEquals("alpha", names.get(1));
}

static void inPlaceReverse(List<?> list) {
    int size = list.size();
    for (int i = 0; i < size / 2; i++) {
        int opposite = size - 1 - i;
        Object temp = list.get(i);
        list.set(i, list.get(opposite));
        list.set(opposite, temp);
    }
}

```

The wildcard capture involves calling a new generic method, `swap`:

```

public void testWildcardCapture() {
    List<String> names = new ArrayList<String>();
    names.add("alpha");
    names.add("beta");
    inPlaceReverse(names);
    assertEquals("beta", names.get(0));
    assertEquals("alpha", names.get(1));
}

static void inPlaceReverse(List<?> list) {
    int size = list.size();
    for (int i = 0; i < size / 2; i++)
        swap(list, i, size - 1 - i);
}

private static <T> void swap(List<T> list, int i, int opposite) {
    T temp = list.get(i);
    list.set(i, list.get(opposite));
    list.set(opposite, temp);
}

```

Super

By doing so, you are giving the wildcard a name.

Super

Upper-bounded wildcards, which you specify using `extends`, are useful for *reading from* a data structure. You can support *writing to* a data structure using lower-bounded wildcards.



You want to be able to create a new MultiHashMap from an existing MultiHashMap. The new map is a subset of the existing map.

You obtain this subset by applying a filter to the values in the existing MultiHashMap. The test shown here creates a multimap of meetings. A meeting can be a one-time event or it can be recurrent. You want to create a new multimap that consists only of meetings that occur on Mondays.

Further, the original meetings multimap consists of java.sql.Date objects. Perhaps it was directly loaded from a database application. You want the new multimap to consist of java.util.Date objects.

```
public void testFilter() {
    MultiHashMap<String,java.sql.Date> meetings=
        new MultiHashMap<String,java.sql.Date>();

    meetings.put("iteration start", createSqlDate(2005, 9, 12));
    meetings.put("iteration start", createSqlDate(2005, 9, 26));
    meetings.put("VP blather", createSqlDate(2005, 9, 12));
    meetings.put("brown bags", createSqlDate(2005, 9, 14));

    MultiHashMap<String,java.util.Date> mondayMeetings =
        new MultiHashMap<String,java.util.Date>();
    MultiHashMap.filter(mondayMeetings, meetings,
        new MultiHashMap.Filter<java.util.Date>() {
            public boolean apply(java.util.Date date) {
                return isMonday(date);
            }
        });
}

assertEquals(2, mondayMeetings.size());
assertEquals(2, mondayMeetings.get("iteration start").size());
assertNull(mondayMeetings.get("brown bags"));
assertEquals(1, mondayMeetings.get("VP blather").size());
}

private boolean isMonday(Date date) {
    Calendar calendar = GregorianCalendar.getInstance();
    calendar.setTime(date);
    return calendar.get(Calendar.DAY_OF_WEEK) == Calendar.MONDAY;
}

private java.sql.Date createSqlDate(int year, int month, int day) {
    java.util.Date date = DateUtil.createDate(year, month, day);
    return new java.sql.Date(date.getTime());
}
```

Super

You can use lower-bounded wildcards on the MultiHashMap filter method to allow transforming from java.sql.Date values to java.util.Date values. In the implementation, the target argument of the filter method is a MultiHashMap whose value type (V) has a *lower bound* of V. This is expressed as ? super V. This means that the value type (V) of the target MultiHashMap

can be the same as V or a supertype of V. The meetings example works because java.util.Date is a supertype of java.sql.Date.

```
...
public class MultiHashMap <K,V> {
    private Map<K, List<V>> map = new HashMap<K, List<V>>();
    ...
    public interface Filter<T> {
        boolean apply(T item);
    }

    public static <K,V> void filter(
        final MultiHashMap<K, ? super V> target,
        final MultiHashMap<K, V> source,
        final Filter<? super V> filter) {
        for (K key : source.keys()) {
            final List<V> values = source.get(key);
            for (V value : values)
                if (filter.apply(value))
                    target.put(key, value);
        }
    }
}
```

Additional
Bounds

Additional Bounds

It is possible to specify additional bounds, or more than one type, in an extends clause. While the first constraint can be either a class or interface, subsequent constraints must be interface types. For example:

```
public static
    <T extends Iterable<T>> void iterateAndCompare(T t)
```

By using additional bounds, you are constraining the parameter passed to implement more than one interface. In the example, the object passed in must implement both the Iterable and Comparable interfaces. This is not normally something you want to do; additional bounds were introduced largely to solve issues of backward compatibility (see below).

If a method requires that an object behave as two different types, it intends for that method to do two different kinds of things to that object. In most cases, this will be a violation of the Single-Responsibility Principle, which is as applicable to methods as it is to classes. There will always be exceptions, of course, but before using additional bounds for this reason, see if you can decompose the method in question.

The more legitimate reason to use additional bounds is demonstrated by code in the Collections class. The Collections class contains static utility methods for operating on collection objects; it includes a method named `max` that returns the largest element in a collection. Under earlier versions of Java, the signature of `max` was:

```
public static Object max(Collection c)
```

The passed-in collection could hold any type of object, but the `max` method depended upon the collection holding objects that implemented the `Comparable` interface. An initial stab at a solution using generics could insist that the collection hold `Comparable` objects:

```
public static<T extends Comparable<? super T>> T max(Collection<? extends T> c)
```

Paraphrased, this signature says: `max` takes a collection of objects of any type and that that type must implement the `Comparable` interface, which in turn must be bound to the type or supertype of elements stored in the collection. The problem is that after erasure, this results in a different signature for the `max` method:

```
public static Comparable max(Collection c)
```

Instead of returning an `Object` reference, `max` would now return a `Comparable` reference. The change in signature would break existing compiled code that used the `max` method, killing compatibility. Using additional bounds effectively solves the problem:

```
public static<T extends Object&Comparable<? super T>> T max(Collection<? extends T> c)
```

Per J2SE specification, a type variable gets erased to its leftmost bound—`Object` in this example, and not `Comparable`. The `max` method now returns an `Object` reference but also defines the additional constraint that the collection objects must implement the appropriate `Comparable` interface.

Raw Types

Raw Types

If you work with an existing system, written in J2SE 1.4 or earlier, it will use collection classes that do not support parameterization. You'll have plenty of code such as:

```
List list = new ArrayList();
list.add("a");
```

You can still use this code under J2SE 5.0. When you use a generic type without binding it to a type parameter, you refer to the type as a *raw type*. The use of raw types is inherently not typesafe: You can add objects of any type to a raw collection and encounter runtime exceptions when retrieving an object of unexpected type. This is the fundamental reason why Sun introduced parameterized types.

Any such unsafe operation is met with a warning by the compiler. Using the default compiler options, the above two lines of code that add to a raw ArrayList result in the following compiler message:

Note: Some input files use unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

Do what the message says and recompile using the VM switch -Xlint:unchecked:

```
javac -source 1.5 -Xlint:unchecked *.java
```

Under Ant, supply an additional nested element on the javac task:

Raw Types

```
<target name="build" depends="init" description="build all">
  <javac
    srcdir="${src.dir}" destdir="${build.dir}"
    source="1.5"
    deprecation="on" debug="on" optimize="off" includes="**">
    <classpath refid="classpath" />
    <compilerarg value="-Xlint:unchecked"/>
  </javac>
</target>
```

Recompiling will present you with more specific details on the source of each unchecked warning:

```
warning: [unchecked] unchecked call to add(E) as a member of the raw type java.util.List
      list.add("a");
      ^
```

The Java compiler will generate an unchecked warning whenever it cannot guarantee the type safety of an operation on a parameterized type. If you are developing on an existing 1.4 or older application, you'll likely receive gobs of warnings. While it may not be possible or desirable to fix all the warnings at once, you should never treat warnings lightly. If you modify a section of legacy code, attempt to introduce generic types. You should be extremely leery of warnings that arise from any new code that you add.

Checked Collections

Suppose you're working with legacy code. In fact, most systems you encounter will include code written for older versions of Java. This is the reality of Java development. I still encounter systems that make pervasive use of Vector and Hashtable—even though Sun recommended use of collections framework classes (List/ArrayList, etc.) instead as of Java 1.2. That was over six years ago!

You will likely encounter raw collections for some time coming. You can quickly add some type safety to your code by using a checked wrapper. Suppose your existing mess of a system creates a list intended to hold Integer wrapper objects:

```
List ages = new ArrayList();
```

Elsewhere in the code, in a method nested a few classes away, another bozo developer adds a new line of code:

```
ages.add("17");
```

And even further away, code in another class is responsible for extracting age values from the collection:

```
int age = ((Integer)ages.get(0)).intValue();
```

Oops! That line of legacy code results in a ClassCastException, since the first element in ages is a String, not an Integer. Sure, you will find the problem “real soon now,” but you will have wasted valuable time in the process.

The best solution would be to parameterize references to ages in each of the three classes in question. Then the second developer wouldn’t even be able to compile the code that inserts a string. But changing those classes might be beyond your control or there might be dozens of affected classes and not three.²

Sun introduced a new set of methods to the Collections class that can help solve this problem with type safety. These methods create *checked wrapper* objects. A checked wrapper object delegates off to the actual collection object, much like the unmodifiable and synchronized wrappers. A checked wrapper ensures that an object passed to the collection is of the appropriate type, preventing inappropriate objects from being inserted into the collection.

Using a checked collection will at least constrain the exception to the point of error. In other words, the code that adds the bad data will generate

²And of course none of those dozens of classes will have tests. The vast majority of legacy systems have no tests.

Checked
Collections

the exception, not the code that attempts to extract it. This will make for quicker debugging efforts. You can make the change in one place:

```
List ages = Collections.checkedList(new ArrayList(), Integer.class);
```

When the Java VM executes this line of code, you receive an exception:

```
java.lang.ClassCastException: Attempt to insert class java.lang.String element into collection
with element type class java.lang.Integer
```

A language unit test demonstrates this in toto:

```
public void testCheckedCollections() {
    List ages =
        Collections.checkedList(new ArrayList(), Integer.class);
    try {
        ages.add("17");
        fail("expected ClassCastException on invalid insert");
    }
    catch (ClassCastException success) {
    }
}
```

Checked collections are not at all magical. The first argument to `checkedList` is the `ArrayList` object you are creating; it is bound to the `Integer` type. The second argument is a `Class` reference to the `Integer` type. Each time you invoke the `add` method, it uses the class reference to determine whether or not the passed parameter is of that type. If not, it throws a `ClassCastException`.

Checked wrappers require the redundancy of having to pass in a type (`Integer.class`) reference in addition to specifying a bind type (`<Integer>`). This is required because of erasure: the bind type information is not available to the list object at runtime. You can encapsulate unchecked wrappers in your own parameterized types, but you will then need to require clients to pass in the `Class` reference.

The `Collection` class provides checked collection wrappers for the types `Collection`, `List`, `Map`, `Set`, `SortedMap`, and `SortedSet`.

Even if you are writing J2SE 5.0 code, using checked collections can protect you from loose and fast developers beyond your sphere of influence. Some cowboy and cowgirl developers enjoy subverting the rules whenever they can. In the case of parameterized types, Java lets them.

You can cast an object of a parameterized type to a raw type. This allows you to store an object of any type in the collection. You will receive a compilation warning, but you can ignore the warning. Do so at your own peril. The results will generally be disastrous. This sort of strategy falls under the category of “don’t do that!”

**Checked
Collections**

Using checked collections in 5.0 code can help solve this problem. The exceptions thrown will come from the sneaky code, letting you pinpoint the source of trouble.

Arrays

You cannot create arrays of bounded parameterized types:

```
List<String>[] names = new List<String>[100]; // this does not compile
```

Attempting to do so generates the compilation error:

```
arrays of generic types are not allowed
```

Again, the problem is that you could easily assign the parameterized type reference to another type reference. You could then add an inappropriate object, causing a `ClassCastException` upon attempted extraction:

```
// this does not compile
List<String>[] namesTable = new List<String>[100];
Object[] objects = (Object[])namesTable;
List<Integer> numbers = new ArrayList<Integer>();
numbers.add(5);
objects[0] = numbers;
String name = namesTable[0].get(0); // would be a ClassCastException
```

Java does allow you to create arrays of parameterized types that are unbounded (i.e., where you use only the wildcard character `?`):

```
public void testArrays() {
    List<?>[] namesTable = new List<?>[100];
    Object[] objects = (Object[])namesTable;
    List<Integer> numbers = new ArrayList<Integer>();
    numbers.add(5);
    objects[0] = numbers;
    try {
        String name = (String)namesTable[0].get(0);
    }
    catch (ClassCastException expected) {
    }
}
```

The chief distinction is that you must acknowledge the potential for problems by casting, since the `List` is a collection of objects of unknown type.

Additional Limitations

Due to the erasure scheme, an object of a parameterized type has no information on the bind type. This creates a lot of implications about what you can and cannot do.

You cannot create new objects using a naked type variable:

```
package util;

import java.util.*;

public class NumericList<T extends Number> {
    private List<T> data = new ArrayList<T>();

    public void initialize(int size) {
        data.clear();
        T zero = new T(0); // does not compile!
        for (int i = 0; i < size; i++)
            data.add(zero);
    }
}
```

This code generates the compile error:

```
unexpected type
found   : type parameter T
required: class
    T zero = new T(0);
           ^
```

**Additional
Limitations**

Erasure means that a naked type variable erases to its upper bound, Number in this example. In most cases, the upper bound is an abstract class such as Number or Object (the default), so creating objects of that type would not be useful. Java simply prohibits it.

You can cast to a naked type variable, but for the same reason, it is not often useful to do so. You cannot use a naked type variable as the target of the instanceof operator.

You can use type variables in a generic static method. But you cannot use the type variables defined by an enclosing class in static variables, static methods, or static nested classes. Because of erasure, there is only one class definition that is shared by all instances, regardless of the type to which they are bound. The class definition is created at compile time. This means that sharing static elements wouldn't work. Each client that binds the parameterized type to something different would expect to have the static member constrained to the type they specified.

Reflection

The reflection package has been retrofitted to support providing parameter information for parameterized types and methods. For example, you can send the message `getTypeParameters` to a class in order to retrieve an array of `TypeParameter` objects; each `TypeParameter` object provides you with enough information to be able to reconstruct the type parameters.

To support these changes, Sun altered the Java byte code specification. Class files now store additional information about type parameters. Most significant, the `Class` class has been modified to be a parameterized type, `Class<T>`. The following assignment works:

```
Class<String> klass = String.class;
```

If you're interested in how you might use this, take a look at the source for the `CheckedCollection` class. It's a static inner class of `java.util.Collections`.

The reflection modifications provide you with information on the *declaration* of parameterized types and methods. What you will *not* get from reflection is information about the binding of a type variable. If you bind an `ArrayList` to a `String`, that information is not known to the `ArrayList` object because of the erasure scheme. Thus reflection has no way of providing it to you. It would be nice to be able to code

```
public class MultiHashMap<K,V> {  
    ...  
    public Class<V> getKeyType() {  
        return V.class; // this will not work!  
    }  
}
```

but it just won't work.

Final Notes

Final Notes

As you've seen, knowing how the erasure scheme works is key to being able to understand and implement parameterized types. From a client perspective, using parameterized types is relatively easy and imparts the significant benefit of type safety. From the perspective of a developer creating parameterized types, it can be a fairly complex adventure.

If you find yourself struggling with how to understand or define a parameterized type, distill the definition and sample use of the parameterized type

into its erased equivalent. Also, take a look at some of the uses in the J2SE 5.0 source code. The collections framework classes and interfaces, such as Map and HashMap, provide some good examples. For more complex examples, take a look at java.util.Collections.

Exercises

1. Create a new parameterized collection type—a Ring. A Ring is a circular list that maintains knowledge of a current element. A client can retrieve and remove the current element. The Ring must support advancing or backing up the current pointer by one position. The method `add` adds an element after the current element. The Ring class should support client iteration through all elements, starting at the current pointer, using `for-each`. The Ring class should throw appropriate exceptions on any operation that is invalid because the ring is empty.

Do *not* use another data structure to store the ring (e.g., a `java.util.LinkedList`). Create your own link structure using a nested node class. Each node, or entry, should contain three things: the data element added, a reference to the next node in the circle, and a reference to the previous node in the circle.

Exercises

This page intentionally left blank

Lesson 15

Assertions and Annotations

J2SE 5.0 introduced a new facility known as annotations. Annotations are a metaprogramming facility that allow you to mark code with arbitrarily defined tags. The tags (generally) have no meaning to the Java compiler or runtime itself. Instead, other tools can interpret these tags. Examples of tools for which annotations might be useful include IDEs, testing tools, profiling tools, and code-generation tools.

In this lesson you will begin to build a testing tool, similar to JUnit, based upon Java's annotation capabilities. The testing tool, like JUnit, will need to allow developers to specify assertions. Instead of coding assertion methods, you will learn to use Java's built-in assertion capability.

You will learn about:

- assertions
- annotations and annotation types
- retention policies for annotations
- annotation targets
- member-value pairs
- default values for annotation members
- allowable annotation member types
- package annotations
- compatibility considerations for annotations

Assertions

Assertions

You have been using assert methods defined as part of the JUnit API. These assert methods, defined in `junit.framework.Assert`, throw an `AssertionFailedError` when appropriate.

Java supports a similar assertion feature that you can turn on or off using a VM flag. An assertion statement begins with the keyword assert. It is followed by a conditional; if the conditional fails, Java throws a RuntimeException of type AssertionError. You can optionally supply a message to store in the AssertionError by following the conditional with a colon and the String message. An example:

```
assert name != null : "name is required";
```

Without the message:

```
assert name != null;
```

Assertions are disabled by default. When assertions are disabled, the VM ignores assert statements. This prevents assert statements from adversely affecting application performance. To turn assertions on:

```
java -ea MainClass
```

or, more explicitly:

Assertions

```
java -enableassertions MainClass
```

Both of these statements will enable assertions for all your code but not for the Java library classes. You can turn assertions off using -da or -disableassertions. To turn assertions on or off for system classes, use -enablesystemassertions (or -esa) or -disablesystemassertions (or -dsa).

Java allows you to enable or disable assertions at a more granular level. You can turn assertions on or off for any individual class, any package and all its subpackages, or the default package.

For example, you might want to enable assertions for all but one class in a package:

```
java -ea:sis.studentinfo... -da:sis.studentinfo.Session SisApplication
```

The example shown will enable assertions for all classes in sis.studentinfo with the exception of Session. The ellipses means that Java will additionally enable assertions for any subpackages of sis.studentinfo, such as sis.studentinfo.ui. You represent the default package with just the ellipses.

Assertions are disabled/enabled in the order in which they appear on the java command line, from left to right.

The assert Statement vs. JUnit Assert Methods

JUnit was built in advance of the Java assertion mechanism, which Sun introduced in J2SE version 1.4. Today, JUnit could be rewritten to use the assertion mechanism. But since use of JUnit is very entrenched, such a JUnit rewrite would be a major undertaking for many shops. Also, the `junit.framework.Assert` methods supplied by JUnit are slightly more expressive. The `assertEquals` methods automatically provide an improved failure message.

When doing test-driven development, you will always need an assertion-based framework. The use of tests in TDD is analogous to design by contract (referred to as the subcontracting principle in Lesson 6). The assertions provide the preconditions, postconditions, and invariants. If you are not doing TDD, you might choose to bolster the quality of your system by introducing `assert` statements directly in the production code.

Sun recommends against using assertions as safeguards against application failure. For example, one potential use would be to check parameters of public methods, failing if client code passes a `null` reference. The downside is that the application may no longer work properly if you turn off assertions. For similar reasons, Sun recommends that you do not include code in assertions that produces side effects (i.e., code that alters system state).

Nothing prohibits you from doing so, however. If you have control over how your application is executed (and you should), then you can ensure that the application is always initiated with assertions enabled properly.

The chief value of Java's `assert` keyword would seem to be as a debugging aid. Well-placed `assert` statements can alert you to problems at their source. Suppose someone calls a method with a `null` reference. Without a protective assertion, you might not receive a `NullPointerException` until well later in the execution, far from the point where the reference was set. Debugging in this situation can be very time consuming.

You also could use assertions to help build a TDD tool in place of JUnit. In the exercise in this chapter, you will use assertions to begin building such a framework.

Annotations

Annotations

You have already seen some examples of built-in *annotations* that Java supports. In Lesson 2 you learned how you can mark a method as `@deprecated`, meaning that the method is slated for eventual removal from the public interface of a class. The `@deprecated` annotation is technically not part of the Java

language. Instead, it is a tag that the compiler can interpret and use as a basis for printing a warning message if appropriate.¹

You have also seen examples of javadoc tags, annotations that you embed in javadoc comments and that you can use to generate Java API documentation web pages. The javadoc.exe program reads through your Java source files, parsing and interpreting the javadoc tags for inclusion in the web output.

Also, in Lesson 9, you learned how you could use the `@Override` tag to indicate that you believed you were overriding a superclass method.

You can also build your own annotation tags for whatever purpose you might find useful. For example, you might want the ability to mark methods with change comments:

```
@modified("JJL", "12-Feb-2005")
public void cancelReservation() {
    // ...
}
```

Subsequently, you could build a tool (possibly an Eclipse plug-in) that would allow you to view at a glance all the change comments, perhaps sorted by initials.

You could achieve equivalent results by insisting that developers provide structured comments that adhere to a standard format. You could then write code to parse through the source file, looking for these comments. However, the support in Java for custom annotation types provides significant advantages.

First, Java validates annotations at compile time. It is not possible to introduce a misspelled or incorrectly formatted annotation. Second, instead of having to write parse code, you can use reflection capabilities to quickly gather annotation information. Third, you can restrict annotations so that they apply to a specific kind of Java element. For example, you can insist that the `@modified` tag applies only to methods.

Building a Testing Tool

Building a Testing Tool



In this lesson, you will build a testing tool similar to JUnit. I'll refer to this tool as TestRunner, since that will be the primary class responsible for executing the tests. The tool will be text-based. You will be able to execute it from Ant. Technically, you don't have to build a

¹`@deprecated` existed in the Java language from its early days and long before Sun introduced formalized annotations support. But it works just like any other compiler-level annotation type.

testing tool using TDD, because it's not intended to be production code. But there's nothing that says you *can't* write tests. We will.

JUnit requires a test class to extend from the class `junit.framework.TestCase`. In `TestRunner` you will use a different mechanism than inheritance: You will use Java annotations to mark a class as a test class.

Getting started is the toughest part, but using Ant can help. You can set up an Ant target to represent the user interface for the test. If not all of the tests pass, you can set things up so that the Ant build fails, in which case you will see a “BUILD FAILED” message. Otherwise you will see a “BUILD SUCCESSFUL” message.

TestRunnerTest

The first test in `TestRunnerTest`, `singleMethodTest`, goes up against a secondary class `SingleMethodTest` defined in the same source file. `SingleMethodTest` provides a single empty test method that should result in a pass, as it would in JUnit.

So far you need no annotations. You pass a reference to the test class, `TestRunnerTest.class`, to an instance of `TestRunner`. `TestRunner` can assume that this parameter is a test class containing only test methods.

To generate test failures, you will use the assert facility in Java, described in the first part of this lesson. Remember that you must enable assertions when executing the Java VM; otherwise, Java will ignore them.

Here is `TestRunnerTest`.

```
package sis.testing;

import java.util.*;
import java.lang.reflect.*;

public class TestRunnerTest {
    public void singleMethodTest() {
        TestRunner runner = new TestRunner(SingleMethodTest.class);

        Set<Method> testMethods = runner.getTestMethods();
        assert 1 == testMethods.size() : "expected single test method";

        Iterator<Method> it = testMethods.iterator();
        Method method = it.next();

        final String testMethodName = "testA";
        assert testMethodName.equals(method.getName()) :
            "expected " + testMethodName + " as test method";
        runner.run();
        assert 1 == runner.passed() : "expected 1 pass";
        assert 0 == runner.failed() : "expected no failures";
    }
}
```

TestRunnerTest

```

public void multipleMethodTest() {
    TestRunner runner = new TestRunner(MultipleMethodTest.class);
    runner.run();

    assert 2 == runner.passed() : "expected 2 pass";
    assert 0 == runner.failed() : "expected no failures";

    Set<Method> testMethods = runner.getTestMethods();
    assert 2 == testMethods.size() : "expected single test method";

    Set<String> methodNames = new HashSet<String>();
    for (Method method: testMethods)
        methodNames.add(method.getName());

    final String testMethodNameA = "testA";
    final String testMethodNameB = "testB";

    assert methodNames.contains(testMethodNameA):
        "expected " + testMethodNameA + " as test method";
    assert methodNames.contains(testMethodNameB):
        "expected " + testMethodNameB + " as test method";
}

class SingleMethodTest {
    public void testA() {}
}

class MultipleMethodTest {
    public void testA() {}
    public void testB() {}
}

```

TestRunner

The second test, `multipleMethodTest`, is a bit of a mess. To create it, I duplicated the first test and modified some of the details. It screams out for refactoring. The problem is that as soon as you extract a common utility method in `TestRunnerTest`, the `TestRunner` class will assume it's a test method and attempt to execute it. The solution will be to introduce an annotation that you can use to mark and distinguish test methods.

TestRunner

First, let's go over the initial implementation of `TestRunner`.

```

package sis.testing;

import java.util.*;
import java.lang.reflect.*;

```

```

class TestRunner {
    private Class testClass;
    private int failed = 0;
    private Set<Method> testMethods = null;

    public static void main(String[] args) throws Exception {
        TestRunner runner = new TestRunner(args[0]);
        runner.run();
        System.out.println(
            "passed: " + runner.passed() + " failed: " + runner.failed());
        if (runner.failed() > 0)
            System.exit(1);
    }

    public TestRunner(Class testClass) {
        this.testClass = testClass;
    }
    public TestRunner(String className) throws Exception {
        this(Class.forName(className));
    }

    public Set<Method> getTestMethods() {
        if (testMethods == null)
            loadTestMethods();
        return testMethods;
    }

    private void loadTestMethods() {
        testMethods = new HashSet<Method>();
        for (Method method: testClass.getDeclaredMethods())
            testMethods.add(method);
    }

    public void run() {
        for (Method method: getTestMethods())
            run(method);
    }

    private void run(Method method) {
        try {
            Object testObject = testClass.newInstance();
            method.invoke(testObject, new Object[] {})2;
        }
        catch (InvocationTargetException e) {
            Throwable cause = e.getCause();
            if (cause instanceof AssertionException)
                System.out.println(cause.getMessage());
            else
                e.printStackTrace();
            failed++;
        }
    }
}

```

TestRunner

²You can use the slightly-more-succinct idiom `new Object[0]` in place of `new Object[] {}`.

```

        }
        catch (Throwable t) {
            t.printStackTrace();
            failed++;
        }
    }

    public int passed() {
        return testMethods.size() - failed;
    }

    public int failed() {
        return failed;
    }
}

```

If you're having a bit of trouble understanding the `run(Method)` method, refer to Lesson 12 for a discussion of reflection. The basic flow in the `run` method is:

- Create a new instance of the test class. This step assumes a no-argument constructor is available in the test class.
- invoke the method (passed in as a parameter) using the new test class instance and an empty parameter list.
- If the `invoke` message send fails, extract the cause from the thrown `InvocationTargetException`; the cause should normally be an `AssertionError`. Java throws an `AssertionError` when an `assert` statement fails.

TestRunner

`TestRunner` supplies two constructors. One takes a `Class` object and for now will be used from `TestRunnerTest` only. The second constructor takes a class name `String` and loads the corresponding class using `Class.forName`. You call this constructor from the `main` method, which provides a bit of user interface for displaying test results.

The `main` method in turn gets the class name from the Ant target:

```

<target name="runAllTests" depends="build" description="run all tests">
    <java classname="sis.testing.TestRunner" failonerror="true" fork="true">
        <classpath refid="classpath" />
        <jvmarg value="-enableassertions"/>
        <arg value="sis.testing.TestRunnerTest" />
    </java>
</target>

```

There are a few interesting things in the `runAllTests` target:

- You specify `failonerror="true"` in the `java` task. If running a Java application returns a nonzero value, Ant considers the execution to have resulted in an error. The build script terminates on an error. Using the `System.exit`

command (see the `main` method in `TestRunner`) allows you to terminate an application immediately and return the value passed as a parameter to it.

- You specify `fork="true"` in the `java` task. This means that the Java application executes as a separate process from the Java process in which Ant executes. The pitfall of not forking is that the Ant build process itself will crash if the Java application crashes.
- You pass the test name to `TestRunner` using a nested `arg` element.
- You pass the argument `enableassertions` to the Java VM using a nested `jvmarg` element.

The @TestMethod Annotation

In order to be able to refactor tests, you must be able to mark the test methods so that other newly extracted methods are not considered tests. The `@TestMethod` annotation precedes the method signature for each method you want to designate as a test. Annotate the two test methods (`singleMethodTest` and `multipleMethodTest`) in `TestRunnerTest` with `@TestMethod`. Also annotate the three additional test methods in the miniclasses (`SingleMethodTest` and `MultipleMethodTest`) used by the `TestRunnerTest` tests.

The
@TestMethod
Annotation

```
package sis.testing;

import java.util.*;
import java.lang.reflect.*;

public class TestRunnerTest {
    private TestRunner runner;
    private static final String methodNameA = "testA";
    private static final String methodNameB = "testB";

    @TestMethod
    public void singleMethodTest() {
        runTests(SingleMethodTest.class);
        verifyTests(methodNameA);
    }

    @TestMethod
    public void multipleMethodTest() {
        runTests(MultipleMethodTest.class);
        verifyTests(methodNameA, methodNameB);
    }
}
```

```

private void runTests(Class testClass) {
    runner = new TestRunner(testClass);
    runner.run();
}

private void verifyTests(String... expectedTestMethodNames) {
    verifyNumberOfTests(expectedTestMethodNames);
    verifyMethodNames(expectedTestMethodNames);
    verifyCounts(expectedTestMethodNames);
}

private void verifyCounts(String... testMethodNames) {
    assert testMethodNames.length == runner.passed() :
        "expected " + testMethodNames.length + " passed";
    assert 0 == runner.failed() : "expected no failures";
}

private void verifyNumberOfTests(String... testMethodNames) {
    assert testMethodNames.length == runner.getTestMethods().size() :
        "expected " + testMethodNames.length + " test method(s)";
}

private void verifyMethodNames(String... testMethodNames) {
    Set<String> actualMethodNames = getTestMethodNames();
    for (String methodName: testMethodNames)
        assert actualMethodNames.contains(methodName):
            "expected " + methodName + " as test method";
}

private Set<String> getTestMethodNames() {
    Set<String> methodNames = new HashSet<String>();
    for (Method method: runner.getTestMethods())
        methodNames.add(method.getName());
    return methodNames;
}
}

class SingleMethodTest {
    @TestMethod public void testA() {}
}

class MultipleMethodTest {
    @TestMethod public void testA() {}
    @TestMethod public void testB() {}
}

```

The
@TestMethod
 Annotation

The **@TestMethod** annotation may appear after any method modifiers such as **public** or **static**. The annotation must appear before the signature of the method (which starts with the return type of the method).

To declare the **@TestMethod** annotation type, you create what looks a lot like an interface declaration:

```
package sis.testing;
public @interface TestMethod {}
```

The only difference between an interface declaration and an annotation type declaration is that you put an “at” sign (@) before the keyword `interface` in an interface declaration. There can be space between @ and `interface`, but the convention is to abut the two.

The code will now compile and you can execute your tests, but you will see at least one `IllegalAccessException` stack trace. The code in `TestRunner` still treats every method as a test method, including the private methods you just extracted. The reflection code in `TestRunner` is unable to invoke these private methods. It’s time to introduce code in `TestRunner` to look for the `@TestMethod` annotations:

```
private void loadTestMethods() {
    testMethods = new HashSet<Method>();
    for (Method method: testClass.getDeclaredMethods())
        if (method.isAnnotationPresent(TestMethod.class))
            testMethods.add(method);
}
```

One line of code is all it takes. You send the message `isAnnotationPresent` to the `Method` object, passing in the type (`TestMethod.class`) of the annotation. If `isAnnotationPresent` returns `true`, you add the `Method` object to the list of tests.

Now the test executes but returns improper results:

```
runAllTests:
[java] passed: 0 failed: 0
```

You’re expecting to see two passed tests but no tests are registered—the `isAnnotationPresent` method is always returning `false`.

Retention

Retention

The `java.lang.annotations` package includes a *meta-annotation* type named `@Retention`. You use meta-annotations to annotate other annotation type declarations. Specifically, you use the `@Retention` annotation to tell the Java compiler how long to retain annotation information. There are three choices, summarized in Table 15.1.

As explained by the table, if you don’t specify an `@Retention` annotation, the default behavior means that you probably won’t be able to extract informa-

Table 15.1 Annotation Retention Policies

RetentionPolicy enum	Annotation Disposition
RetentionPolicy.SOURCE	Discarded at compile time
RetentionPolicy.CLASS (default)	Stored in the class file; can be discarded by the VM at runtime
RetentionPolicy.RUNTIME	Stored in the class file; retained by the VM at runtime

tion on the annotation at runtime.³ An example of `RetentionPolicy.CLASS` is the `@Override` annotation discussed in Lesson 9.

You will have the most need for `RetentionPolicy.RUNTIME` so that tools such as your TestRunner will be able to extract annotation information from their target classes. If you build tools that work directly with source code (for example, a plug-in for an IDE such as Eclipse), you can use `RetentionPolicy.SOURCE` to avoid unnecessarily storing annotation information in the class files. An example use might be an `@Todo` annotation used to mark sections of code that need attention.

To get reflection code to recognize `@TestMethod` annotations, you must modify the annotation type declaration:

```
Annotation  
Targets
package sis.testing;  
  
import java.lang.annotation.*;  
  
@Retention(RetentionPolicy.RUNTIME)  
public @interface TestMethod {}
```

Your two `TestRunnerTest` tests should now pass:

```
runAllTests:  
[java] passed: 2 failed: 0
```

Annotation Targets

You have designed the `@TestMethod` annotation to be used by developers to mark test methods. Annotations can modify many other *element types*: types (classes, interfaces, and enums), fields, parameters, constructors, local vari-

³The VM may choose to retain this information.

ables, and packages. By default, you may use an annotation to modify any element. You can also choose to constrain an annotation type to modify one *and only one* element type. To do so, you supply an `@Target` meta-annotation on your annotation type declaration.

Since you didn't specify an `@Target` meta-annotation for `@TestMethod`, a developer could use the tag to modify any element, such as a field. Generally no harm would be done, but a developer could mark a field by accident, thinking that he or she marked a method. The test method would be ignored until someone noticed the mistake. Adding an `@Target` to an annotation type is one more step toward helping a developer at compile time instead of making him or her decipher later troubles.

Add the appropriate `@Target` meta-annotation to `@TestMethod`:

```
package sis.testing;

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface TestMethod {}
```

The parameter to `@Target` must be an `ElementType` enum, defined in `java.lang.annotation`. This enum supplies constants corresponding to the element types that an annotation type can modify: `TYPE`, `FIELD`, `METHOD`, `PARAMETER`, `CONSTRUCTOR`, `LOCAL_VARIABLE`, `ANNOTATION_TYPE`, and `PACKAGE`. There are special considerations for annotations with a target of `ElementType.PACKAGE`. See the section Package Annotations later in this lesson for more information.

Annotation Targets

To demonstrate what `@Target` does for your annotation types, modify `TestRunnerTest`. Instead of marking a method with `@TestMethod`, mark a field instead:

```
// ...
public class TestRunnerTest {
    private @TestMethod TestRunner runner;

    @TestMethod
    public void singleMethodTest() {
        // ...
    }
}
```

When you compile, you should see an error similar to the following:

```
annotation type not applicable to this kind of declaration
private @TestMethod TestRunner runner;
^
```

Remove the extraneous annotation, recompile, and rerun your tests.

Skipping Test Methods

 From time to time, you may want to bypass the execution of certain test methods. Suppose you have a handful of failing methods. You might want to concentrate on getting a green bar on one failed method at a time before moving on to the next. The failure of the other test methods serves as a distraction. You want to “turn them off.”

In JUnit, you can skip a method by either commenting it out or by renaming the method so that its signature does not represent a test method signature. An easy way to skip a test method is to precede its name with an `x`. For example, you could rename `testCreate` to `XtestCreate`. JUnit looks for methods whose names start with the word `test`, so it will not find `XtestCreate`.

You do not want to make a habit of commenting out tests. It is poor practice to leave methods commented out for any duration longer than your current programming session. You should avoid checking in code with commented-out tests. Other developers won’t understand your intent. My first inclination when I see commented-out code, particularly test code, is to delete it.

Commenting out test methods is risky. It is easy to forget that you have commented out tests. It can also be difficult to find the tests that are commented out. It would be nice if JUnit could warn you that you’ve left tests commented out.

A similar problem exists for your new `TestRunner` class. The simplest way of bypassing a test would be to remove its `@TestMethod` annotation. The problem with doing that is the same as the problem with commenting out a test. It’s easy to “lose” a test in a system with any significant number of tests.

For this exercise, you will make the necessary modifications to `TestRunner` to ignore designated methods. You will create a new annotation type, `@Ignore`, and change code in `TestRunner` to recognize this annotation. The `@Ignore` annotation will allow developers to supply a parameter with a text description of why the test is being skipped. You will modify `TestRunner` to print these descriptions.

Modifying TestRunner

Modifying TestRunner

Add a new test method, `ignoreMethodTest`, to `TestRunnerTest`. It will go up against a new test class, `IgnoreMethodTest`, which contains three methods marked with `@TestMethod`. One of the test methods (`testC`) is additionally marked `@Ignore`. You must verify that this test method is not executed.

```

package sis.testing;

import java.util.*;
import java.lang.reflect.*;

public class TestRunnerTest {
    private TestRunner runner;
    // ...
    @TestMethod
    public void ignoreMethodTest() {
        runTests(IgnoreMethodTest.class);
        verifyTests(methodNameA, methodNameB);
    }
    // ...
}

// ...
class IgnoreMethodTest {
    @TestMethod public void testA() {}
    @TestMethod public void testB() {}

    @Ignore
    @TestMethod public void testC() {}
}

```

The `@Ignore` annotation declaration looks a lot like the `@TestMethod` declaration.

```

package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Ignore {}

```

**Modifying
TestRunner**

To get your tests to pass, make the following modification to `TestRunner`.

```

package sis.testing;

import java.util.*;
import java.lang.reflect.*;

class TestRunner {
    ...
    private void loadTestMethods() {
        testMethods = new HashSet<Method>();
        for (Method method: testClass.getDeclaredMethods())
            if (method.isAnnotationPresent(TestMethod.class) &&
                !method.isAnnotationPresent(Ignore.class))
                testMethods.add(method);
    }
    ...
}

```

Single-Value Annotations

 The `@Ignore` annotation is a *marker* annotation—it marks whether a method should be ignored or not. You merely need to test for the presence of the annotation using `isAnnotationPresent`. Now you need developers to supply a reason for ignoring a test. You will modify the `@Ignore` annotation to take a reason String as a parameter.

To support a single parameter in an annotation type, you supply a member method named `value` with an appropriate return type and no parameters. Annotation type member methods cannot take any parameters.

```
package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Ignore {
    String value();
}
```

Mark one of the methods in `IgnoreMethodTest` with just `@Ignore`. Do not supply any parameters:

```
class IgnoreMethodTest {
    @TestMethod public void testA() {}
    @TestMethod public void testB() {}
    @Ignore()
    @TestMethod public void testC() {}
}
```

Note that `@Ignore` is a shortcut for `@Ignore()`.

When you compile, you will see an error message:

```
annotation testing.Ignore is missing value
  @Ignore
  ^
```

The compiler uses the corresponding annotation type declaration to ensure that you supplied the proper number of parameters.

Change the test target class, `IgnoreMethodTest`, to supply a reason with the `@Ignore` annotation.

```
public class TestRunnerTest {
    public static final String IGNORE_REASON1 = "because";
    // ...
}
class IgnoreMethodTest {
    @TestMethod public void testA() {}
```

Single-Value Annotations

```

@TestMethod public void testB() {}
@Ignore(TestRunnerTest.IGNORE_REASON1)
    @TestMethod public void testC() {}
}

```

Rerun your tests. They pass, so you haven't broken anything. But you also want the ability to print a list of the ignored methods. Modify the test accordingly:

```

@TestMethod
public void ignoreMethodTest() {
    runTests(IgnoreMethodTest.class);
    verifyTests(methodNameA, methodNameB);
    assertIgnoreReasons();
}

private void assertIgnoreReasons() {
    Map<Method, Ignore> ignoredMethods = runner.getIgnoredMethods();
    Map.Entry<Method, Ignore> entry = getSoleEntry(ignoredMethods);
    assert "testC".equals(entry.getKey().getName());
    "unexpected ignore method: " + entry.getKey();
    Ignore ignore = entry.getValue();
    assert IGNORE_REASON1.equals(ignore.value());
}

private <K, V> Map.Entry<K, V> getSoleEntry(Map<K, V> map) {
    assert 1 == map.size(): "expected one entry";
    Iterator<Map.Entry<K, V>> it = map.entrySet().iterator();
    return it.next();
}

```

Single-Value Annotations

You return the ignored methods as a collection of mappings between Method objects and the “ignored reason” string. Since you expect there to be only one ignored method, you can introduce the utility method `getSoleEntry` to extract the single Method key from the Map. In my excitement over figuring out how to use generics (see Lesson 14), I've gone a little overboard here and made `getSoleEntry` into a generic method that you could use for any collection. There's no reason you couldn't code it specifically to the key and value types of the map.

Now make the necessary changes to `TestRunner` to get it to store the ignored methods for later extraction:

```

package sis.testing;

import java.util.*;
import java.lang.reflect.*;

class TestRunner {
    // ...
    private Map<Method, Ignore> ignoredMethods = null;
    // ...
}

```

```

private void loadTestMethods() {
    testMethods = new HashSet<Method>();
    ignoredMethods = new HashMap<Method, Ignore>();
    for (Method method: testClass.getDeclaredMethods()) {
        if (method.isAnnotationPresent(TestMethod.class))
            if (method.isAnnotationPresent(Ignore.class)) {
                Ignore ignore = method.getAnnotation(Ignore.class);
                ignoredMethods.put(method, ignore);
            }
        else
            testMethods.add(method);
    }
}

public Map<Method, Ignore> getIgnoredMethods() {
    return ignoredMethods;
}
// ...
}

```

You can send the `getAnnotation` method to any element that can be annotated, passing it the annotation type name (`Ignore.class` here). The `getAnnotation` method returns an annotation type reference to the actual annotation object. Once you have the annotation object reference (`ignore`), you can send messages to it that are defined in the annotation type interface.

You can now modify the text user interface to display the ignored methods.

A TestRunner
User Interface
Class

A TestRunner User Interface Class

At this point, the `main` method is no longer a couple of simple hacked-out lines. It's time to move this to a separate class responsible for presenting the user interface.

Since `TestRunner` is a utility for test purposes, as I mentioned earlier, tests aren't absolutely required. For the small bit of nonproduction user interface code you'll write for the test runner, you shouldn't feel compelled to test first. You're more than welcome to do so, but I'm not going to here.

The following listing shows a refactored user interface class that prints out ignored methods. About the only thing interesting in it is the "clever" way I return the number of failed tests in the `System.exit` call. Why? Why not? It's more succinct than an `if` statement, it doesn't obfuscate the code, and it returns additional information that the build script or operating system could use.

```

package sis.testing;

import java.lang.reflect.*;
import java.util.*;

```

```

public class TestRunnerUI {
    private TestRunner runner;

    public static void main(String[] args) throws Exception {
        TestRunnerUI ui = new TestRunnerUI(args[0]);
        ui.run();
        System.exit(ui.getNumberOfFailedTests());
    }

    public TestRunnerUI(String testClassName) throws Exception {
        runner = new TestRunner(testClassName);
    }

    public void run() {
        runner.run();
        showResults();
        showIgnoredMethods();
    }

    public int getNumberOfFailedTests() {
        return runner.failed();
    }

    private void showResults() {
        System.out.println(
            "passed: " + runner.passed() +
            " failed: " + runner.failed());
    }

    private void showIgnoredMethods() {
        if (runner.getIgnoredMethods().isEmpty())
            return;

        System.out.println("\nIgnored Methods");
        for (Map.Entry<Method, Ignore> entry:
             runner.getIgnoredMethods().entrySet()) {
            Ignore ignore = entry.getValue();
            System.out.println(entry.getKey() + ": " + ignore.value());
        }
    }
}

```

**Array
Parameters**

Array Parameters



You want to allow developers to provide multiple separate reason strings. To do so, you can specify `String[]` as the return type for the annotation type member value. An `@Ignore` annotation can then contain multiple reasons by using a construct that looks similar to an array initializer:

```
@Ignore({"why", "just because"})
```

Here's the updated annotation type declaration:

```
package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Ignore {
    String[] value();
}
```

If you only need to supply a single string for an annotation type member with a return type of `String[]`, Java allows you to eliminate the array-style initialization. These annotations for the current definition of `@Ignore` are equivalent:

```
@Ignore("why")
@Ignore({"why"})
```

You will need to modify `TestRunnerTest` to support the change to `Ignore`.

```
package sis.testing;

import java.util.*;
import java.lang.reflect.*;

public class TestRunnerTest {
    public static final String IGNORE_REASON1 = "because";
    public static final String IGNORE_REASON2 = "why not";
    ...

    @TestMethod
    public void ignoreMethodTest() {
        runTests(IgnoreMethodTest.class);
        verifyTests(methodNameA, methodNameB);
        assertIgnoreReasons();
    }

    private void assertIgnoreReasons() {
        Map<Method, Ignore> ignoredMethods = runner.getIgnoredMethods();
        Map.Entry<Method, Ignore> entry = getSoleEntry(ignoredMethods);
        assert "testC".equals(entry.getKey().getName());
        assertEquals("unexpected ignore method: " + entry.getKey(), entry.getValue());
        String[] ignoreReasons = entry.getValue().value();
        assertEquals(2, ignoreReasons.length);
        assertEquals(IGNORE_REASON1, ignoreReasons[0]);
        assertEquals(IGNORE_REASON2, ignoreReasons[1]);
    }
    ...
}
```

Array Parameters

```

class SingleMethodTest {
    @TestMethod public void testA() {}
}

class MultipleMethodTest {
    @TestMethod public void testA() {}
    @TestMethod public void testB() {}
}

class IgnoreMethodTest {
    @TestMethod public void testA() {}
    @TestMethod public void testB() {}

    @Ignore({TestRunnerTest.IGNORE_REASON1,
             TestRunnerTest.IGNORE_REASON2})
    @TestMethod public void testC() {}
}

```

Multiple Parameter Annotations



You may want annotations to support multiple parameters. As an example, suppose you want developers to add their initials when ignoring a test method. A proper annotation might be:

```
@Ignore(reasons={"just because", "and why not"}, initials="jjl")
```

Now that you have more than one annotation parameter, you must supply *member-value pairs*. Each member-value pair includes the member name, which must match an annotation type member, followed by the equals (=) sign, followed by the constant value for the member.

The second member-value pair in the above example has `initials` as a member name and "jjl" as its value. In order to support this annotation, you must modify the `@Ignore` annotation type declaration to include `initials` as an additional member. You must also rename the value member to `reasons`. Each key in an annotation member-value pair must match a member name in the annotation type declaration.

```

package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Ignore {
    String[] reasons();
    String initials();
}

```

Multiple
Parameter
Annotations

You can specify member-value pairs in any order within an annotation. The order need not match the member order of the annotation type declaration.

Here are the corresponding modifications to TestRunnerTest:

```
package sis.testing;
// ...
public class TestRunnerTest {
    // ...
    public static final String IGNORE_INITIALS = "jj1";
    // ...
    private void assertIgnoreReasons() {
        Map<Method, Ignore> ignoredMethods = runner.getIgnoredMethods();
        Map.Entry<Method, Ignore> entry = getSoleEntry(ignoredMethods);
        assert "testC".equals(entry.getKey().getName());
        assertEquals("unexpected ignore method: " + entry.getKey(), entry.getValue());
        String[] ignoreReasons = entry.reasons();
        assertEquals(2, ignoreReasons.length);
        assertEquals(IGNORE_REASON1, ignoreReasons[0]);
        assertEquals(IGNORE_REASON2, ignoreReasons[1]);
        assertEquals(IGNORE_INITIALS, entry.initials());
    }
    // ...
}
class IgnoreMethodTest {
    @TestMethod public void testA() {}
    @TestMethod public void testB() {}

    @Ignore(
        reasons={TestRunnerTest.IGNORE_REASON1,
                  TestRunnerTest.IGNORE_REASON2},
        initials=TestRunnerTest.IGNORE_INITIALS)
    @TestMethod public void testC() {}
}
```

Multiple Parameter Annotations

You do not need to make modifications to TestRunner. You will need to make a small modification to TestRunnerUI to extract the reasons and initials properly from the Ignore object.

```
private void showIgnoredMethods() {
    if (runner.getIgnoredMethods().isEmpty())
        return;

    System.out.println("\nIgnored Methods");
    for (Map.Entry<Method, Ignore> entry:
        runner.getIgnoredMethods().entrySet()) {
        Ignore ignore = entry.getValue();
        System.out.printf("%s: %s (by %s)",
                         entry.getKey(),
```

```

        Arrays.toString(ignore.reasons()),
        ignore.initials());
    }
}

```

Default Values

 The reason for ignoring a test method is likely to be the same most of the time. Most often, you will want to temporarily comment out a test while fixing other broken tests. Supplying a reason each time can be onerous, so you would like to have a default ignore reason. Here's how you might reflect this need in a TestRunner test:

```

@TestMethod
public void ignoreWithDefaultReason() {
    runTests(DefaultIgnoreMethodTest.class);
    verifyTests(methodNameA, methodNameB);
    Map<Method, Ignore> ignoredMethods = runner.getIgnoredMethods();
    Map.Entry<Method, Ignore> entry = getSoleEntry(ignoredMethods);
    Ignore ignore = entry.getValue();
    assertEquals(TestRunner.DEFAULT_IGNORE_REASON,
        ignore.reasons()[0]);
}

class DefaultIgnoreMethodTest {
    @TestMethod public void testA() {}
    @TestMethod public void testB() {}
    @Ignore(initials=TestRunnerTest.IGNORE_INITIALS)
    @TestMethod public void testC() {}
}

```

Default Values

You will need to define the constant `DEFAULT_IGNORE_REASON` in the `TestRunner` class to be whatever string you desire:

```

class TestRunner {
    public static final String DEFAULT_IGNORE_REASON =
        "temporarily commenting out";
    // ...
}

```

You can supply a default value on any annotation type member. The default must be a constant at compile time. The new definition of `@Ignore` includes a default value on the `reasons` member. Note use of the keyword `default` to separate the member signature from the default value.

```

package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)

```

```
public @interface Ignore {
    String[] reasons() default TestRunner.DEFAULT_IGNORE_REASON;
    String initials();
}
```

Additional Return Types and Complex Annotation Types

In addition to `String` and `String[]`, an annotation value can be a primitive, an enum, a `Class` reference, an annotation type itself, or an array of any of these types.

The following test (in `TestRunnerTest`) sets up the requirement for an `@Ignore` annotation to include a date. The `Date` type will be an annotation; its members each return an `int` value.

```
@TestMethod
public void dateTest() {
    runTests(IgnoreDateTest.class);
    Map<Method, Ignore> ignoredMethods = runner.getIgnoredMethods();
    Map.Entry<Method, Ignore> entry = getSoleEntry(ignoredMethods);
    Ignore ignore = entry.getValue();
    sis.testing.Date date = ignore.date();
    assert 1 == date.month();
    assert 2 == date.day();
    assert 2005 == date.year();
}

class IgnoreDateTest {
    @Ignore(
        initials=TestRunnerTest.IGNORE_INITIALS,
        date=@Date(month=1, day=2, year=2005))
    @TestMethod public void testC() {}
}
```

The annotation in `IgnoreDateTest` is known as a *complex annotation*—an annotation that includes another annotation. `@Ignore` includes a member, `date`, whose value is another annotation, `@Date`.

The definition of the `sis.testing.Date` annotation type:

```
package sis.testing;

public @interface Date {
    int month();
    int day();
    int year();
}
```

The `@Ignore` annotation type can now define a date member that returns a `testing.Date` instance:

```
package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Ignore {
    String[] reasons() default TestRunner.DEFAULT_IGNORE_REASON;
    String initials();
    Date date();
}
```

Since the `Date` annotation type is only used as part of another annotation, it does not need to specify a retention or a target.

You may not declare a recursive annotation type—that is, an annotation type member with the same return type as the annotation itself.

To get your tests to compile and pass, you'll also need to modify `IgnoreMethodTest` and `DefaultIgnoreMethodTest`:

```
class IgnoreMethodTest {
    @TestMethod public void testA() {}
    @TestMethod public void testB() {}

    @Ignore(
        reasons={TestRunnerTest.IGNORE_REASON1,
                  TestRunnerTest.IGNORE_REASON2},
        initials=TestRunnerTest.IGNORE_INITIALS,
        date=@Date(month=1, day=2, year=2005))
    @TestMethod public void testC() {}
}

class DefaultIgnoreMethodTest {
    @TestMethod public void testA() {}
    @TestMethod public void testB() {}
    @Ignore(initials=TestRunnerTest.IGNORE_INITIALS,
           date=@Date(month=1, day=2, year=2005))
    @TestMethod public void testC() {}
}
```

Package
Annotations

Package Annotations



Suppose you want the ability to designate packages as testing packages. Further, you want your testing tool to run “performance-related” tests separately from other tests. In order to accomplish this,

you can create an annotation whose target is a package. A test for this annotation:⁴

```
@TestMethod
public void packageAnnotations() {
    Package pkg = this.getClass().getPackage();
    TestPackage testPackage = pkg.getAnnotation(TestPackage.class);
    assert testPackage.isPerformance();
}
```

You extract annotation information from the Package object like you would from any other element object. You can obtain a Package object by sending the message `getPackage` to any Class object.

The annotation declaration is straightforward:

```
package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.PACKAGE)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestPackage {
    boolean isPerformance() default false;
}
```

However, the question is, Where does a package annotation go? Would it go before the package statement in every source file that belongs to the package? Or before just one of them? Or is it stored elsewhere?

Java limits you to at most one annotated package statement per package. This means that you can't just place an annotation before the package statement in an arbitrary source file.

The answer depends on which compiler you are using. Sun recommends a specific scheme that is based on a file system. Other compiler vendors may choose a different scheme. They may have to if the compile environment is not based on a file system.

The Sun scheme requires you to create a source file named `package-info.java` in the source directory that corresponds to the package you want to annotate. Sun's Java compiler reads this pseudo-source file but produces no visible class files as output. (In fact, it is not possible to include a hyphen [-] in a class name.) This file should include any package annotations, followed by the appropriate package statement. You should not include anything else in `package-info.java`.

Here's what `package-info.java` might look like in the `sis.testing` package:

```
@TestPackage(isPerformance=true) package sis.testing;
```

⁴Note use of the variable name `pkg`, since `package` is a keyword.

Compatibility Considerations

Sun has, as much as possible, designed the annotations facility to support changes to an annotation with minimal impact on existing code. This section goes through the various modification scenarios and explains the impact of each kind of change to an annotation type.

When you add a new member to an annotation type, provide a default if possible. Using a default will allow code to continue to use the compiled annotation type without issue. But this *can* cause a problem if you try to access the new member. Suppose you access the new member from an annotation compiled using the annotation type declaration *without* the new member. If no default exists on the new member, this will generate an exception.

If you remove an annotation type member, you will obviously cause errors on recompilation of any likewise annotated sources. However, any existing class files that use the modified annotation type will work fine until their sources are recompiled.

Avoid removing defaults, changing the return type, or removing target elements with existing annotation types. These actions all have the potential to generate exceptions.

If you change the retention type, the behavior is generally what you would expect. For example, if you change from `RUNTIME` to `CLASS`, the annotation is no longer readable at runtime.

When in doubt, write a test to demonstrate the behavior!

**Additional
Notes on
Annotations**

Additional Notes on Annotations

- An annotation with no target can modify any Java element.
- In an annotation type declaration, the only parameterized type that you can return is the `Class` type.
- The `@Documented` meta-annotation type lets you declare an annotation type to be included in the published API generated by tools such as javadoc.
- The `@Inherited` meta-annotation type means that an annotation type is inherited by all subclasses. It will be returned if you send `getAnnotation` to a method or class object but not if you send `getDeclaredAnnotations`.
- You cannot use `null` as an annotation value.

- You can modify an element only once with a given annotation. For example, you cannot supply two `@Ignore` annotations for a test method.
- In order to internally support annotation types, Sun modified the `Arrays` class to include implementations of `toString` and `hashCode` for `Arrays`.

Summary

Annotations are a powerful facility that can help structure the notes you put into your code. One of the examples that is touted the most is the ability to annotate interface declarations so that tools can generate code from the pertinent methods.

The chief downside of using annotations is that you make your code dependent upon an annotation type when your code uses it. Changes to the annotation type declaration could negatively impact your code, although Sun has built in some facilities to help maintain binary compatibility. You also must have the annotation type class file present in order to compile. This should not be surprising: An annotation type is effectively an interface type.

Again, the rule of thumb is to use annotation types prudently. An annotation type is an interface and should represent a stable abstraction. As with any interface, ensure that you have carefully considered the implications of introducing an annotation type into your system. A worst-case scenario, where you needed to make dramatic changes to an annotation type, would involve a massive search and replace⁵ followed by a compile.

Exercises

Exercises

1. Using the `RingTest` and `Ring` classes from the previous lesson, introduce an `assert` into the `add` method that rejects a `null` argument. Make sure you write a test! Don't forget to enable assertions before running your tests.

⁵For the time being, IDEs might not provide sophisticated support for manipulating and navigating annotations. You may need to resort to search and replace (or compile, identify, and replace) to effect a major change. Integrated IDE support for annotations should emerge quickly. The support in IDEA is already at a level I would consider acceptable.

2. Create an annotation `@Dump`. This annotation can be applied to any field in a class. Then create a `ToStringer` class which, when passed an object, will spit out a dump of each field that has been marked for use in the `toString` method.
3. Modify the dump annotation to allow for sorting the fields by adding an optional `order` parameter. The order should be a positive integer. If a field is not annotated, it should appear last in the list of fields.
4. Add another parameter for dump called `quote`, which should be a boolean indicating whether or not to surround the value with quotes. This is useful for objects whose `toString` representation might be empty or have leading or trailing spaces.
5. Add an `outputMethod` field to the `@Dump` annotation. This specifies the method for `Tostringer` to use in order to get a printable representation of the field. Its value should default to `toString`. This is useful for when you have an object with a `toString` representation you cannot change, such as an object of a system class type.
6. Change the `outputMethod` annotation to `outputMethods` and have it support a String array of method names. The `ToString` code should construct a printable representation of the object by calling each of these method names in order and concatenating the results. Separate each result with a single space. (You might consider adding another annotation to designate the separator character.)

Exercises

This page intentionally left blank

Additional Lesson I

Swing, Part 1

Swing is Java's standard library for cross-platform GUI (graphical user interface) development. The Swing API provides a rich feature set, allowing you to construct sophisticated user interfaces. Using Swing, you can construct applications that allow the user to interact through basic controls¹ such as push buttons, entry fields, and list boxes or using advanced controls such as tables, trees, and drag & drop.

This chapter introduces Swing. It is an overview that shows you the basics: how to construct simple applications using some of the more common widgets. By no means does this chapter encompass everything about Swing. In fact, it only scratches the surface. Authors have devoted entire books, and even multiple volumes, to the topic.

Learning the basics of Swing, however, will provide you with a foundation for understanding how the rest of it works and how to go about finding more information. Once you have learned how to build a table widget using a table model, for example, learning how to build a tree widget using a tree model is easy. The Java API documentation usually provides sufficient information about how to use a widget. For more complex widgets, including the tree model and things such as sophisticated widget layouts, the Java API documentation often has a link to a good Sun tutorial on the topic.

More important, in this chapter you will learn various approaches to and philosophies about how to test Swing applications. Developers often view testing Swing applications as a difficult proposition and decide instead to forgo it. As a result, Swing code is frequently an untested, poorly designed mess.

This chapter is as much about ideas for testing Swing as it is about Swing itself. Testing Swing applications can be difficult, but that isn't an excuse to

Swing, Part 1

¹Also known as widgets or components, controls are elements of graphical user interfaces that either present information or allow end users some aspect of control over their interaction with the user interface.

not do so. There are huge benefits to having well-tested, well-designed user interface code.

There are two different aspects of design for user interfaces. The Java you design and code to construct the user interface is the aspect you will concern yourself with in this chapter. Another aspect of design is the aesthetic and functional quality of the user interface—its look and feel.

The look and feel represents requirements—how does an end user need to interact with the application? The customer, or perhaps a user interface design expert working with the customer, should derive and present these requirements to the development team. Many shops involve members of the development team in the GUI design as well. The format in which the customer presents requirements to the developers could be screen snapshots, crude drawings, or various formalized diagrams.

You will learn about:

- Swing application design
- panels and frames
- buttons and action listeners
- lists and list models
- Swing layouts: BorderLayout, GridLayout, FlowLayout, BoxLayout, GridBagLayout

Swing

Swing is not your only choice for GUI development, but it is available to you “out of the box.” The Eclipse IDE was built using SWT (Standard Widget Toolkit), an API that you could also choose to use for your applications.

Swing is actually built atop another Java API known as AWT (Abstract Window Toolkit). Sun introduced AWT with the first version of Java as the sole means for producing GUIs. It consisted of a small number of widgets that at the time were guaranteed to be available on all platforms supported by Java. This “lowest common denominator” design decision allowed you to build cross-platform GUI applications but restricted your expressiveness with respect to GUI design.

AWT is known as a heavyweight GUI framework. Java tightly couples each AWT control to an equivalent control directly supported by the operating system. The operating system manages each AWT control.

Swing, which was introduced a few years after AWT, is, in contrast, a lightweight GUI framework. Swing eliminates the lowest common denomina-

tor problem by creating controls that use heavyweight controls as a “shell.” Swing components inherit from AWT components. They add custom rendering and interaction code so that they act and look far more sophisticated than the controls supported by all operating systems.

You can attach a look and feel to your Swing application so that it mimics a specific operating system. For example, you can tell Java to render your Swing controls so that they look like Motif controls or like Windows controls.

Getting Started



You will build a simple Swing application that presents a lists of courses to the end user. It will also allow the user to add new courses.

As you learn Swing, you might start with spike solutions—bits of code that demonstrate how Swing should work. Once you get these spikes working, you will go back and figure out how to test the Swing code. This chapter sometimes presents its material in this fashion: I will demonstrate how to code something in Swing, then supply code that shows how to test it.

Most applications, whether they use Swing or not, initiate with a *frame*. A frame is a top-level window with a title bar and border by default. The frame is largely drawn by and under the control of your operating system. Within Java, the Swing frame class is `javax.swing.JFrame`. Building an initial application in Java is as simple as constructing a `JFrame`, setting its size, and making it visible.

Create the class `Sis` (short for Student Information System).

```
package sis.ui;

import javax.swing.*;

public class Sis {
    static final int WIDTH = 350;
    static final int HEIGHT = 500;

    public static void main(String[] args) {
        new Sis().show();
    }

    public void show() {
        JFrame frame = new JFrame();
        frame.setSize(WIDTH, HEIGHT);
        frame.setVisible(true);
    }
}
```

Getting Started



Figure 1 A Frame Window

The definition for Sis includes a `main` method to allow you to execute the application from the command line. About the only things you ever want to do in a `main` method are to construct a new instance and call a single method on it. You may first need to call a method to parse the command-line arguments. If your `main` method is more involved, you need to refactor. Resist adding any real logic to `main`—`main` is usually not comprehensively tested.

Compile and execute Sis. You should see a simple frame window (see Figure 1). Experiment with this window and note that you can do things such as size it, minimize it, or close it like any other windowed application. You may also want to experiment with the `setSize` and `setVisible` messages to see how they impact the frame. Try omitting either or both message sends.

The only problem is that when you close this window, the Java process does not terminate. Behind the scenes, Swing creates user threads. Exiting the `main` method does not result in the termination of these user threads. See the section Shutting Down in Lesson 13 for a discussion of user threads and daemon threads.

You can tell the `JFrame` to terminate the Java process by telling it to exit on `close`:

```
package sis.ui;

import javax.swing.*;

public class Sis {
    static final int WIDTH = 300;
    static final int HEIGHT = 200;

    public static void main(String[] args) {
        new Sis().show();
    }
}
```

Getting Started

```

void show() {
    JFrame frame = new JFrame();
    frame.setSize(WIDTH, HEIGHT);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
}
}

```

Now that you understand what it takes to build a frame, you can toss this code away and write a couple of tests. You will want to refer to the Java API documentation to see what kinds of queries you can send to a `JFrame` object. The tests show that you can inspect each piece of information you use to initialize the frame. You can also query whether or not the frame is visible.

```

package sis.ui;

import junit.framework.*;
import javax.swing.*;

public class SisTest extends TestCase {
    private Sis sis;
    private JFrame frame;

    protected void setUp() {
        sis = new Sis();
        frame = sis.getFrame();
    }

    public void testCreate() {
        final double tolerance = 0.05;
        assertEquals(Sis.HEIGHT, frame.getSize().getHeight(), tolerance);
        assertEquals(Sis.WIDTH, frame.getSize().getWidth(), tolerance);
        assertEquals(JFrame.EXIT_ON_CLOSE,
                    frame.getDefaultCloseOperation());
    }

    public void testShow() {
        sis.show();
        assertTrue(frame.isVisible());
    }
}

```

Getting Started

The code that meets this test ends up slightly different than the spike code:

```

package sis.ui;

import javax.swing.*;

public class Sis {
    static final int WIDTH = 300;
    static final int HEIGHT = 200;

    private JFrame frame = new JFrame();

```

```

public static void main(String[] args) {
    new Sis().show();
}

Sis() {
    frame.setSize(WIDTH, HEIGHT);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}

void show() {
    frame.setVisible(true);
}

JFrame getFrame() {
    return frame;
}
}

```

When you run the test, you will see the frame window display on screen, because the test instantiates Sis. Resist embedding such tests that pop up windows. It is very distracting and will slow down your tests considerably. You will learn techniques later that eliminate the need to render the user interface while running tests. For now, a test or two that pops up windows *might* be considered acceptable.

The frame window does not disappear even after the test completes. This is a bigger distraction that you can fix. In your test's `tearDown` method, tell the application to close:

```

public class SisTest extends TestCase {
    ...
    protected void tearDown() {
        sis.close();
    }
    ...
}

```

Getting Started

Close the application by having it dispose of the frame window:

```

package sis.ui;

import javax.swing.*;

public class Sis {
    ...
    private JFrame frame = new JFrame();
    ...
    void close() {
        frame.dispose();
    }
}

```

View First or Model First?

When developing a user interface, the question often arises of which to approach first: building the model or building the view? There is no right answer. It depends on your comfort level and ultimately on the approach you derive the most success from.

In the core 15 lessons of Agile Java, you learned to build Java classes in the absence of a user interface or place in a larger system. In reality, you will find that context often changes your design. Application needs may significantly impact the public interface of model classes.

I've been most successful when I've designed a system "application first." Each time I've built model classes in isolation, I've been forced to alter their behavior considerably when attempting to drop them into an application.

Even though I center the development of a system around its application classes, I do so with the view and its needs clearly in mind. And in reality, I rarely develop any of a system "first." I've found it far more effective to bounce around between the various layers, using tests and their results as my guide.

Swing Application Design

Designing a Swing application revolves around the organization of responsibilities germane to any application based on a user interface. These responsibilities include displaying information, managing input, modeling the business logic, and managing the application flow.

The terminology for these responsibilities is reasonably standard. A *view* displays things to the end user. The JFrame object in Sis is a view object. A *controller* manages input from the end user via the keyboard, mouse, or other device. A view may interact with one or more *models*, also known as domain objects. For example, code in the Sis application will need to interact with model objects of type Course and Student. Finally, the *application* coordinates models, views, and controllers. The application is responsible for the sequencing of a user's experience.

There are many ways to approach building Swing applications. You might hear different terms for these responsibilities. You might also encounter terms that imply various overlaps and combinations of the responsibilities. For example, Swing classes themselves often combine view and controller logic. (In fact, when you hear me refer to view in this chapter, I usually mean the combination of view and controller.) Nonetheless, the core responsibilities always exist. In this lesson, we will see where TDD takes us in terms of design—I've not forced any rigid design on the development of your example.

Swing
Application
Design

For the example, you will initially code the view portion of the Sis application. Per the Single-Responsibility Principle, the only code you should have in a view class is code responsible for displaying and arranging the user interface. A significant benefit of organizing the code this way is that you can execute the panel as a stand-alone application. This allows you to concentrate on developing the look, or *layout*, of the user interface.

In contrast, many Swing applications do not follow this rule. The view class in these applications is cluttered with other interactions. For example, the view class might send a message to a model class that in turn accesses data from the database. It is virtually impossible to display the view in isolation—that is, without executing the entire application. This design results in a significantly decreased rate of layout development. Often, you can access the view in question only by traversing a number of other application screens.

Panels

A JFrame envelopes a content pane—a place where you put visual content. A content pane object is a *container* of type java.awt.Container. You add other components to a container, including things such as list boxes, text, and containers themselves. In Swing, the workhorse container class is JPanel. Like JFrame, the JPanel class is a view class, used only for display purposes.

You may have noticed the package and naming convention. Swing components all begin with the letter J (JFrame, JPanel, JList, and so on); their containing package is javax.swing or a subpackage of javax.swing. The package java.awt (or a subpackage thereof) contains AWT classes. AWT class names do not use a prefix.

Your first piece of relevant user interface will be to display the text “Courses:” to the end user. You could directly embed this text in the JFrame’s content pane. A better approach is to build up a second container consisting of the text, then embed this container in the content pane. To do so, you will create a JPanel subclass that displays text to the end user. You will then embed this JPanel in the JFrame’s content pane.

Here is the spike for the JPanel:

```
package sis.ui;  
  
import javax.swing.*;  
import java.awt.*;
```

```

public class CoursesPanel extends JPanel {
    public static void main(String[] args) {
        show(new CoursesPanel());
    }

    private static void show(JPanel panel) {
        JFrame frame = new JFrame();
        frame.setSize(100, 100);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(panel); // 1
        frame.setVisible(true);
    }

    public CoursesPanel() {
        JLabel label = new JLabel("Courses:");
        add(label);
    }
}

```

The spike uses a `main` method containing a bit of “driver” code to allow you to manually test the panel. If you were to code a second panel class, you would be compelled to refactor this driver code into a common utility class.

Within the constructor of `CoursesPanel`, you create a new `JLabel` object. A `JLabel` is a widget that displays text to an end user. You specify the text via the constructor of `JLabel`. In order for the `JLabel` to be displayed, you must add it to the `JPanel` using the `add` method.

`CoursesPanel` does not define `add`, nor does the `JPanel` class from which `CoursesPanel` extends. You must traverse the inheritance hierarchy up to the class `java.awt.Container` in order to find the definition for `add`. It makes sense for `add` to belong there: You add other components, including `JPanel` objects, to `Container` objects.

Following is the inheritance hierarchy for `CoursesPanel`.

```

Object
 |
+-java.awt.Component
 |
+-java.awt.Container
 |
+-javax.swing.JComponent
 |
+-javax.swing.JPanel
 |
+-sis.ui.CoursesPanel

```

Panels

Everything in Swing sits atop the AWT framework. Everything is a Component. All Swing components (`JComponent` objects) are also containers. `JPanel` and `CoursesPanel` are therefore containers.



Figure 2 A JPanel

In order to display a panel, you can add it to the content pane of a JFrame. The line marked 1 in the `show` method of CoursesPanel accomplishes this.

Compile and execute the CoursesPanel spike. The result should look like Figure 2.

What should a test for CoursesPanel prove? For now, the only significant thing that CoursesPanel does is to present a label to the user. The test should thus ensure that the panel contains a label with the correct text:

```
package sis.ui;

import junit.framework.*;
import javax.swing.*;
import java.awt.*;

public class CoursesPanelTest extends TestCase {
    public void testCreate() {
        CoursesPanel panel = new CoursesPanel();
        JLabel label = getLabel(panel);
        assertEquals(CoursesPanel.LABEL_TEXT, label.getText());
    }

    private JLabel getLabel(JPanel panel) {
        for (Component component: panel.getComponents())
            if (component instanceof JLabel)
                return (JLabel)component;
        return null;
    }
}
```

Panels

The job of the method `getLabel` is to take as its parameter a JPanel and return the first JLabel object encountered within the JPanel. Since a JPanel is a Container, you can ask it for the list of components it contains using the `getComponents` message. Each element contained within the list is of type `java.awt.Component`. You must test the type of each element using `instanceof` to determine if it is a JLabel. You must also cast the matching element in order to return it as a JLabel type.

The use of `instanceof` suggests that perhaps a better approach works to determine if a component exists. For now, the technique will suffice until you have to test for a second type of component—or a second label.

Is this test necessary? The answer is debatable. You will always manually execute a Swing application from time to time to take a look, even if you were to write thousands of automated tests. In doing so, you will quickly determine whether or not a specific component appears.

One could also ask the opposite question: *Is this test sufficient?* Not only could you test that the widget appeared in the window but you could also test that it was presented in the appropriate font and appeared at the correct coordinates.

My view is that the test may be mild overkill, but it's not a difficult test to write. I would rather expend the effort than find out the hard way that someone accidentally changed the `JLabel` text in one of 200 screens. Positioning-based layout tests (tests that prove where the label appears onscreen) are another matter. They are notoriously difficult to write and even more difficult to maintain. A bit of aesthetic distress is not usually a barrier to successful application execution.

More important is that you write tests for any dynamic user interface capabilities. For example, you might allow the application to change the color of text when the user presses a button. You will want to write a test to ensure that this action and reaction works as you expect.

The `CoursesPanel` class differs from the spike only in that it defines a class constant representing the label text.

```
package sis.ui;

import javax.swing.*;
import java.awt.*;

public class CoursesPanel extends JPanel {
    static final String LABEL_TEXT = "Courses";
    ...
    public CoursesPanel() {
        JLabel label = new JLabel(LABEL_TEXT);
        add(label);
    }
}
```

Panels

In the `Sis` application, you want `CoursesPanel` to represent the main view. The modified `SisTest` uses a similar mechanism to that in `CoursesPanelTest` to prove that the content pane contains a `CoursesPanel` instance.

```
package sis.ui;

import junit.framework.*;
import javax.swing.*;
import java.awt.*;
```

```

public class SisTest extends TestCase {
    ...
    public void testCreate() {
        final double tolerance = 0.05;
        assertEquals(Sis.HEIGHT, frame.getSize().getHeight(), tolerance);
        assertEquals(Sis.WIDTH, frame.getSize().getWidth(), tolerance);
        assertEquals(JFrame.EXIT_ON_CLOSE,
                    frame.getDefaultCloseOperation());
        assertTrue(containsCoursesPanel(frame));
    }
}

private boolean containsCoursesPanel(JFrame frame) {
    Container pane = frame.getContentPane();
    for (Component component: pane.getComponents())
        if (component instanceof CoursesPanel)
            return true;
    return false;
}
...
}

```

You're already familiar with how to get a frame to include and show a panel. You wrote virtually the same code in the "driver" for CoursesPanel.

```

package sis.ui;

import javax.swing.*;

public class Sis {
    ...
    Sis() {
        frame.setSize(WIDTH, HEIGHT);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(new CoursesPanel());
    }
    ...
}

```

Refactoring

The tests pass.

Refactoring

In the course of building this example for the book, I coded the Sis constructor to include the statement that makes the frame visible. Michael Feathers reminded me that constructors are for initialization only. It's a bit of a stretch, I countered, but you could consider displaying the window as part of its initialization. To which he replied, using wisdom from Ron Jeffries, that a constructor has a lot of nerve doing something it isn't asked to do.

I listened to the voices of these two and fixed the deficiency. However, I still consider that GUI widget and layout construction is simple initialization. Putting panels within a frame is layout initialization. While you could conceivably separate this layout initialization from object initialization, there is little value in doing so. A JPanel subclass with uninitialized contents is of little use. Forcing clients to make an additional call to initialize its layout is redundant.

Separating the `setVisible` statement from the constructor is of potential value, however. In the case of `SisTest`, doing so allowed for a separate initialization test that doesn't force the frame to be rendered. In production systems, it's often valuable to initialize a frame behind the scenes, later controlling its visibility separately.

Ultimately the decision is somewhat arbitrary and thus academic. With regards to `CoursesPanel`, a future requirement may require a developer to create a `CoursesPanel` subclass. Putting initialization code in a separate method makes it easier to override or extend the initialization functionality. Yet you want to avoid designing for future what-ifs (that often never come). It's just as easy to wait and make the change down the road—if necessary.

An acceptable compromise is to consider simple design rule #3 and extract initialization to a private method for readability purposes:

```
public CoursesPanel() {
    createLayout();
}

private void createLayout() {
    JLabel label = new JLabel(LABEL_TEXT);
    add(label);
}
```

Refactoring

The code in `SisTest.containsCoursesPanel` is very similar in form to that in `CoursesPanelTest.getLabel`. Both methods loop through the components in a container and test for a match.

The duplicate code is a problem that will only get worse. Adding buttons, list boxes, or other component types will require a new method for each type. Additionally, suppose you have two labels on a panel. The current code will grab only the first label.

One solution is to provide a name for each component. You can then loop through the list of subcomponents until you find a matching name. This solution requires a little more management overhead in your view class, but it makes testing easier. It also can make aggregate operations on components easier, as you'll see later.

All component and container classes inherit from `java.awt.Component`. In this class you will find the methods `setName` and `getName` that respectively take and return a `String` object.

Starting with the code in `SisTest`:

```
public void testCreate() {
    final double tolerance = 0.05;
    assertEquals(Sis.HEIGHT, frame.getSize().getHeight(), tolerance);
    assertEquals(Sis.WIDTH, frame.getSize().getWidth(), tolerance);
    assertEquals(JFrame.EXIT_ON_CLOSE,
        frame.getDefaultCloseOperation());
    assertNotNull(getComponent(frame, CoursesPanel.NAME));
}

private Component getComponent(JFrame frame, String name) {
    Container container = frame.getContentPane();
    for (Component component: container.getComponents())
        if (name.equals(component.getName()))
            return component;
    return null;
}
```

The corresponding changes to `CoursesPanel`:

```
package sis.ui;
...
public class CoursesPanel extends JPanel {
    static final String NAME = "coursesPanel";
    ...
    public CoursesPanel() {
        setName(NAME);
        createLayout();
    }
}
```

Refactoring

Part of the secret to eliminating duplication is to push toward use of abstractions. For example, the new `getComponent` method deals in terms of abstract components instead of the concrete type `CoursesPanel`. Pushing the code in `CoursesPanelTest` in this direction leads to:

```
package sis.ui;
...
public class CoursesPanelTest extends TestCase {
    public void testCreate() {
        CoursesPanel panel = new CoursesPanel();
        JLabel label =
            (JLabel)getComponent(panel, CoursesPanel.LABEL_NAME);
        assertEquals(CoursesPanel.LABEL_TEXT, label.getText());
    }

    private Component getComponent(Container container, String name) {
        for (Component component: container.getComponents())
            if (name.equals(component.getName()))
```

```

        return component;
    return null;
}
}

// in CoursesPanel:
...
public class CoursesPanel extends JPanel {
    static final String LABEL_NAME = "coursesLabel";
    ...
    private void createLayout() {
        JLabel label = new JLabel(LABEL_TEXT);
        label.setName(LABEL_NAME);
        add(label);
    }
}

```

At this point, the `getComponent` methods are different only by the first line. For lack of a better place, move them into a new class as class methods and refactor. Here is the code for the resulting class, `sis.ui.Util`, as well as for the modified test classes:

```

// sis.ui.Util
package sis.ui;

import java.awt.*;
import javax.swing.*;

class Util {
    static Component getComponent(Container container, String name) {
        for (Component component: container.getComponents())
            if (name.equals(component.getName()))
                return component;
        return null;
    }

    static Component getComponent(JFrame frame, String name) {
        return getComponent(frame.getContentPane(), name);
    }
}
// sis.ui.SisTest
public void testCreate() {
    final double tolerance = 0.05;
    assertEquals(Sis.HEIGHT, frame.getSize().getHeight(), tolerance);
    assertEquals(Sis.WIDTH, frame.getSize().getWidth(), tolerance);
    assertEquals(JFrame.EXIT_ON_CLOSE,
                frame.getDefaultCloseOperation());
    assertNotNull(Util.getComponent(frame, CoursesPanel.NAME));
}

// sis.ui.CoursesPanelTest
public void testCreate() {
    CoursesPanel panel = new CoursesPanel();

```

Refactoring

```

JLabel label =
    (JLabel)Util.getComponent(panel, CoursesPanel.LABEL_NAME);
assertEquals(CoursesPanel.LABEL_TEXT, label.getText());
}

```

The new method calls are a bit more unwieldy. One reason is the new need to cast. You'll refactor when or if casting duplication becomes apparent. But the short-term solution is a vast improvement—you've eliminated method-level duplication that otherwise would become rampant in short time.

More Widgets



The SIS application should allow users to add new courses. To support this, CoursesPanel can contain a couple of entry fields in which the user may type the course department and course number. The user should be able to click an “Add” button that adds a new course, created using the entered department and number, to the list.

The test for the view:

```

package sis.ui;

import junit.framework.*;
import javax.swing.*;
import java.awt.*;
import static sis.ui.CoursesPanel.*;

public class CoursesPanelTest extends TestCase {
    public void testCreate() {
        CoursesPanel panel = new CoursesPanel();
        JLabel label =
            (JLabel)Util.getComponent(panel, LABEL_NAME);
        assertEquals(LABEL_TEXT, label.getText());

        JList list =
            (JList)Util.getComponent(panel, COURSES_LIST_NAME);
        assertEquals(0, list.getModel().getSize());

        JButton button =
            (JButton)Util.getComponent(panel, ADD_BUTTON_NAME);
        assertEquals(ADD_BUTTON_TEXT, button.getText());

        JLabel departmentLabel =
            (JLabel)Util.getComponent(panel, DEPARTMENT_LABEL_NAME);
        assertEquals(DEPARTMENT_LABEL_TEXT, departmentLabel.getText());

        JTextField departmentField =
            (JTextField)Util.getComponent(panel, DEPARTMENT_FIELD_NAME);
        assertEquals("", departmentField.getText());
    }
}

```

More Widgets

```

JLabel numberLabel =
    (JLabel)Util.getComponent(panel, NUMBER_LABEL_NAME);
assertEquals(NUMBER_LABEL_TEXT, numberLabel.getText());

JTextField numberField =
    (JTextField)Util.getComponent(panel, NUMBER_FIELD_NAME);
assertEquals("", numberField.getText());
}
}

```

The qualified class constants were getting a bit unwieldy, so I decided to do a static import of the CoursesPanel class. In this case, there is little possibility for confusion about the origin of the class constants.

The role of new component types JButton, JTextField, and JList should be apparent. Code for most of the assertions against the new widgets is similar to the previously coded test for the label.

The assertion against the JList object is different. It proves that the JList on a newly constructed CoursesPanel is empty. You can determine how many elements a JList contains by first obtaining its model, then asking the model for its size. A JList uses a model object to contain its data. You will learn more about list models in the upcoming section, List Models.

The modified view class:

```

package sis.ui;

import javax.swing.*;
import java.awt.*;

public class CoursesPanel extends JPanel {
    static final String NAME = "coursesPanel";
    static final String LABEL_TEXT = "Courses";
    static final String LABEL_NAME = "coursesLabel";
    static final String COURSES_LIST_NAME = "coursesList";
    static final String ADD_BUTTON_TEXT = "Add";
    static final String ADD_BUTTON_NAME = "addButton";
    static final String DEPARTMENT_FIELD_NAME = "deptField";
    static final String NUMBER_FIELD_NAME = "numberField";
    static final String DEPARTMENT_LABEL_NAME = "deptLabel";
    static final String NUMBER_LABEL_NAME = "numberLabel";
    static final String DEPARTMENT_LABEL_TEXT = "Department";
    static final String NUMBER_LABEL_TEXT = "Number";

    public static void main(String[] args) {
        show(new CoursesPanel());
    }

    private static void show(JPanel panel) {
        JFrame frame = new JFrame();
        frame.setSize(300, 200);
    }
}

```

More Widgets

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.getContentPane().add(panel); // 1
frame.setVisible(true);
}

public CoursesPanel() {
    setName(NAME);
    createLayout();
}

private void createLayout() {
    JLabel label = new JLabel(LABEL_TEXT);
    label.setName(LABEL_NAME);

    JList list = new JList();
    list.setName(COURSES_LIST_NAME);

    JButton addButton = new JButton(ADD_BUTTON_TEXT);
    addButton.setName(ADD_BUTTON_NAME);

    int columns = 20;

    JLabel departmentLabel = new JLabel(DEPARTMENT_LABEL_TEXT);
    departmentLabel.setName(DEPARTMENT_LABEL_NAME);

    JTextField departmentField = new JTextField(columns);
    departmentField.setName(DEPARTMENT_FIELD_NAME);

    JLabel numberLabel = new JLabel(NUMBER_LABEL_TEXT);
    numberLabel.setName(NUMBER_LABEL_NAME);

    JTextField numberField = new JTextField(columns);
    numberField.setName(NUMBER_FIELD_NAME);

    add(label);
    add(list);
    add(addButton);
    add(departmentLabel);
    add(departmentField);
    add(numberLabel);
    add(numberField);
}
```

More Widgets

The tests should pass. You should see a window similar to that in Figure 3 when you compile and execute the view class.

The user interface is a mess. You'll rectify this in the upcoming section entitled Layout. Also, nothing that would seem to be a JList appears in the window. One reason is that you have added no elements to the list—it is empty. You'll fix this in time as well.

Even with only seven widgets, you've written a good amount of tedious code to test and build the user interface. Let's see what we can do to tighten it up.

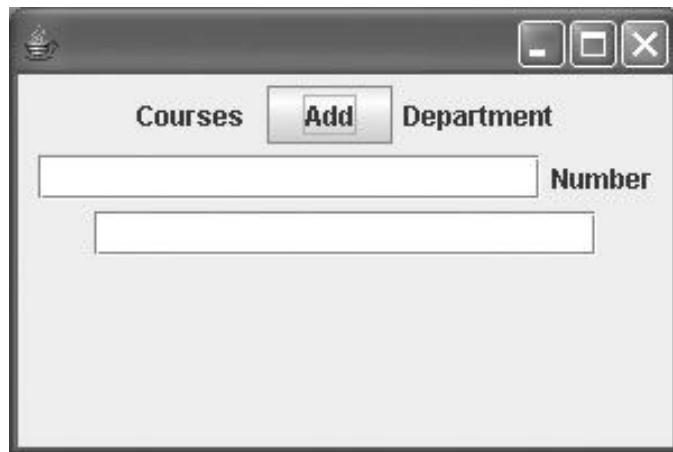


Figure 3

Refactoring

The `getComponent` method calls require casts. For example:

```
JTextField numberField =
    (JTextField)Util.getComponent(panel, NUMBER_FIELD_NAME);
```

Since you must retrieve two text fields from the panel, you have two lines of code with the same cast. This is a form of duplication that you can eliminate, using convenience methods that both eliminate the cast and make the code clearer.

The separate utility class `Util` contains the `getComponent` method. However, it is perhaps more appropriate for `CoursesPanel` to have this responsibility, since a panel is a container of components.

The following listing shows a highly refactored `CoursesPanelTest`. Some of the changes to `CoursesPanelTest` will necessitate changes in `CoursesPanel`—see the listing after this one.

```
package sis.ui;

import junit.framework.*;
import javax.swing.*;
import java.awt.*;
import static sis.ui.CoursesPanel.*;

public class CoursesPanelTest extends TestCase {
    private CoursesPanel panel;

    protected void setUp() {
        panel = new CoursesPanel();
```

Refactoring

```

}

public void testCreate() {
    assertLabelText(COURSES_LABEL_NAME, COURSES_LABEL_TEXT);
    assertEmptyList(COURSES_LIST_NAME);
    assertButtonText(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);
    assertLabelText(DEPARTMENT_LABEL_NAME, DEPARTMENT_LABEL_TEXT);
    assertEmptyField(DEPARTMENT_FIELD_NAME);
    assertLabelText(NUMBER_LABEL_NAME, NUMBER_LABEL_TEXT);
    assertEmptyField(NUMBER_FIELD_NAME);
}

private void assertLabelText(String name, String text) {
    JLabel label = panel.getLabel(name);
    assertEquals(text, label.getText());
}

private void assertEmptyField(String name) {
    JTextField field = panel.getField(name);
    assertEquals("", field.getText());
}

private void assertEmptyList(String name) {
    JList list = panel.getList(name);
    assertEquals(0, list.getModel().getSize());
}

private void assertButtonText(String name, String text) {
    JButton button = panel.getButton(name);
    assertEquals(text, button.getText());
}
}

```

Refactoring

While some of these refactorings eliminate duplication, technically not all of them do. However, those refactorings that do not eliminate duplication help improve readability. First, even though only one list needs to be verified, creating `assertEmptyList` more clearly states the intent. Second, extracting `assertButtonText` allows the entire body of `testCreate` to contain single-line, simple, consistently phrased assertion statements.

Each of the new assertion methods contains two implicit tests. First, if no component with the given name exists within the panel, a `NullPointerException` will be thrown, failing the test. Second, if the component is not of the expected type, a `ClassCastException` will be thrown, also failing the test. If this implicitness bothers you, feel free to add additional assertions (e.g. `assertNotNull`), but I don't view them as necessary.

The new assertion methods are very general purpose. When (or if) you add a second panel to the SIS application, you can refactor the assertion methods to a common test utility class. You might choose to move them to a `junit.framework.TestCase` subclass that all panel test classes inherit.

In most systems, Swing code and associated test code is excessive and repetitive. Take the time up front to do these refactorings. You will significantly minimize the amount of redundancy and overall code in your system.

You can refactor component creation in CoursesPanel in a similar manner. The following listing shows new component creation methods in addition to the get methods required by the test.

```
package sis.ui;

import javax.swing.*;
import java.awt.*;

public class CoursesPanel extends JPanel {
    static final String NAME = "coursesPanel";
    static final String COURSES_LABEL_TEXT = "Courses";
    static final String COURSES_LABEL_NAME = "coursesLabel";
    ...
    private void createLayout() {
        JLabel coursesLabel =
            createLabel(COURSES_LABEL_NAME, COURSES_LABEL_TEXT);
        JList coursesList = createList(COURSES_LIST_NAME);
        JButton addButton =
            createButton(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);

        int columns = 20;
        JLabel departmentLabel =
            createLabel(DEPARTMENT_LABEL_NAME, DEPARTMENT_LABEL_TEXT);
        JTextField departmentField =
            createField(DEPARTMENT_FIELD_NAME, columns);
        JLabel numberLabel =
            createLabel(NUMBER_LABEL_NAME, NUMBER_LABEL_TEXT);
        JTextField numberField =
            createField(NUMBER_FIELD_NAME, columns);

        add(coursesLabel);
        add(coursesList);
        add(addButton);
        add(departmentLabel);
        add(departmentField);
        add(numberLabel);
        add(numberField);
    }

    private JLabel createLabel(String name, String text) {
        JLabel label = new JLabel(text);
        label.setName(name);
        return label;
    }

    private JList createList(String name) {
        JList list = new JList();
    }
}
```

Refactoring

```

        list.setName(name);
        return list;
    }

    private JButton createButton(String name, String text) {
        JButton button = new JButton(text);
        button.setName(name);
        return button;
    }

    private JTextField createField(String name, int columns) {
        JTextField field = new JTextField(columns);
        field.setName(name);
        return field;
    }

    JLabel getLabel(String name) {
        return (JLabel)Util.getComponent(this, name);
    }

    JList getList(String name) {
        return (JList)Util.getComponent(this, name);
    }

    JButton getButton(String name) {
        return (JButton)Util.getComponent(this, name);
    }

    JTextField getField(String name) {
        return (JTextField)Util.getComponent(this, name);
    }
}

```

**Button Blicks
and
Actionlisteners**

Button Clicks and ActionListeners

When you click a Swing button, some action should take place. When you click on the Add button in CoursesPanel, you want a new course to show up in the list of courses. Code to accomplish this will require three steps:

1. Read the contents of the course department and course number text fields.
2. Create a new Course object using the course department and course number text.
3. Put the Course object in the model for the courses list.

Adding a new course with these steps is a mixture of user interface responsibility and business logic. Remember, you want CoursesPanel to be a more or less dumb class that just shows information to the user. Clicking a button

is a controller event, one that you can respond to with an action. The details of the action—the business logic—does not belong in CoursesPanel.

For now, you still need to write a test for the view class. Code in the panel class still needs to tell “someone” to take action when a user clicks the Add button. The panel test will prove that clicking on an Add button triggers some action—an action that has yet to be defined.

You can wire a button click to an action method by using a callback. Java supplies the interface `java.awt.event.ActionListener` for this purpose. To implement `ActionListener`, you code the action method `actionPerformed`. Code in `actionPerformed` should do whatever you want to happen when someone clicks the button.

After defining an `ActionListener` class, you can assign an instance of it to the button. When a user clicks the button, logic in `JButton` calls back to the `actionPerformed` method on the `ActionListener` object.

```
package sis.ui;

import junit.framework.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static sis.ui.CoursesPanel.*;

public class CoursesPanelTest extends TestCase {
    private CoursesPanel panel;
    private boolean wasClicked;

    protected void setUp() {
        panel = new CoursesPanel();
    }
    ...
    public void testAddButtonClick() {
        JButton button = panel.getButton(ADD_BUTTON_NAME);

        wasClicked = false;
        panel.addCourseAddListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                wasClicked = true;
            }
        });

        button.doClick();
        assertTrue(wasClicked);
    }
}
```

**Button Clicks
and
ActionListeners**

You will usually want to implement listeners as anonymous inner classes. Here, the sole job of the `ActionListener` is to ensure that the `actionPerformed`

method gets called when the button is clicked. The JButton class provides the method `doClick` to emulate a user clicking on a button.

CoursesPanel must supply a new method, `addCourseAddListener`. This method simply attaches the ActionListener callback object to the JButton object. Some production client using CoursesPanel will be responsible for defining this callback and passing it to the view. The view remains blissfully ignorant of any business logic.

```
package sis.ui;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class CoursesPanel extends JPanel {
    ...
    private JButton addButton;
    ...
    private void createLayout() {
        ...
        addButton = createButton(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);
        ...
    }
    ...
    void addCourseAddListener(ActionListener listener) {
        addButton.addActionListener(listener);
    }
    ...
}
```

List Models

You must change `addButton` to be a field and not a local variable for this to work.

The method `addCourseAddListener` is a single line of code. For view code other than layout, this is your ideal. If you find yourself putting while loops, if statements, or other convoluted logic in your view class, stop! It likely contains business or application logic that you should represent elsewhere.

List Models

You have gotten the view to represent part of the equation: Tell “someone” to do “something” when the user clicks Add. You know that the result of clicking Add should be that the courses panel shows a new course. You want your view class to treat these as two discrete operations; you’ll implement logic elsewhere to connect the two.

The view should allow client code to pass a Course object and, as a result, display the course object. It doesn't care how the Course object comes into being. Add a new test to CoursesPanelTest:

```
package sis.ui;

import junit.framework.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import sis.studentinfo.*;
import static sis.ui.CoursesPanel.*;

public class CoursesPanelTest extends TestCase {
    ...
    public void testAddCourse() {
        Course course = new Course("ENGL", "101");
        panel.addCourse(course);
        JList list = panel.getList(COURSES_LIST_NAME);
        ListModel model = list.getModel();
        assertEquals("ENGL 101", model.getElementAt(0).toString());
    }
    ...
}
```

The test sends the addCourse message to the CoursePanel. It then extracts the underlying model of the JList. A list model is a collection class that notifies the JList when something in the collection changes.

The JList view needs to show a useful representation of the objects contained in the model. To do so, JList code sends the `toString` message to each object in the model. The final line in the test asserts that the printable representation of the first object in the model is the concatenated department and course number.

List Models

```
package sis.ui;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import sis.studentinfo.*;

public class CoursesPanel extends JPanel {
    ...
    private DefaultListModel coursesModel = new DefaultListModel();
    ...
    private void createLayout() {
        JLabel coursesLabel =
            createLabel(COURSES_LABEL_NAME, COURSES_LABEL_TEXT);

        JList coursesList = createList(COURSES_LIST_NAME, coursesModel);
        ...
    }
}
```

```

    }
    ...
    void addCourse(Course course) {
        coursesModel.addElement(course);
    }
    ...
    private JList createList(String name, ListModel model) {
        JList list = new JList(model);
        list.setName(name);
        return list;
    }
}

```

A list model is a class that implements the interface `javax.swing.ListModel`. For `JList` objects, you will normally use the `ListModel` implementation class `DefaultListModel`. Interestingly, the `ListModel` interface declares no interface methods for adding elements. `DefaultListModel` does (`addElement`). You will want to declare the model reference (`coursesModel`) as being of the implementation type `DefaultListModel` and not of the interface type `ListModel`.

In the `CoursesPanel` method `createList`, you now pass a `ListModel` reference to the `JList` as you construct it. The `addCourse` method takes the `Course` passed to it and stuffs it into the model.

Use of `toString`

I mentioned in Lesson 9 that you should not depend upon the `toString` method for production code. A `toString` method is generally more useful for developer-related debugging activities. A developer might need to change `Course` output from the form:

List Models

`ENGL 101`

to something like:

`[Course:ENGL,101]`

The modified string would be inappropriate for the `CoursesPanel` user interface view. Another conflict can arise if two different views must show `Course` information in different formats.

In either case, you can and should introduce a display adapter class that wraps each course and provides the `toString` definition needed. You store these adapter objects in the `JList` model.



The user now wants to see a hyphen between each department and course number in the list. Modify the test to require this new display format:

```

public void testAddCourse() {
    Course course = new Course("ENGL", "101");
    panel.addCourse(course);
    JList list = panel.getList(COURSES_LIST_NAME);

    ListModel model = list.getModel();
    assertEquals("ENGL-101", model.getElementAt(0).toString());
}

```

A simple implementation of the adapter class is to have it subclass Course and override the definition of `toString`.

```

package sis.ui;

import sis.studentinfo.*;

class CourseDisplayAdapter extends Course {
    CourseDisplayAdapter(Course course) {
        super(course.getDepartment(), course.getNumber());
    }

    @Override
    public String toString() {
        return String.format(
            "%s-%s", getDepartment(), getNumber());
    }
}

```

The CoursesPanel `addCourse` method must change:

```

void addCourse(Course course) {
    coursesModel.addElement(new CourseDisplayAdapter(course));
}

```

Later, when you need to write code to obtain a Course selected in the list box, you'll need to extract the target Course from its adapter.

The Application

The Application

Now that the view is in place, you can specify how the application should use it. It's time to tie things together.² Here's a new SisTest method, `testAddCourse`.

²Depending on your mindset, you might have found it easier to have started building from the application and driven down into the view. Even while I was constructing the view, I had in mind how I wanted the application to tie things together.

```

package sis.ui;

import junit.framework.*;
import javax.swing.*;
import java.awt.*;
import sis.studentinfo.*;

public class SisTest extends TestCase {
    ...
    public void testAddCourse() {
        CoursesPanel panel =
            (CoursesPanel)Util.getComponent(frame, CoursesPanel.NAME);
        panel.setText(CoursesPanel.DEPARTMENT_FIELD_NAME, "MATH");
        panel.setText(CoursesPanel.NUMBER_FIELD_NAME, "300");

        JButton button = panel.getButton(CoursesPanel.ADD_BUTTON_NAME);
        button.doClick();

        Course course = panel.getCourse(0);
        assertEquals("MATH", course.getDepartment());
        assertEquals("300", course.getNumber());
    }
}

```

The test drives the application from the perspective of an end user. It's almost an acceptance test.

First, the test sets values into the department and course number fields. It then emulates a click of the Add button using the button method `click`. In order to ensure that the application is behaving correctly, the tests asks the embedded CoursesPanel to return the first Course in its list.

You'll need to add a couple of methods to CoursesPanel to support the test:

```

package sis.ui;
...
public class CoursesPanel extends JPanel {
    ...
    Course getCourse(int index) {
        Course adapter =
            (CourseDisplayAdapter)coursesModel.getElementAt(index);
        return adapter;
    }
    ...
    void setText(String textFieldName, String text) {
        getField(textFieldName).setText(text);
    }
}

```

Since a CourseDisplayAdapter extends from Course, you can assign the extracted adapter object to a Course reference to return.

The Application

Code changes to Sis (including a few minor refactorings):

```
package sis.ui;

import javax.swing.*;
import java.awt.event.*;
import sis.studentinfo.*;

public class Sis {
    ...
    private CoursesPanel panel;
    ...

    public Sis() {
        initialize();
    }

    private void initialize() {
        createCoursesPanel();

        frame.setSize(WIDTH, HEIGHT);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(panel);
    }

    ...
    void createCoursesPanel() {
        panel = new CoursesPanel();
        panel.addCourseAddListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    addCourse();
                }
            }
        );
    }
}

private void addCourse() {
    Course course =
        new Course(
            panel.getText(CoursesPanel.DEPARTMENT_FIELD_NAME),
            panel.getText(CoursesPanel.NUMBER_FIELD_NAME));
    panel.addCourse(course);
}
```

The Application

The Sis class ties together the action listener and the ability to add a course to the panel. Most of the code should look familiar—it is the client code you built in tests for CoursesPanel.

You'll need to add the getText method to CoursesPanel:

```
String getText(String textFieldName) {
    return getField(textFieldName).getText();
}
```

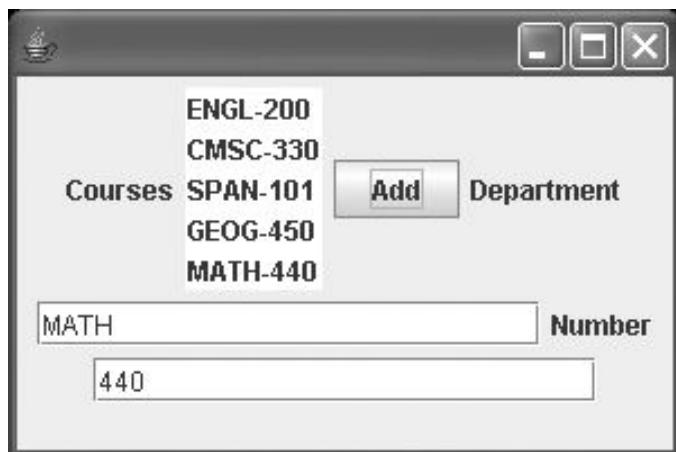


Figure 4

You can now run Sis as a stand-alone application and experiment with adding courses. The screen shot in Figure 4 shows Sis after a user has entered five courses. It's still a mess!

Layout

Layout

The user interface for the Sis application is so poorly laid out that it's confusing to the end user. The problem is that by default, Swing lays out components to flow from left to right, in the order you add them to a container. When insufficient room remains to place a component on a "line," Swing wraps it, just like a word in a word processor. The component appears to the left side of the panel below the current line.

In Figure 4, the first line consists of four widgets: the "Courses" label, the list of courses, the Add button, and the "Department" label. The second line consists of the department field, the "Number" label, and the course number field.

Resize the window and make it as wide as your screen. Swing will redraw the widgets. If your screen is wide enough, all of the components should flow from left to right on a single line.

Swing provides several alternate layout mechanisms, each in a separate class, to help you produce aesthetically pleasing user interfaces. Swing refers to these classes as *layout managers*. You can associate a different layout man-

ager with each container. The default layout manager, `java.awt.FlowLayout`, is not very useful if you want to create a professional-looking user interface.

Getting a view to look “just right” is an incremental, taxing exercise. You will find that mixing and matching layouts is an easier strategy than trying to stick to a single layout for a complex view.

An alternative to hand-coding layouts is to use a tool. Many IDEs provide GUI (view) composition tools that allow you to visually edit a layout. Using a tool can reduce the tedium of trying to get a user interface to be perfect.

You will learn to use a few of the more-significant layout mechanisms in an attempt to make `CoursesPanel` look good. Little of this work requires you to test first. Instead, you should test last. Make a small change, compile, run your tests, execute `CoursesPanel` as a stand-alone application, view the results, react!

Francis Katasani

GridLayout

You’ll start with a simply understood but usually inappropriate layout, `GridLayout`. A `GridLayout` divides the container into equal-sized rectangles based on the number of rows and columns you specify. As you add components to the container, the `GridLayout` puts each in a cell (rectangle), moving from left to right, top to bottom (by default). The layout manager resizes each component to fit its cell.

Make the following changes in `CoursesPanel`:

```
private void createLayout() {  
    JLabel coursesLabel =  
        createLabel(COURSES_LABEL_NAME, COURSES_LABEL_TEXT);  
  
    JList coursesList = createList(COURSES_LIST_NAME, coursesModel);  
    addButton =  
        createButton(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);  
    int columns = 20;  
    JLabel departmentLabel =  
        createLabel(DEPARTMENT_LABEL_NAME, DEPARTMENT_LABEL_TEXT);  
    JTextField departmentField =  
        createField(DEPARTMENT_FIELD_NAME, columns);  
    JLabel numberLabel =  
        createLabel(NUMBER_LABEL_NAME, NUMBER_LABEL_TEXT);  
    JTextField numberField =  
        createField(NUMBER_FIELD_NAME, columns);  
  
    int rows = 4;  
    int cols = 2;  
    setLayout(new GridLayout(rows, cols));
```

Layout

```
    add(coursesLabel);
    add(coursesList);
    add(addButton);
    add(new JPanel());
    add(departmentLabel);
    add(departmentField);
    add(numberLabel);
    add(numberField);
}
```

You assign a layout manager to the panel by sending it the message `setLayout`. In `createLayout`, you send the message `setLayout` to the `CoursesPanel` object, passing it a `GridLayout` instance.

The result of executing `CoursesPanel` using the `GridLayout` is shown in Figure 5.

The `coursesLabel` ends up in the upper left rectangle. The `coursesList`, the second component to be added, is in the upper right rectangle. The Add button drops down to the second row of rectangles, and is followed by an empty `JPanel` to fill the next rectangle. Each of the final two rows displays a label and its corresponding field.

Since each rectangle must be the same size, `GridLayout` doesn't have a lot of applicability in organizing "typical" interfaces with lots of buttons, fields, lists, and labels. It does have applicability if, for example, you are presenting a suite of icons to the end user. `GridLayout` does contain additional methods to improve upon its look, but you will usually want to use a more-sophisticated layout manager.

Layout



Figure 5

BorderLayout

BorderLayout is a simple but effective layout manager. BorderLayout allows you to place up to five components within the container: one each at the compass points north, east, south, and west and one to fill the remainder, or center (see Figure 6).

For CoursesPanel, you will replace the GridLayout with a BorderLayout. Your BorderLayout will use three of the available areas: north, to contain the “Courses” Label; center, to contain the list of courses; and south, to contain the remainder of the widgets. You will organize the southern, or “bottom,” widgets in a subpanel that uses a separate layout manager.

The `createLayout` method already is overly long, at close to 30 lines. CoursesPanel is a simple interface so far. Imagine a sophisticated panel with several dozen widgets. Unfortunately, it is common to encounter Swing code that does all of the requisite initialization and layout in a single method. Developers create panels within panels, they create and initialize components and place them in panels, they create layouts, and so on.

A more effective code composition is to extract the creation of each panel into a separate method. This not only makes the code far more readable but also provides more flexibility in rearranging the layout. I’ve refactored the code in CoursesPanel to reflect this cleaner organization.

```
private void createLayout() {
    JLabel coursesLabel =
        createLabel(COURSES_LABEL_NAME, COURSES_LABEL_TEXT);

    JList coursesList = createList(COURSES_LIST_NAME, coursesModel);
```

Layout

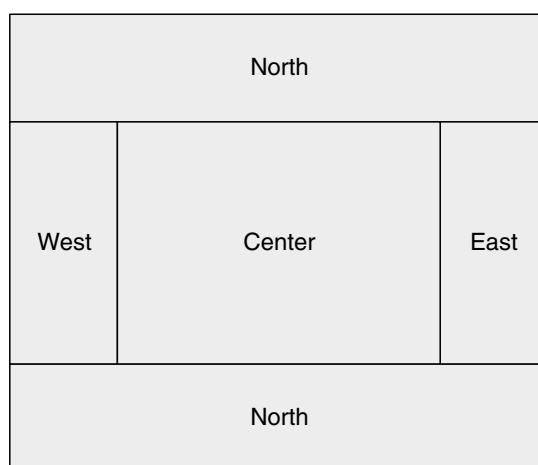


Figure 6 A Border Layout Configuration

```

setLayout(new BorderLayout());

add(coursesLabel, BorderLayout.NORTH);
add(coursesList, BorderLayout.CENTER);
add(createBottomPanel(), BorderLayout.SOUTH);
}

JPanel createBottomPanel() {
    addButton = createButton(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);

    JPanel panel = new JPanel();
    panel.setLayout(new BorderLayout());

    panel.add(addButton, BorderLayout.NORTH);
    panel.add(createFieldsPanel(), BorderLayout.SOUTH);

    return panel;
}

JPanel createFieldsPanel() {
    int columns = 20;
    JLabel departmentLabel =
        createLabel(DEPARTMENT_LABEL_NAME, DEPARTMENT_LABEL_TEXT);
    JTextField departmentField =
        createField(DEPARTMENT_FIELD_NAME, columns);
    JLabel numberLabel =
        createLabel(NUMBER_LABEL_NAME, NUMBER_LABEL_TEXT);
    JTextField numberField =
        createField(NUMBER_FIELD_NAME, columns);

    int rows = 2;
    int cols = 2;

    JPanel panel = new JPanel();
    panel.setLayout(new GridLayout(rows, cols));
    panel.add(departmentLabel);
    panel.add(departmentField);

    panel.add(numberLabel);
    panel.add(numberField);

    return panel;
}

```

Layout

The code in the CoursesPanel constructor sets its layout to a new instance of BorderLayout. It puts the label on the north (top) side of the panel and the list in the center of the panel. It puts the result of the method createBottomPanel, another JPanel, on the south (bottom) side of the panel (see Figure 7). The benefit of putting the list in the center is that it will expand as the frame window expands. The other widgets retain their original size.



Figure 7

The code in `createBottomPanel` creates a JPanel that also uses a BorderLayout to organize its components. It places the Add button north and the resulting JPanel from `createFieldsPanel` south. The `createFieldsPanel` uses a GridLayout to organize the department and course number labels and fields. The result (Figure 7) is a considerable improvement but is still not good enough. Again, make sure you experiment with resizing the frame window to see how the layout reacts.

A Test Problem

If you rerun your tests, you now get three NullPointerException errors. How can that be, since you neither changed logic nor added/removed any components?

When you investigate the stack trace for the NullPointerException, you should discover that some of the `get` methods to extract a component from a container are failing. The problem is that the Util method `getComponent` only looks at components directly embedded within a container. Your layout code now embeds containers within containers (JPanels within JPanels). The code in `getComponent` ignores Components that have been added to subpanels.

The Util class doesn't have any tests associated with it. At this point, to help fix the problem and enhance your test coverage, you need to add appropriate tests. `UtilTest` contains three tests that should cover most expected circumstances:

```
package sis.ui;

import junit.framework.*;
import javax.swing.*;
import java.awt.*;
```

Layout

```

public class UtilTest extends TestCase {
    private JPanel panel;

    protected void setUp() {
        panel = new JPanel();
    }

    public void testNotFound() {
        assertNull(Util.getComponent(panel, "abc"));
    }

    public void testDirectlyEmbeddedComponent() {
        final String name = "a";
        Component component = new JLabel("x");
        component.setName(name);
        panel.add(component);
        assertEquals(component, Util.getComponent(panel, name));
    }

    public void testSubcomponent() {
        final String name = "a";
        Component component = new JLabel("x");
        component.setName(name);

        JPanel subpanel = new JPanel();
        subpanel.add(component);

        panel.add(subpanel);

        assertEquals(component, Util.getComponent(panel, name));
    }
}

```

Layout

The third test, `testSubcomponent`, should fail for the same reason your other tests are failing. To fix the problem, you will need to modify `getComponent`. For each component in a container, you will need to determine whether or not that component is a container (using `instanceof`). If so, you will need to traverse all of that subcontainer's components, repeating the same process for each. The most effective way to accomplish this is to make recursive calls to `getComponent`.

```

static Component getComponent(Container container, String name) {
    for (Component component: container.getComponents()) {
        if (name.equals(component.getName()))
            return component;
        if (component instanceof Container) {
            Container subcontainer = (Container)component;
            Component subcomponent = getComponent(subcontainer, name);

```

```

        if (subcomponent != null)
            return subcomponent;
    }
}
return null;
}

```

Your tests should all pass with this change.

BoxLayout

The `BoxLayout` class allows you to lay your components out on either a horizontal or vertical axis. Components are not wrapped when you resize the container; also, components do not grow to fill any area. The bottom panel, which must position an Add button and the fields subpanel vertically, one atop the other, is an ideal candidate for `BoxLayout`.

```

JPanel createBottomPanel() {
    addButton = createButton(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);

    JPanel panel = new JPanel();
    panel.setLayout(new BoxLayout(panel, BoxLayout.PAGE_AXIS));

    panel.add(Box.createRigidArea(new Dimension(0, 6)));
    addButton.setAlignmentX(Component.CENTER_ALIGNMENT);
    panel.add(addButton);
    panel.add(Box.createRigidArea(new Dimension(0, 6)));
    panel.add(createFieldsPanel());

    panel.setBorder(BorderFactory.createEmptyBorder(8, 8, 8, 8));

    return panel;
}

```

Layout

You must pass an instance of the panel to the constructor of `BoxPanel` and a constant indicating the direction in which to lay out components. The constant `PAGE_AXIS` by default represents a top-to-bottom, or vertical, orientation. The other option is `LINE_AXIS`, which represents horizontal orientation by default.⁴

You can create invisible “rigid areas” to separate components with whitespace. These rigid areas retain a fixed size even when the container is resized.

⁴You can change the orientation by sending `applyComponentOrientation` to the container. Older versions of `BoxLayout` supported only explicit `X_AXIS` and `Y_AXIS` constants. The newer constants allow for dynamic reorganization, perhaps to support internationalization needs.



Figure 8 Using *BoxLayout*

The class method `createRigidArea` takes a `Dimension` object as a parameter. A `Dimension` is a width (0 in this example) by a height (6).

You may want to align each component with respect to the axis. In the example, you center the Add button around the vertical axis by sending it the message `setAlignmentX` with the parameter `Component.CENTER_ALIGNMENT`.

A final tweak is to supply an invisible border around the entire panel. The `BorderFactory` class can provide several types of borders that you can pass to the panel's `setBorder` method. Creating an empty border requires four parameters, each representing the width of the spacing from the outside edge of the panel. Other borders you can create include beveled borders, line borders, compound borders, etched borders, matte borders, raised beveled borders, and titled borders. Take a look at the Java API documentation and experiment with the effects that using different borders produces.

The code now produces an effective, but not quite perfect, layout. See Figure 8.

GridLayout

GridLayout

To fine-tune a layout, you will be forced to work with `GridBagLayout`, a highly configurable but more complex layout manager. `GridBagLayout` is similar to `GridLayout` in that it lets you organize components in a grid of rectangles. However, `GridBagLayout` gives you far more control. First, each rectangle is not fixed in size—it is generally sized according to the default, or “preferred” size of the component it contains. Components can span multiple rows or columns.

The modified `createFieldsPanel` code shows how to lay out labels and fields using a `GridBagLayout`. (The code in the method presumes that you have statically imported `java.awt.GridBagConstraints.*`.)

```

JPanel createFieldsPanel() {
    GridBagLayout layout = new GridBagLayout();

    JPanel panel = new JPanel(layout);
    int columns = 20;
    JLabel departmentLabel =
        createLabel(DEPARTMENT_LABEL_NAME, DEPARTMENT_LABEL_TEXT);
    JTextField departmentField =
        createField(DEPARTMENT_FIELD_NAME, columns);
    JLabel numberLabel =
        createLabel(NUMBER_LABEL_NAME, NUMBER_LABEL_TEXT);
    JTextField numberField =
        createField(NUMBER_FIELD_NAME, columns);

    layout.setConstraints(departmentLabel,
        new GridBagConstraints(
            0, 0, // x, y
            1, 1, // gridwidth, gridheight
            40, 1, // weightx, weighty
            LINE_END, // anchor
            NONE, // fill
            new Insets(3, 3, 3, 3), // top-left-bottom-right
            0, 0)); // padx, ipady
    layout.setConstraints(departmentField,
        new GridBagConstraints(1, 0, 2, 1, 60, 1,
            CENTER, HORIZONTAL,
            new Insets(3, 3, 3, 3), 0, 0));
    layout.setConstraints(numberLabel,
        new GridBagConstraints(0, 1, 1, 1, 40, 1,
            LINE_END, NONE,
            new Insets(3, 3, 3, 3), 0, 0));
    layout.setConstraints(numberField,
        new GridBagConstraints(1, 1, 2, 1, 60, 1,
            CENTER, HORIZONTAL,
            new Insets(3, 3, 3, 3), 0, 0));

    panel.add(departmentLabel);
    panel.add(departmentField);
    panel.add(numberLabel);
    panel.add(numberField);

    return panel;
}

```

GridBagLayout

After creating a `GridBagLayout` and setting it into the panel, you call the method `setConstraints` on the layout for each widget to add. The `setConstraints` method takes two parameters: a Component object and a `GridBagConstraints`

object. A `GridBagConstraints` object contains several constraints for the Component object. This table very briefly summarizes the constraints (see the API documentation for complete details):

<code>gridx/gridy</code>	The cell at which to begin drawing the component. The upper left cell is 0, 0.
<code>gridwidth/gridheight</code>	The amount of rows or columns the component should span. The default value is 1 for each, meaning that a component takes a single cell.
<code>weightx/weighty</code>	The weighting constraint is used to determine how much of the additional space is allocated to a component if space remains after laying out all the components in a row or column.
<code>anchor</code>	The anchor constraint is used to determine how it is placed within the cell if a component is smaller than its display area. The default is <code>GridBagConstraints.CENTER</code> .
<code>fill</code>	The fill constraint specifies how the component should grow to fill the display area if a component is smaller than its display area. Values are <code>NONE</code> (don't resize), <code>HORIZONTAL</code> , <code>VERTICAL</code> , and <code>BOTH</code> .
<code>insets</code>	Uses an <code>Insets</code> object to specify the space between a component and the edges of its display area.
<code>ipadx/ipady</code>	Specifies how much space to add to the minimum size of a component.

GridLayout

Each of the fields in `GridBagConstraints` is `public`. You can construct a `GridBagConstraints` object with no parameters, then set individual fields as necessary. Or, as in the listing for `createFieldsPanel`, you can specify every possible constraint using the alternate `GridBagConstraints` constructor.

The best tactic is to sketch, on paper or whiteboard, a grid representing how you want the output to look. Use the `gridx/gridy` and `gridwidth/gridheight` constraints to determine the relative sizes and positions of the components. Then concentrate on the `anchor` and `fill` aspects of each component. Code the layout to these sketched specifications and modify if necessary. You can then experiment with the `insets` and `weightx/weighty` constraints (and occasionally the `ipadx/ipady` constraints) to tweak the spacing between components.

Obviously there is a lot of redundancy in `createFieldsPanel`. The following code is modestly refactored.

```

JPanel createFieldsPanel() {
    GridBagLayout layout = new GridBagLayout();

    JPanel panel = new JPanel(layout);
    int columns = 20;

    addField(panel, layout, 0,
        DEPARTMENT_LABEL_NAME, DEPARTMENT_LABEL_TEXT,
        DEPARTMENT_FIELD_NAME, columns);
    addField(panel, layout, 1,
        NUMBER_LABEL_NAME, NUMBER_LABEL_TEXT,
        NUMBER_FIELD_NAME, columns);

    return panel;
}

private void addField(
    JPanel panel, GridBagLayout layout, int row,
    String labelName, String labelText,
    String fieldName, int fieldColumns) {

    JLabel label = createLabel(labelName, labelText);
    JTextField field = createField(fieldName, fieldColumns);

    Insets insets = new Insets(3, 3, 3, 3); // top-left-bottom-right
    layout.setConstraints(label,
        new GridBagConstraints(
            0, row, // x, y
            1, 1, // gridwidth, gridheight
            40, 1, // weightx, weighty
            LINE_END, // anchor
            NONE, // fill
            insets, 0, 0)); // padx, ipady
    layout.setConstraints(field,
        new GridBagConstraints(1, row,
            2, 1, 60, 1, CENTER, HORIZONTAL,
            insets, 0, 0));

    panel.add(label);
    panel.add(field);
}

```

GridBagLayout

The onerous nature of such code should drive you in the direction of extreme refactoring. Consider replacing repetition in the construction of `GridBagConstraints` objects by using a simplified utility constructor. If you need to represent more than a couple of fields and associated labels, consider representing each pair using a data class. You can then represent the entire set of fields in a table, iterating through it to create the layout.

Figure 9 shows a layout that is getting close to being acceptable. Resize it to see how the field components fill to their display area.



Figure 9 Using *GridBagLayout*

As of Java 1.4, Sun introduced the `SpringLayout` class. This layout manager is primarily designed for the use of GUI composition tools. The basic concept of `SpringLayout` is that you define a layout by tying the edges of components together using constraints known as springs.

In `CoursesPanel`, you might create a spring to attach the west (left) edge of the department text field to the east (right) edge of the department label. The spring object attaching the two components is a fixed five pixels in size. Another spring might attach the east edge of the department text field to the east side of the panel itself, also separated by a spring of fixed length. As the panel grows in width, the department text field would grow accordingly.

Creating a `SpringLayout` by hand is fairly easy for a panel with only a few components. It can also be incredibly frustrating and difficult for a more complex layout. In most cases, you will want to leave the job to a layout tool.

Moving Forward

Moving Forward

You've only scratched the surface of Swing! In the next chapter, you'll tighten up the look and the feel of `CoursesPanel` with some fine-tuning. I'll then run through a few more generally useful Swing topics.

Additional Lesson II

Swing, Part 2

In the last lesson you learned how to build a basic Swing application, using panels, labels, buttons, text fields, and lists. You also learned how to use Swing's layout managers for enhancing the look of your user interface (UI).

In this chapter, you will learn about:

- Scroll panes
- Borders
- Setting text in the title bar
- Icons
- Keyboard support
- Button mnemonics
- Required fields
- Keyboard listeners
- The Swing Robot class
- Field edits and document filters
- Formatted text fields
- Tables
- Mouse listeners
- Cursors
- SwingUtilities methods `invokeAndWait` and `invokeLater`

Swing, Part 2

Miscellaneous Aesthetics

In this section you'll learn how to enhance the look of the existing CoursesPanel.

JScrollPane

If you add more than a few courses using sis.ui.Sis, you'll note that the JList shows only some of them. To fix this problem, you can wrap the JList in a scroll pane. The scroll pane acts as a viewport onto the list. When the list's model contains more information than can be displayed given the current JList size, the scroll pane draws scroll bars. The scroll bars allow you to move the viewport either horizontally or vertically to reveal hidden information.

The bold code in this listing for the CoursesPanel adds a scroll pane. You can specify whether you want to show scroll bars always or only as needed using either `setVerticalScrollBarPolicy` or `setHorizontalScrollBarPolicy`. I find it more aesthetically pleasing to always show the vertical scroll bar.

```
private void createLayout() {
    JLabel coursesLabel =
        createLabel(COURSES_LABEL_NAME, COURSES_LABEL_TEXT);

    JList coursesList = createList(COURSES_LIST_NAME, coursesModel);
    JScrollPane coursesScroll = new JScrollPane(coursesList);
    coursesScroll.setVerticalScrollBarPolicy(
        ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);

    setLayout(new BorderLayout());

    add(coursesLabel, BorderLayout.NORTH);
    add(coursesScroll, BorderLayout.CENTER);
    add(createBottomPanel(), BorderLayout.SOUTH);
}
```

Miscellaneous Aesthetics

Borders

The courses list and associated labels directly abut the edge of the panel. You can use a *border* to create a buffer area between the edges of the panel and any of its components.

```
private void createLayout() {
    JLabel coursesLabel =
        createLabel(COURSES_LABEL_NAME, COURSES_LABEL_TEXT);

    JList coursesList = createList(COURSES_LIST_NAME, coursesModel);
    JScrollPane coursesScroll = new JScrollPane(coursesList);
```

```

coursesScroll.setVerticalScrollBarPolicy(
    ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);

setLayout(new BorderLayout());

final int pad = 6;
setBorder(BorderFactory.createEmptyBorder(pad, pad, pad, pad));

add(coursesLabel, BorderLayout.NORTH);
add(coursesScroll, BorderLayout.CENTER);
add(createBottomPanel(), BorderLayout.SOUTH);
}

```

You can use the BorderFactory class to create one of several different border types. Most of the borders are for decorative purposes; the empty border is the exception. You can also combine borders using createCompoundBorder. The reworked createLayout method demonstrates the use of a few different border types.

```

private void createLayout() {
    JList coursesList = createList(COURSES_LIST_NAME, coursesModel);
    JScrollPane coursesScroll = new JScrollPane(coursesList);
    coursesScroll.setVerticalScrollBarPolicy(
        ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);

   .setLayout(new BorderLayout());

    final int pad = 6;
    Border emptyBorder =
        BorderFactory.createEmptyBorder(pad, pad, pad, pad);
    Border bevelBorder =
        BorderFactory.createBevelBorder(BevelBorder.RAISED);
    Border titledBorder =
        BorderFactory.createTitledBorder(bevelBorder, COURSES_LABEL_TEXT);
    setBorder(BorderFactory.createCompoundBorder(emptyBorder,
        titledBorder));

    add(coursesScroll, BorderLayout.CENTER);
    add(createBottomPanel(), BorderLayout.SOUTH);
}

```

Miscellaneous
Aesthetics

The use of a titled border eliminates the need for a separate JLabel to represent the “Courses:” text. The elimination of the JLabel will break testCreate in CoursesPanelTest—did you remember to make this change by testing first?

Adding a Title

No text appears in the SIS frame window’s titlebar. Rectify this by updating testCreate in SisTest:

```
public void testCreate() {
    final double tolerance = 0.05;
    assertEquals(Sis.HEIGHT, frame.getSize().getHeight(), tolerance);
    assertEquals(Sis.WIDTH, frame.getSize().getWidth(), tolerance);
    assertEquals(JFrame.EXIT_ON_CLOSE,
                frame.getDefaultCloseOperation());
    assertNotNull(Util.getComponent(frame, CoursesPanel.NAME));
    assertEquals(Sis.COURSES_TITLE, frame.getTitle());
}
```

JFrame supplies constructors that allow you to pass in titlebar text. The code in Sis:

```
public class Sis {
    ...
    static final String COURSES_TITLE = "Course Listing";
    private JFrame frame = new JFrame(COURSES_TITLE);
    ...
}
```

Icons

The final aesthetic element you'll add is an icon for the window. By default, you get the cup-o-Java icon, which appears as a mini-icon in the titlebar and when you minimize the window. Since the icon is part of the titlebar, it falls under the control of the frame window.

A test can simply ask for the icon from a frame by sending it the message `getIconImage`. The method, implemented in `java.awt.Frame`, returns an object of type `java.awt.Image`. The code in `testCreate` asserts that the icon retrieved from the SIS frame is the same as one explicitly loaded by name. Both the test and the `sis.ui.Sis` code will use a common utility method to load the image: `ImageUtil.create`.

Miscellaneous Aesthetics

```
public void testCreate() {
    final double tolerance = 0.05;
    assertEquals(Sis.HEIGHT, frame.getSize().getHeight(), tolerance);
    assertEquals(Sis.WIDTH, frame.getSize().getWidth(), tolerance);
    assertEquals(JFrame.EXIT_ON_CLOSE,
                frame.getDefaultCloseOperation());
    assertNotNull(Util.getComponent(frame, CoursesPanel.NAME));
    assertEquals(Sis.COURSES_TITLE, frame.getTitle());

    Image image = frame.getIconImage();
    assertEquals(image, ImageUtil.create("/images/courses.gif"));
}
```

Create the class `ImageUtilTest` in the `sis.util` package. There are several ways to write a test against the `create` method. The best way would be to dynamically generate an image using a collection of pixels and write it out to

disk. After the `create` method loaded the image, you could assert that the pixels in the loaded image were the same as the original pixel collection. Unfortunately, this is an involved solution, one that is best solved using an API that eases the dynamic creation of images.

A simpler technique is to allow the test to assume that a proper image already exists on disk under a recognized filename. The test can then simply assert that the image load was successful by ensuring that the loaded image is not `null`. The image must appear on the classpath.¹

```
package sis.util;

import junit.framework.*;
import java.awt.*;

public class ImageUtilTest extends TestCase {
    public void testLoadImage() {
        assertNull(ImageUtil.create("/images/bogusFilename.gif"));
        assertNotNull(ImageUtil.create("/images/courses.gif"));
    }
}
```

This may seem weak, but it will provide an effective solution for now. You must ensure that the image `courses.gif` continues to stay in existence throughout the lifetime of the project. Instead of using an image file associated with the SIS project, you might consider creating an image explicitly used for testing purposes.

Java provides several different ways to load and manipulate images. You will use the most direct.

```
package sis.util;

import javax.swing.*;
import java.awt.*;

public class ImageUtil {
    public static Image create(String path) {
        java.net.URL imageURL = ImageUtil.class.getResource(path);
        if (imageURL == null)
            return null;
        return new ImageIcon(imageURL).getImage();
    }
}
```

Miscellaneous
Aesthetics

¹I've modified the `build.xml` Ant script for this section to copy any file in `src/images` into `classes/images` each time a compile takes place. This allows you to quickly remove the entire contents of the `classes` directory without having to worry about retaining any images.

The class `Class` defines the method `getResource`. This method allows you to locate resources, including files, regardless of whether the application is loaded from a JAR file, individual class files, or some other source such as the Internet.² The result of calling `getResource` is a URL—the unique address of the resource.

Once you have a proper URL, you can pass it to the `ImageIcon` constructor to obtain an `ImageIcon` object. You can use `ImageIcon` objects for various purposes, such as decorating buttons or labels. Since the `Frame` class requires an `Image` object, not an `ImageIcon`, here you use the `getImage` method to produce the return value for `create`.

Note that the image filename passed to `getResource` is `/images/courses.gif`—a path that uses a leading slash. This indicates that the resource should be located starting from the root of each classpath entry. Thus, if you execute classes from the directory `c:\swing2\classes`, you should put `courses.gif` in `c:\swing2\classes\images`. If you execute classes by loading them from a JAR file, it should contain `courses.gif` with a relative path of `images`.

Here are the changes to the `initialize` method in the `Sis` class.

```
private void initialize() {
    createCoursesPanel();

    Image image = ImageUtil.create("/images/courses.gif");
    frame.setIconImage(image);

    frame.setSize(WIDTH, HEIGHT);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(panel);
}
```

You now have an acceptably pleasing interface—at least from a visual standpoint. The revised layout is shown in Figure 1.

Feel

Feel

The visual appearance of the interface is important, but more important is its “feel.” The feel of an application is what a user experiences as he or she interacts with it. Examples of elements relevant to the feel of an application include:

²Technically, the image is loaded using the class loader as that of the `Class` object on which you call `getResource`. For applications that do not use a custom class loader, using the class loader of the `ImageUtil` class will work just fine.

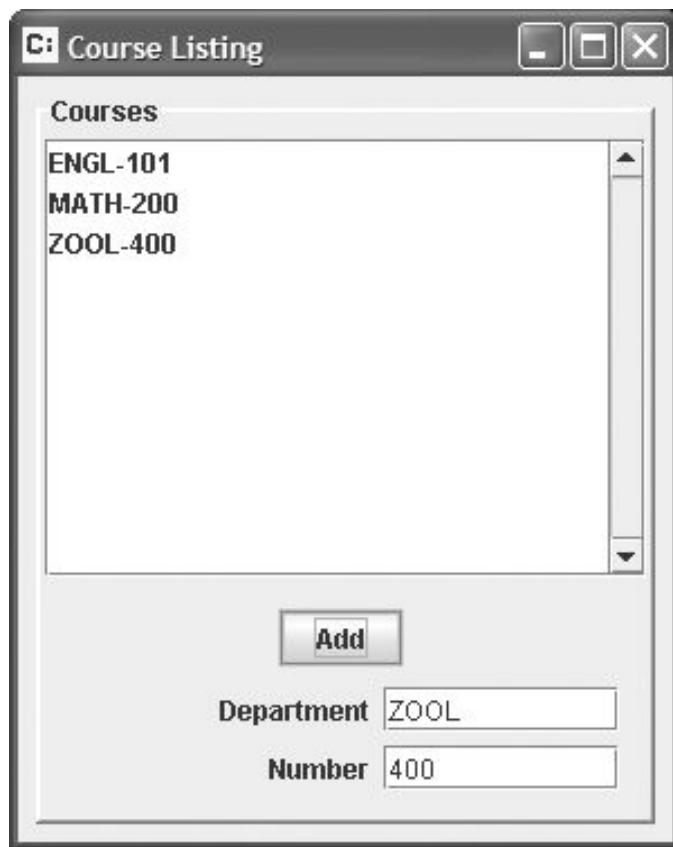


Figure 1 A Good Look

- ability to effect all behaviors using either the keyboard or the mouse
- tabbing sequence—can the user visit all text fields in the proper order and are irrelevant items omitted from the tab sequence?
- field limits—is the user restricted from entering too many characters?
- field constraints—is the user restricted from entering inappropriate data into a field?
- button activation—are buttons deactivated when their use is inappropriate?

Feel

It is possible to address both the look and feel at the same time. In the last section, you decorated the list of courses with a scroll pane. This improved the look of the interface and at the same time provided the “feel” to allow a user to scroll the list of courses when necessary.

Keyboard Support

One of the primary rules of GUIs is that a user must be able to completely control the application using either the keyboard or the mouse. Exceptions exist. Using the keyboard to draw a polygon is ineffective, as is entering characters using the mouse. (Both are of course possible.) In such cases, the application developer often chooses to bypass the rule. But in most cases it is extremely inconsiderate to ignore the needs of a keyboard-only or mouse-only user.

By default, Java supplies most of the necessary support for dual keyboard and mouse control. For example, you can activate a button by either clicking on it with the mouse or tabbing to it and pressing the space bar.

Button Mnemonics

An additional way you can activate a button is using an Alt-key combination. You combine pressing the Alt key with another key. The other key is typically a letter or number that appears in the button text. You refer to this key as a *mnemonic* (technically, a device to help you remember something; in Java, it's simply a single-letter shortcut). An appropriate mnemonic for the Add button is the letter A.

The mnemonic is a view class element. The specifications for the mnemonic remain constant with the view's appearance. Therefore the more appropriate place to test and manage the mnemonic is in the view class.

In CoursesPanelTest:

```
public void testCreate() {
    assertEquals(COURSES_LIST_NAME);
    assertEquals(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);
    assertEquals(DEPARTMENT_LABEL_NAME, DEPARTMENT_LABEL_TEXT);
    assertEquals(DEPARTMENT_FIELD_NAME);
    assertEquals(NUMBER_LABEL_NAME, NUMBER_LABEL_TEXT);
    assertEquals(NUMBER_FIELD_NAME);

    JButton button = panel.getButton(ADD_BUTTON_NAME);
    assertEquals(ADD_BUTTON_MNEMONIC, button.getMnemonic());
}
```

Feel

In CoursesPanel:

```
public class CoursesPanel extends JPanel {
    ...
    static final char ADD_BUTTON_MNEMONIC = 'A';
    ...
    JPanel createBottomPanel() {
        addButton = createButton(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);
```

```
 addButton.setMnemonic(ADD_BUTTON_MNEMONIC);
...
return panel;
}
...
}
```

After seeing your tests pass, execute sis.ui.Sis as an application. Enter a course department and number, then press Alt-A to demonstrate use of the mnemonic.

Required Fields

A valid course requires both a department and course number. However, sis.ui.Sis allows you to omit either or both, yet still press the Add button. You want to modify the application to disallow this circumstance.

One solution would be to wait until the user presses Add, then ensure that both department and course number contain a nonempty string. If not, then present a message pop-up that explains the requirement. While this solution will work, it creates a more annoying user experience. Users don't want continual interruption from message pop-ups. A better solution is to proactively disable the Add button until the user has entered information in both fields.

You can monitor both fields and track when a user enters information in them. Each time a user presses a character, you can test the field contents and enable or disable the Add button as appropriate.

Managing enabling/disabling of the Add button is an application characteristic, not a view characteristic. It involves business-related logic, as it is based on the business need for specific data. As such, the test and related code belongs not in the panel class, but elsewhere.

Another indicator that the code does not belong in the view class is that you have interaction between two components. You want the controller to notify other classes of an event (keys being pressed) and you want those other classes to tell the view what to present (a disabled or enabled button) under certain circumstances. These are two separate concerns. You don't want logic in the view trying to mediate things.

To solve the bigger problem, however, it is easier to provide tests against CoursesPanel that prove each of the two smaller occurrences. A first test ensures that a listener is notified when keys are pressed. A second test ensures that CoursesPanel can enable and disable buttons.

Start with a test (in CoursesPanelTest) for enabling and disabling buttons:

Feel

```
public void testEnableDisable() {
    panel.setEnabled(ADD_BUTTON_NAME, true);
    JButton button = panel.getButton(ADD_BUTTON_NAME);
    assertTrue(button.isEnabled());

    panel.setEnabled(ADD_BUTTON_NAME, false);
    assertFalse(button.isEnabled());
}
```

The code in CoursesPanel is a one-line reactive method—no logic:

```
void setEnabled(String name, boolean state) {
    getButton(name).setEnabled(state);
}
```

The second test shows how to attach a keystroke listener to a field:

```
public void testAddListener() throws Exception {
    KeyListener listener = new KeyAdapter() {};
    panel.addFieldListener(DEPARTMENT_FIELD_NAME, listener);
    JTextField field = panel.getField(DEPARTMENT_FIELD_NAME);
    KeyListener[] listeners = field.getKeyListeners();
    assertEquals(1, listeners.length);
    assertSame(listener, listeners[0]);
}
```

A KeyAdapter is an abstract implementation of the KeyListener interface that does nothing. The first line in the test creates a concrete subclass of KeyAdapter that overrides nothing. After adding the listener (using addFieldListener) to the panel, the test ensures that the panel properly sets the listener into the text field. The code in CoursesPanel is again trivial:

```
void addFieldListener(String name, KeyListener listener) {
    getField(name).addKeyListener(listener);
}
```

The harder test is at the application level. A listener in the Sis object should receive messages when a user types into the department and number fields. You must prove that the listener's receipt of these messages triggers logic to enable/disable the Add button. You must also prove that various combinations of text/no text in the fields results in the appropriate state for the Add button.

A bit of programming by intention in SisTest will provide you with a test skeleton:

```
public void testKeyListeners() throws Exception {
    sis.show();

    JButton button = panel.getButton(CoursesPanel.ADD_BUTTON_NAME);
    assertFalse(button.isEnabled());
    selectField(CoursesPanel.DEPARTMENT_FIELD_NAME);
    type('A');
```

Feel

```

selectField(CoursesPanel.NUMBER_FIELD_NAME);
type('1');
assertTrue(button.isEnabled());
}

```

The test ensures that the button is disabled by default. After typing values into the department and number fields, it verifies that the button is enabled. The trick, of course, is how to select a field and emulate typing into it. Swing provides a few solutions. Unfortunately, each requires you to render (make visible) the actual screen. Thus the first line in the test is a call to the `show` method of Sis.

The solution I'll present involves use of the class `java.awt.Robot`. The `Robot` class emulates end-user interaction using the keyboard and/or mouse. Another solution requires you to create keyboard event objects and pass them to the fields using a method on `java.awt.Component` named `dispatchEvent`.

You can construct a `Robot` object in the `SisTest` `setUp` method. (After building this example, I noted persistent use of the `CoursesPanel` object, so I also refactored its extraction to `setUp`.)

```

public class SisTest extends TestCase {
    ...
    private CoursesPanel panel;
    private Robot robot;

    protected void setUp() throws Exception {
        ...
        panel = (CoursesPanel)Util.getComponent(frame, CoursesPanel.NAME);
        robot = new Robot();
    }
}

```

The `selectField` method isn't that tough:

```

private void selectField(String name) throws Exception {
    JTextField field = panel.getField(name);
    Point point = field.getLocationOnScreen();
    robot.mouseMove(point.x, point.y);
    robot.mousePress(InputEvent.BUTTON1_MASK);
    robot.mouseRelease(InputEvent.BUTTON1_MASK);
}

```

Feel

After obtaining a field object, you can obtain its absolute position on the screen by sending it the message `getLocationOnScreen`. This returns a `Point` object—a coordinate in Cartesian space represented by an *x* and *y* offset.³ You can send this coordinate as an argument to `Robot`'s `mouseMove` method.

³The upper left corner of your screen has an (*x*, *y*) coordinate of (0, 0). The value of *y* increases as you move down the screen. For example, you would express two pixels to the right and one pixel down as (2, 1).

Subsequently, sending a `mousePress` and `mouseRelease` message to the Robot results in a virtual mouse-click at that location.

The type method is equally straightforward:

```
private void type(int key) throws Exception {
    robot.keyPress(key);
    robot.keyRelease(key);
}
```

The code in Sis adds a single listener to each text field. This listener waits on `keyReleased` events. When it receives one, the listener calls the method `setAddButtonState`. The code in `setAddButtonState` looks at the contents of the two fields to determine whether or not to enable the Add button.

```
public class Sis {
    ...
    private void initialize() {
        createCoursesPanel();
        createKeyListeners();
        ...
    }
    ...
    void createKeyListeners() {
        KeyListener listener = new KeyAdapter() {
            public void keyReleased(KeyEvent e) {
                setAddButtonState();
            }
        };
        panel.addFieldListener(CoursesPanel.DEPARTMENT_FIELD_NAME,
            listener);
        panel.addFieldListener(CoursesPanel.NUMBER_FIELD_NAME, listener);

        setAddButtonState();
    }
    void setAddButtonState() {
        panel.setEnabled(CoursesPanel.ADD_BUTTON_NAME,
            !isEmpty(CoursesPanel.DEPARTMENT_FIELD_NAME) &&
            !isEmpty(CoursesPanel.NUMBER_FIELD_NAME));
    }
    private boolean isEmpty(String field) {
        String value = panel.getText(field);
        return value.equals("");
    }
}
```

Feel

Note that the last line in `createKeyListeners` calls `setAddButtonState` in order to set the Add button to its default (initial) state.

The code in `testKeyListeners` doesn't represent all possible scenarios. What if the user enters nothing but space characters? Is the button properly disabled if one of the two fields has data but the other does not?

You could enhance `testKeyListeners` with these scenarios. A second test shows a different approach, one that directly interacts with `setAddButtonState`. This test covers a more complete set of circumstances.

```
public void testSetAddButtonState() throws Exception {
    JButton button = panel.getButton(CoursesPanel.ADD_BUTTON_NAME);
    assertFalse(button.isEnabled());

    panel.setText(CoursesPanel.DEPARTMENT_FIELD_NAME, "a");
    sis.setAddButtonState();
    assertFalse(button.isEnabled());

    panel.setText(CoursesPanel.NUMBER_FIELD_NAME, "1");
    sis.setAddButtonState();
    assertTrue(button.isEnabled());

    panel.setText(CoursesPanel.DEPARTMENT_FIELD_NAME, " ");
    sis.setAddButtonState();
    assertFalse(button.isEnabled());

    panel.setText(CoursesPanel.DEPARTMENT_FIELD_NAME, "a");
    panel.setText(CoursesPanel.NUMBER_FIELD_NAME, " ");
    sis.setAddButtonState();
    assertFalse(button.isEnabled());
}
```

The test fails. A small change to `isEmpty` fixes things.

```
private boolean isEmpty(String field) {
    String value = panel.getText(field);
    return value.trim().equals("");
}
```

Field Edits

When you provide an effective user interface, you want to make it as difficult as possible for users to enter invalid data. You learned that you don't want to interrupt users with pop-ups when requiring fields. Similarly, you want to avoid presenting pop-ups to tell users they have entered invalid data.

A preferred solution involves verifying and even modifying data in a text field as the user enters it. As an example, a course department must contain only uppercase letters. “CMSC” is a valid department, but “Cmsc” and “cmSC” are not. To make life easier for your users, you can make the department text field automatically convert each lowercase letter to an uppercase letter as the user types it.

Feel

The evolution of Java has included several attempts at solutions for dynamically editing fields. Currently, there are at least a half dozen ways to go about it. You will learn two of the preferred techniques: using `JFormattedTextField` and creating custom `DocumentFilter` classes.

You create a custom filter by subclassing `javax.swing.text.DocumentFilter`. In the subclass, you override definitions for any of three methods: `insertString`, `remove`, and `replace`. You use these methods to restrict invalid input and/or transform invalid input into valid input.

As a user enters or pastes characters into the text field, the `insertString` method is indirectly called. The `replace` method gets invoked when a user first selects existing characters in a text field before typing or pasting new characters. The `remove` method is invoked when the user deletes characters from the text field. You will almost always need to define behavior for `insertString` and `replace`, but you will need to do so only occasionally for `remove`.

Once you have defined the behavior for the custom filter, you attach it to a text field's document. The document is the underlying data model for the text field; it is an implementation of the interface `javax.swing.text.Document`. You obtain the `Document` object associated with a `JTextField` by sending it the message `getDocument`. You can then attach the custom filter to the `Document` using `setDocumentFilter`.

Testing the Filter

How will you test the filter? You could code a test in `CoursesPanel` that uses the Swing robot (as described in the Required Fields section). But for the purpose of testing units, the robot is a last-resort technique that you should use only when you must. In this case, a `DocumentFilter` subclass is a stand-alone class that you can test directly.

Feel

In some cases, you'll find that Swing design lends itself to easy testing. For custom filters, you'll have to do a bit of legwork first to get around a few barriers.

`UpcaseFilterTest` appears directly below. The individual test method `testInsert` is straightforward and easy to read, the result of some refactoring. In `testInsert`, you send the message `insertString` directly to an `UpcaseFilter` instance. The second argument to `insertString` is the column at which to begin inserting. The third argument is the text to insert. (For now, the fourth argument is irrelevant, and I'll discuss the first argument shortly).

Inserting the text "abc" at column 0 should generate the text "ABC". Inserting "def" at position 1 (i.e., before the second column) should generate the text "ADEFBC".

```

package sis.ui;

import javax.swing.*;
import javax.swing.text.*;
import junit.framework.*;

public class UpcaseFilterTest extends TestCase {
    private DocumentFilter filter;
    protected DocumentFilter.FilterBypass bypass;
    protected AbstractDocument document;

    protected void setUp() {
        bypass = createBypass();
        document = (AbstractDocument)bypass.getDocument();
        filter = new UpcaseFilter();
    }

    public void testInsert() throws BadLocationException {
        filter.insertString(bypass, 0, "abc", null);
        assertEquals("ABC", documentText());

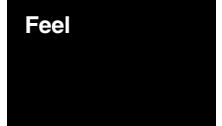
        filter.insertString(bypass, 1, "def", null);
        assertEquals("ADEFBC", documentText());
    }

    protected String documentText() throws BadLocationException {
        return document.getText(0, document.getLength());
    }

    protected DocumentFilter.FilterBypass createBypass() {
        return new DocumentFilter.FilterBypass() {
            private AbstractDocument document = new PlainDocument();
            public Document getDocument() {
                return document;
            }
            public void insertString(
                int offset, String string, AttributeSet attr) {
                try {
                    document.insertString(offset, string, attr);
                }
                catch (BadLocationException e) {}
            }
            public void remove(int offset, int length) { }
            public void replace(int offset,
                int length, String string, AttributeSet attrs) { }
        };
    }
}

```

Feel



The setup is considerably more involved than the test itself.

If you look at the javadoc for `insertString`, you'll see that it takes a reference of type `DocumentFilter.FilterBypass` as its first argument. A filter bypass is es-

sentially a reference to the document that ignores any filters. After you transform data in `insertString`, you must call `insertString` on the filter bypass. Otherwise, you will create an infinite loop!

The difficulty with respect to testing is that Swing provides no direct way to obtain a filter bypass object. You need the bypass in order to test the filter.

The solution presented above is to provide a new implementation of `DocumentFilter.FilterBypass`. This implementation stores a concrete instance of an `AbstractDocument` (which implements the `Document` interface) known as a `PlainDocument`. To flesh out the bypass, you must supply implementations for the three methods `insertString`, `remove`, and `replace`. For now, the test only requires you to implement `insertString`.

The `insertString` method doesn't need to take a bypass object as its first parameter, since it is defined in the filter itself. Its job is to call the document's `insertString` method directly (i.e., without calling back into the `DocumentFilter`). Note that this method can throw a `BadLocationException` if the start position is out of range.

Once you have a `DocumentFilter.FilterBypass` instance, the remainder of the setup and test is easy. From the bypass object, you can obtain and store a document reference. You assert that the contents of this document were appropriately updated.

The test (`UpcaseFilterTest`) contains a lot of code. You might think that the robot-based test would have been easier to write. In fact, it would have. However, robots have their problems. Since they take control of the mouse and keyboard, you have to be careful not to do anything else while the tests execute. Otherwise you can cause the robot tests to fail. This alone is reason to avoid them at all costs. If you must use robot tests, find a way to isolate them and perhaps execute them at the beginning of your unit-test suite.

Also, the test code for the second filter you write will be as easy to code as the corresponding robot test code. Both filter tests would require the `documentText` and `createBypass` methods as well as most of the `setUp` method.

Feel

Coding the Filter

You're more than halfway done with building a filter. You've completed the hard part—writing a test for it. Coding the filter itself is trivial.

```
package sis.ui;

import javax.swing.text.*;

public class UpcaseFilter extends DocumentFilter {
    public void insertString(
        DocumentFilter.FilterBypass bypass,
```

```

        int offset,
        String text,
        AttributeSet attr) throws BadLocationException {
    bypass.insertString(offset, text.toUpperCase(), attr);
}
}

```

When the filter receives the `insertString` message, its job in this case is to convert the text argument to uppercase, and pass this transformed data off to the bypass.

Once you've demonstrated that your tests all still pass, you can now code the `replace` method. The test modifications:

```

...
public class UpcaseFilterTest extends TestCase {
    ...
    public void testReplace() throws BadLocationException {
        filter.insertString(bypass, 0, "XYZ", null);
        filter.replace(bypass, 1, 2, "tc", null);
        assertEquals("XTC", documentText());

        filter.replace(bypass, 0, 3, "p8A", null);
        assertEquals("P8A", documentText());
    }
    ...
    protected DocumentFilter.FilterBypass createBypass() {
        return new DocumentFilter.FilterBypass() {
            ...
            public void replace(int offset,
                int length, String string, AttributeSet attrs) {
                try {
                    document.replace(offset, length, string, attrs);
                }
                catch (BadLocationException e) {}
            }
        };
    }
}

```

The test shows that the `replace` method takes an additional argument. The third parameter represents the number of characters to be replaced, starting at the position represented by the second argument. The production code:

```

package sis.ui;

import javax.swing.text.*;

public class UpcaseFilter extends DocumentFilter {
    ...
    public void replace(
        DocumentFilter.FilterBypass bypass,
        int offset,

```

Feel

```
    int length,  
    String text,  
    AttributeSet attr) throws BadLocationException {  
    bypass.replace(offset, length, text.toUpperCase(), attr);  
}  
}
```

UpcaseFilter is complete. You need not concern yourself with filtering removal operations when uppercasing input.

Attaching the Filter

You have proved the functionality of UpcaseFilter as a standalone unit. To prove that the department field in CoursesPanel transforms its input into uppercase text, you need only demonstrate that the appropriate filter has been attached to the field.

Should the code in CoursesPanel attach the filters to its fields, or should code in Sis retrieve the fields and attach the filters? Does the test belong in SisTest or in CoursesPanelTest? A filter is a combination of business rule and view functionality. It enforces a business constraint (for example, “Department abbreviations are four uppercase characters”). A filter also enhances the feel of the application by making it easier for the user to enter only valid information.

Remember: Keep the view class simple. Put as much business-related logic in domain (Course) or application classes (Sis). The filter representation of the business logic is very dependent upon Swing. The filters are essentially plug-ins to the Swing framework. You don't want to make the domain class dependent upon such code. Thus, the only remaining choice is the application class.

The code in SisTest:

Feel

```
public void testCreate() {
    ...
    CoursesPanel panel =
        (CoursesPanel)Util.getComponent(frame, CoursesPanel.NAME);
    assertNotNull(panel);

    ...
    verifyFilter(panel);
}

private void verifyFilter(CoursesPanel panel) {
    DocumentFilter filter =
        getFilter(panel, CoursesPanel.DEPARTMENT_FIELD_NAME);
    assertTrue(filter.getClass() == UpcaseFilter.class);
}
```

```

private DocumentFilter getFilter(
    CoursesPanel panel, String fieldName) {
    JTextField field = panel.getField(fieldName);
    AbstractDocument document = (AbstractDocument)field.getDocument();
    return document.getDocumentFilter();
}
...
}

```

The code in Sis:

```

private void initialize() {
    createCoursesPanel();
    createKeyListeners();
    createInputFilters();
    ...
}

private void createInputFilters() {
    JTextField field =
        panel.getField(CoursesPanel.DEPARTMENT_FIELD_NAME);
    AbstractDocument document = (AbstractDocument)field.getDocument();
    document.setDocumentFilter(new UpcaseFilter());
}

```

A Second Filter

You also want to constrain the number of characters in both the department and course number field. In fact, in most applications that require field entry, you will want the ability to set field limits. You can create a second custom filter, LimitFilter. The following code listing shows only the production class. The test, LimitFilterTest (see the code at <http://www.LangrSoft.com/agileJava/code/>) contains a lot of commonality with UpcaseFilterTest that you can factor out.

```

package sis.ui;

import javax.swing.text.*;

public class LimitFilter extends DocumentFilter {
    private int limit;

    public LimitFilter(int limit) {
        this.limit = limit;
    }

    public void insertString(
        DocumentFilter.FilterBypass bypass,
        int offset,
        String str,
        AttributeSet attrSet) throws BadLocationException {
        replace(bypass, offset, 0, str, attrSet);
    }
}

```

Feel

```

public void replace(
    DocumentFilter.FilterBypass bypass,
    int offset,
    int length,
    String str,
    AttributeSet attrSet) throws BadLocationException {
    int newLength =
        bypass.getDocument().getLength() - length + str.length();
    if (newLength > limit)
        throw new BadLocationException(
            "New characters exceeds max size of document", offset);
    bypass.replace(offset, length, str, attrSet);
}
}

```

Note the technique of having `insertString` delegate to the `replace` method. The other significant bit of code involves throwing a `BadLocationException` if the replacement string is too large.

Building such a filter and attaching it to the course number field is easy enough. You construct a `LimitFilter` by passing it the character length. For example, the code snippet `new LimitFilter(3)` creates a filter that prevents more than three characters.

The problem is that you can set only a single filter on a document. You have a couple of choices. The first (bad) choice is to create a separate filter for each combination. For example, you might have filter combinations `UppercaseLimitFilter` and `NumericOnlyLimitFilter`. A better solution involves some form of abstraction—a `ChainableFilter`. The `ChainableFilter` class subclasses `DocumentFilter`. It contains a sequence of individual filter classes and manages calling each in turn. The code available at <http://www.LangrSoft.com/agileJava/code/> for this lesson demonstrates how you might build such a construct.⁴

Feel

JFormattedTextField

Another mechanism for managing field edits is to use the class `javax.swing.JFormattedTextField`, a subclass of `JTextField`. You can attach formatters to the field to ensure that the contents conform to your specification. Further, you can retrieve the contents of the field as appropriate object types other than text.

You want to provide an effective date field for the course. This date represents when the course is first made available in the system. Users must enter the date in the format mm/dd/yy. For example, 04/15/02 is a valid date.

⁴The listing does not appear here for space reasons.

The test extracts the field as a JFormattedTextField, then gets a formatter object from the JFormattedTextField. A formatter is a subclass of javax.swing.JFormattedTextField.AbstractFormatter. In verifyEffectiveDate, you expect that the formatter is a DateFormatter. The DateFormatter in turn wraps a java.text.SimpleDateFormat instance whose format pattern is MM/dd/yy.⁵

The final part of the test ensures that the field holds on to a date instance. When the user clicks Add, code in sis.ui.Sis can extract the contents of the effective date field as a java.util.Date object.

```
private void verifyEffectiveDate() {
    assertEquals(EFFECTIVE_DATE_LABEL_NAME,
                EFFECTIVE_DATE_LABEL_TEXT);

    JFormattedTextField dateField =
        (JFormattedTextField)panel.getField(EFFECTIVE_DATE_FIELD_NAME);
    DateFormatter formatter = (DateFormatter)dateField.getFormatter();
    SimpleDateFormat format = (SimpleDateFormat)formatter.getFormat();
    assertEquals("MM/dd/yy", format.toPattern());
    assertEquals(Date.class, dateField.getValue().getClass());
}
```

Code in CoursesPanel constructs the JFormattedTextField, passing a SimpleDateFormat to its constructor. The code sends the message setValue to dateField in order to supply the Date object in which to store the edited results.

```
JPanel createFieldsPanel() {
    GridBagLayout layout = new GridBagLayout();

    JPanel panel = new JPanel(layout);
    int columns = 20;

    addField(panel, layout, 0,
             DEPARTMENT_LABEL_NAME, DEPARTMENT_LABEL_TEXT,
             createField(DEPARTMENT_FIELD_NAME, columns));

    addField(panel, layout, 1,
             NUMBER_LABEL_NAME, NUMBER_LABEL_TEXT,
             createField(NUMBER_FIELD_NAME, columns));

    Format format = new SimpleDateFormat("MM/dd/yy");
    JFormattedTextField dateField = new JFormattedTextField(format);
    dateField.setValue(new Date());
    dateField.setColumns(columns);
    dateField.setName(EFFECTIVE_DATE_FIELD_NAME);
```

Feel

⁵The capital letter M is used for month, while the lowercase letter m is used for minutes.

```

addField(panel, layout, 2,
    EFFECTIVE_DATE_LABEL_NAME, EFFECTIVE_DATE_LABEL_TEXT,
    dateField);

return panel;
}

```

If you execute the application with these changes, you will note that the effective date field allows you to type invalid input. When you leave the field, it reverts to a valid value. You can override this default behavior; see the API documentation for `JFormattedTextField` for the alternatives.

A design issue now exists. The code to create the formatted text field is in `CoursesPanel` and the associated test is in `CoursesPanelTest`. This contrasts with the goal I previously stated to manage edits at the application level!

You want to completely separate the view and application concerns. A solution involves the single responsibility principle. It will also eliminate some of the duplication and code clutter that I've allowed to fester in `CoursesPanel` and `Sis`.

A `Field` object is a data object whose attributes describe the information necessary to be able to create Swing text fields. A `Field` is implementation-neutral, however, and has no knowledge of Swing. A `FieldCatalog` contains the collection of available fields. It can return a `Field` object given its name.

The `CoursesPanel` class needs only contain a list of field names that it must render. The `CoursesPanel` code can iterate through this list, asking a `FieldCatalog` for the corresponding `Field` object. It can then send the `Field` object to a factory, `TextFieldFactory`, whose job is to return a `JTextField`. The factory will take information from the `Field` object and use it to add various constraints on the `JTextField`, such as formats, filters, and length limits.

Feel

The code for the new classes follows. I also show the code in `CoursesPanel` that constructs the text fields.

```

// FieldCatalogTest.java
package sis.ui;

import junit.framework.*;
import static sis.ui.FieldCatalog.*;

public class FieldCatalogTest extends TestCase {
    public void testAllFields() {
        FieldCatalog catalog = new FieldCatalog();

        assertEquals(3, catalog.size());
    }
}

```

```

Field field = catalog.get(NUMBER_FIELD_NAME);
assertEquals(DEFAULT_COLUMNS, field.getColumns());
assertEquals(NUMBER_LABEL_TEXT, field.getLabel());
assertEquals(NUMBER_FIELD_LIMIT, field.getLimit());

field = catalog.get(DEPARTMENT_FIELD_NAME);
assertEquals(DEFAULT_COLUMNS, field.getColumns());
assertEquals(DEPARTMENT_LABEL_TEXT, field.getLabel());
assertEquals(DEPARTMENT_FIELD_LIMIT, field.getLimit());
assertTrue(field.isUppercaseOnly());

field = catalog.get(EFFECTIVE_DATE_FIELD_NAME);
assertEquals(DEFAULT_COLUMNS, field.getColumns());
assertEquals(EFFECTIVE_DATE_LABEL_TEXT, field.getLabel());
assertSame(DEFAULT_DATE_FORMAT, field.getFormat());
}
}

// FieldCatalog.java
package sis.ui;

import java.util.*;
import java.text.*;

public class FieldCatalog {

    public static final DateFormat DEFAULT_DATE_FORMAT =
        new SimpleDateFormat("MM/dd/yy");

    static final String DEPARTMENT_FIELD_NAME = "deptField";
    static final String DEPARTMENT_LABEL_TEXT = "Department";
    static final int DEPARTMENT_FIELD_LIMIT = 4;

    static final String NUMBER_FIELD_NAME = "numberField";
    static final String NUMBER_LABEL_TEXT = "Number";
    static final int NUMBER_FIELD_LIMIT = 3;

    static final String EFFECTIVE_DATE_FIELD_NAME = "effectiveDateField";
    static final String EFFECTIVE_DATE_LABEL_TEXT = "Effective Date";

    static final int DEFAULT_COLUMNS = 20;

    private Map<String,Field> fields;

    public FieldCatalog() {
        loadFields();
    }

    public int size() {
        return fields.size();
    }
}

```

Feel

```

private void loadFields() {
    fields = new HashMap<String,Field>();

    Field fieldSpec = new Field(DEPARTMENT_FIELD_NAME);
    fieldSpec.setLabel(DEPARTMENT_LABEL_TEXT);
    fieldSpec.setLimit(DEPARTMENT_FIELD_LIMIT);
    fieldSpec.setColumns(DEFAULT_COLUMNS);
    fieldSpec.setUpcaseOnly();

    put(fieldSpec);

    fieldSpec = new Field(NUMBER_FIELD_NAME);
    fieldSpec.setLabel(NUMBER_LABEL_TEXT);
    fieldSpec.setLimit(NUMBER_FIELD_LIMIT);
    fieldSpec.setColumns(DEFAULT_COLUMNS);

    put(fieldSpec);

    fieldSpec = new Field(EFFECTIVE_DATE_FIELD_NAME);
    fieldSpec.setLabel(EFFECTIVE_DATE_LABEL_TEXT);
    fieldSpec.setFormat(DEFAULT_DATE_FORMAT);
    fieldSpec.setInitialValue(new Date());
    fieldSpec.setColumns(DEFAULT_COLUMNS);

    put(fieldSpec);
}

private void put(Field fieldSpec) {
    fields.put(fieldSpec.getName(), fieldSpec);
}

public Field get(String fieldName) {
    return fields.get(fieldName);
}
}

```

// TextFieldFactoryTest.java
package sis.ui;

```

import javax.swing.*;
import javax.swing.text.*;
import java.util.*;
import java.text.*;
import junit.framework.*;
import sis.util.*;

public class TextFieldFactoryTest extends TestCase {
    private Field fieldSpec;
    private static final String FIELD_NAME = "fieldName";
    private static final int COLUMNS = 1;

    protected void setUp() {
        fieldSpec = new Field(FIELD_NAME);

```

Feel

```

        fieldSpec.setColumns(COLUMNS);
    }

    public void testCreateSimpleField() {
        final String textView = "value";
        fieldSpec.setInitialValue(textView);

        JTextField field = TextFieldFactory.create(fieldSpec);

        assertEquals(COLUMNS, field.getColumns());
        assertEquals(FIELD_NAME, field.getName());
        assertEquals(textView, field.getText());
    }

    public void testLimit() {
        final int limit = 3;
        fieldSpec.setLimit(limit);

        JTextField field = TextFieldFactory.create(fieldSpec);

        AbstractDocument document = (AbstractDocument)field.getDocument();
        ChainableFilter filter =
            (ChainableFilter)document.getDocumentFilter();
        assertEquals(limit, ((LimitFilter)filter).getLimit());
    }

    public void testUpcase() {
        fieldSpec.setUpcaseOnly();

        JTextField field = TextFieldFactory.create(fieldSpec);

        AbstractDocument document = (AbstractDocument)field.getDocument();
        ChainableFilter filter =
            (ChainableFilter)document.getDocumentFilter();
        assertEquals(UpcaseFilter.class, filter.getClass());
    }

    public void testMultipleFilters() {
        fieldSpec.setLimit(3);
        fieldSpecsetUpcaseOnly();

        JTextField field = TextFieldFactory.create(fieldSpec);

        AbstractDocument document = (AbstractDocument)field.getDocument();
        ChainableFilter filter =
            (ChainableFilter)document.getDocumentFilter();

        Set<Class> filters = new HashSet<Class>();
        filters.add(filter.getClass());
        filters.add(filter.getNext().getClass());

        assertTrue(filters.contains(LimitFilter.class));
        assertTrue(filters.contains(UpcaseFilter.class));
    }
}

```

Feel

```

public void testCreateFormattedField() {
    final int year = 2006;
    final int month = 3;
    final int day = 17;
    fieldSpec.setInitialValue(DateUtil.createDate(year, month, day));
    final String pattern = "MM/dd/yy";
    fieldSpec.setFormat(new SimpleDateFormat(pattern));

    JFormattedTextField field =
        (JFormattedTextField)TextFieldFactory.create(fieldSpec);

    assertEquals(1, field.getColumns());
    assertEquals(FIELD_NAME, field.getName());

    DateFormatter formatter = (DateFormatter)field.getFormatter();
    SimpleDateFormat format = (SimpleDateFormat)formatter.getFormat();
    assertEquals(pattern, format.toPattern());
    assertEquals(Date.class, field.getValue().getClass());
    assertEquals("03/17/06", field.getText());

    TestUtil.assertDateEquals(year, month, day,
        (Date)field.getValue()); // a new utility method
    }
}

// TextFieldFactory.java
package sis.ui;

import javax.swing.*;
import javax.swing.text.*;

public class TextFieldFactory {
    public static JTextField create(Field fieldSpec) {
        JTextField field = null;

        if (fieldSpec.getFormat() != null)
            field = createFormattedTextField(fieldSpec);
        else {
            field = new JTextField();
            if (fieldSpec.getInitialValue() != null)
                field.setText(fieldSpec.getInitialValue().toString());
        }

        if (fieldSpec.getLimit() > 0)
            attachLimitFilter(field, fieldSpec.getLimit());

        if (fieldSpec.isUppercaseOnly())
            attachUppercaseFilter(field);

        field.setColumns(fieldSpec.getColumns());
        field.setName(fieldSpec.getName());
        return field;
    }
}

```

Feel

```

private static void attachLimitFilter(JTextField field, int limit) {
    attachFilter(field, new LimitFilter(limit));
}

private static void attachUppercaseFilter(JTextField field) {
    attachFilter(field, new UppercaseFilter());
}

private static void attachFilter(
    JTextField field, ChainableFilter filter) {
    AbstractDocument document = (AbstractDocument)field.getDocument();
    ChainableFilter existingFilter =
        (ChainableFilter)document.getDocumentFilter();
    if (existingFilter == null)
        document.setDocumentFilter(filter);
    else
        existingFilter.setNext(filter);
}

private static JTextField createFormattedTextField(Field fieldSpec) {
    JFormattedTextField field =
        new JFormattedTextField(fieldSpec.getFormat());
    field.setValue(fieldSpec.getInitialValue());
    return field;
}
}

// CoursesPanelTest.java
...
public void testCreate() {
    assertEmptyList(COURSES_LIST_NAME);
    assertButtonText(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);

    String[] fields =
        { FieldCatalog.DEPARTMENT_FIELD_NAME,
          FieldCatalog.NUMBER_FIELD_NAME,
          FieldCatalog.EFFECTIVE_DATE_FIELD_NAME };
    assertFields(fields);

    JButton button = panel.getButton(ADD_BUTTON_NAME);
    assertEquals(ADD_BUTTON_MNEMONIC, button.getMnemonic());
}

private void assertFields(String[] fieldNames) {
    FieldCatalog catalog = new FieldCatalog();
    for (String fieldName: fieldNames) {
        assertNotNull(panel.getField(fieldName));
        // can't compare two JTextField items for equality,
        // so we must go on faith here that CoursesPanel
        // creates them using TextFieldFactory
        Field fieldSpec = catalog.get(fieldName);
    }
}

```

Feel

```

        assertLabelText(fieldSpec.getLabelName(), fieldSpec.getLabel());
    }
}

...

// CoursesPanel.java
...

JPanel createFieldsPanel() {
    GridBagLayout layout = new GridBagLayout();

    JPanel panel = new JPanel(layout);
    int i = 0;
    FieldCatalog catalog = new FieldCatalog();

    for (String fieldName: getFieldNames()) {
        Field fieldSpec = catalog.get(fieldName);
        addField(panel, layout, i++,
            createLabel(fieldSpec),
            TextFieldFactory.create(fieldSpec));
    }

    return panel;
}

private String[] getFieldNames() {
    return new String[]
        { FieldCatalog.DEPARTMENT_FIELD_NAME,
          FieldCatalog.NUMBER_FIELD_NAME,
          FieldCatalog.EFFECTIVE_DATE_FIELD_NAME };
}

private void addField(
    JPanel panel, GridBagLayout layout, int row,
    JLabel label, JTextField field) {
    ...
    panel.add(label);
    panel.add(field);
}
...

```

Feel

Notes:

- TestUtil.assertEquals is a new utility method whose implementation should be obvious.
- I finally moved the DateUtil class from the sis.studentinfo package to the sis.util package. This change impacts a number of existing classes.
- You must also edit Sis and CoursesPanel (and tests) to remove constants and code for constructing filters/formatters. See <http://www.LangrSoft.com/agileJava/code/> for full code.

- The Field class, which is omitted from these listings, is a simple data class with virtually no logic.
- You'll want to update the Course class to contain the new attribute, effective date.

Tables

The CoursesPanel JList shows a single string for each Course object. This presentation is adequate because you only display two pieces of data: the department and course number. Adding the new effective date attribute to the summary string, however, would quickly make the list look jumbled. The JList control, in fact, works best when you have only a single piece of data to represent per row in the list.

A JTable is a very effective control that allows you to present each object as a sequence of columns. A JTable can look a lot like a spreadsheet. In fact, JTable code and documentation uses the same terms as spreadsheets: rows, columns, and cells.

The JTable class gives you a considerable amount of control over look and feel. For example, you can decide whether or not you want to allow users to edit individual cells in the table, you can allow the user to rearrange the columns, and you can control the width of each column.

For this exercise, you'll replace the JList with a JTable. The best place to start is to create the data model that will underly the JTable. Just as you inserted Course objects into a list model for the JList, you will put Course objects into a model that you attach to the JTable.

Creating the JTable model is slightly more involved. The JList was able to get a printable representation by sending the `toString` message to each object it contained. The JTable must treat each attribute for a given object separately. For every cell it must display, the JTable sends the message `getValueAt` to the model. It passes the row and column representing the current cell. The `getValueAt` method must return a string to display at this location.

There would be no easy way for the model to figure out what attribute you want to display for a given column. As such, you must provide a model implementation yourself. You must supply the `getValueAt` method in this implementation, as well as two other methods: `getRowCount` and `getColumnCount`. To enhance the look and feel of the table, you will likely implement other methods. The test below, `CoursesTableModelTest`, shows that the model implements the method `getColumnName` to return a text header for each column.

Tables

```

package sis.ui;

import junit.framework.*;
import sis.studentinfo.*;
import sis.util.*;
import java.util.*;

public class CoursesTableModelTest extends TestCase {
    private CoursesTableModel model;

    protected void setUp() {
        model = new CoursesTableModel();
    }

    public void testCreate() {
        assertEquals(0, model.getRowCount());
        assertEquals(3, model.getColumnCount());
        FieldCatalog catalog = new FieldCatalog();

        Field department =
            catalog.get(FieldCatalog.DEPARTMENT_FIELD_NAME);
        assertEquals(department.getShortName(), model.getColumnName(0));

        Field number = catalog.get(FieldCatalog.NUMBER_FIELD_NAME);
        assertEquals(number.getShortName(), model.getColumnName(1));

        Field effectiveDate =
            catalog.get(FieldCatalog.EFFECTIVE_DATE_FIELD_NAME);
        assertEquals(effectiveDate.getShortName(),
                    model.getColumnName(2));
    }

    public void testAddRow() {
        Course course = new Course("CMSC", "110");

        course.setEffectiveDate(DateUtil.createDate(2006, 3, 17));

        model.add(course);

        assertEquals(1, model.getRowCount());
        final int row = 0;
        assertEquals("CMSC", model.getValueAt(row, 0));
        assertEquals("110", model.getValueAt(row, 1));
        assertEquals("03/17/06", model.getValueAt(row, 2));
    }
}

```

Tables

The test presents another use for the `FieldCatalog`—returning an appropriate column header for each field. It uses a new `Field` attribute with the more abstract concept of a “short name,” an abbreviated name for use in constrained display spaces. You’ll need to update `Field` and `FieldCatalog`/`FieldCatalogTest` to supply this new information.

The easiest way to build a table model is to subclass javax.swing.table.AbstractTableModel. You then need only supply definitions for `getValueAt`, `getRowCount`, and `getColumnCount`. You will want to store a collection of courses in the model. You'll need a method (`add`) that allows adding a new Course to the model.

You can also choose to work with javax.swing.table.DefaultTableModel. Sun provided this concrete implementation to make your job a bit simpler. However, DefaultTableModel requires that you organize your data first (in the form of either Vector objects or Object arrays) and then pass it to the model. It's almost as easy, and ultimately more effective, to create your own AbstractTableModel subclass.

```
package sis.ui;

import javax.swing.table.*;
import java.text.*;
import java.util.*;
import sis.studentinfo.*;

class CoursesTableModel extends AbstractTableModel {
    private List<Course> courses = new ArrayList<Course>();
    private SimpleDateFormat formatter =
        new SimpleDateFormat("MM/dd/yy");
    private FieldCatalog catalog = new FieldCatalog();
    private String[] fields = {
        FieldCatalog.DEPARTMENT_FIELD_NAME,
        FieldCatalog.NUMBER_FIELD_NAME,
        FieldCatalog.EFFECTIVE_DATE_FIELD_NAME };

    void add(Course course) {
        courses.add(course);
        fireTableRowsInserted(courses.size() - 1, courses.size());
    }

    Course get(int index) {
        return courses.get(index);
    }

    public String getColumnName(int column) {
        Field field = catalog.get(fields[column]);
        return field.getShortName();
    }

    // abstract (req'd) methods: getValueAt, getColumnCount, getRowCount
    public Object getValueAt(int row, int column) {
        Course course = courses.get(row);
        String fieldName = fields[column];
        if (fieldName.equals(FieldCatalog.DEPARTMENT_FIELD_NAME))
            return course.getDepartment();
        else if (fieldName.equals(FieldCatalog.NUMBER_FIELD_NAME))
            return course.getNumber();
    }
}
```

Tables

```

        else if (fieldName.equals(FieldCatalog.EFFECTIVE_DATE_FIELD_NAME))
            return formatter.format(course.getEffectiveDate());
        return "";
    }

    public int getColumnCount() {
        return fields.length;
    }

    public int getRowCount() {
        return courses.size();
    }
}

```

Note the subtle redundancies that abound. The table must contain a list of fields you are interested in displaying. In `getValueAt`, you obtain the field name at the column index provided. You use this name in a pseudo–switch statement to determine which Course getter method to call. A shorter bit of code would involve a switch statement:

```

switch (column) {
    case 0: return course.getDepartment();
    case 1: return course.getNumber();
    case 2: return formatter.format(course.getEffectiveDate());
    default: return "";
}

```

While it is acceptable, I dislike the disconnect between the column number and the attribute. Changes to the columns or their order can easily result in errors. (At least your tests would catch the problem.) But see the sidebar for a discussion of the solution I present.

You will, of course, need to make a few more changes to get the JTable working. In `CoursesPanelTest`:

```

public void testCreate() {
    assertEmptyTable(COURSES_TABLE_NAME);
    assertButtonText(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);
    ...
}

public void testAddCourse() {
    Course course = new Course("ENGL", "101");
    panel.addCourse(course);
    JTable table = panel.getTable(COURSES_TABLE_NAME);
    CoursesTableModel model = (CoursesTableModel)table.getModel();
    assertEquals(course, model.get(0));
}

private void assertEmptyTable(String name) {
    JTable table = panel.getTable(name);
    assertEquals(0, table.getModel().getRowCount());
}

```

Tables

Domain Maps

The implementation of `getValueAt` contains significant redundancy. You test the selected field name against each possible field name in order to obtain the corresponding attribute from the `Course` object. A more radical solution is to implement your domain objects as hash tables. The hash table key is the attribute name. You would set the value in the `Course` like this:

```
course.set(DEPARTMENT_FIELD_NAME, "CMSC");
```

And you would retrieve the value like this:

```
String courseName = (String)course.get(DEPARTMENT_FIELD_NAME);
```

or

```
String courseName = course.getString(DEPARTMENT_FIELD_NAME);
```

The code in `getValueAt` becomes very succinct:

```
Course course = courses.get(row);
return course.get(fields[column]);
```

This solution can greatly diminish the redundancy in your system, but it introduces other concerns. First is the decreased readability when you attempt to follow the code or debug it. Also, the solution would require casting, either at the point of call or embedded within a bunch of utility methods such as `getString`, `getDate`, and `getInt`.

In `CoursesPanel`:

```
static final String COURSES_TABLE_NAME = "coursesTable";
...
private void createLayout() {
    JTable coursesTable = createCoursesTable();
    JScrollPane coursesScroll = new JScrollPane(coursesTable);
    coursesScroll.setVerticalScrollBarPolicy(
        ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
    ...
}

private JTable createCoursesTable() {
    JTable table = new JTable(coursesTableModel);
    table.setName(COURSES_TABLE_NAME);
    table.setShowGrid(false);
    table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    return table;
}

void addCourse(Course course) {
    coursesTableModel.add(course);
}

Course getCourse(int index) {
    return coursesTableModel.get(index);
}
```

Tables

You can remove the class `CourseDisplayAdapter` and any references to the old courses list.

Make sure you take a look at the Java API documentation for `JTable` and the various table model classes. The `JTable` class contains quite a bit of customization capabilities.

Feedback

Part of creating a good user experience is ensuring that you provide plenty of feedback. Sun has already built a lot of feedback into Swing. For example, when you click the mouse button atop a `JButton`, the `JButton` repaints to appear as if it were physically depressed. This kind of information reassures the user about his or her actions.

The Sis application lacks a pertinent piece of feedback: When the user enters one of the filtered or formatted text fields, how does he or she know what they're expected to type? The user will eventually discover the constraints they are under. But he or she will waste some time as they undergo guesswork, trial, and error.

You can instead provide your users with helpful information ahead of time. Several options exist:

- Put helpful information in the label for the field. Generally, though, there is not enough screen real estate to do so.
- Provide a separate online help window that describes how the application works.
- As the user moves the mouse over fields, display a small pop-up rectangle with relevant information. This is known as hover help, or tool tips. All modern applications such as Internet Explorer provide tool tips as you hover your mouse over toolbar buttons.
- As the user moves the mouse over fields, display relevant information in a status bar at the bottom of the window.

Feedback

For this exercise, you will choose the last option and create a status bar.

Unfortunately, for mouse-based testing, you must usually render (display) the user interface in order to test it. The reason is that components do not have established sizes until they are rendered. You can again use the Swing robot to help you write your tests. Note that the `setUp` in this test uses a couple of Swing utility methods that I will display in subsequent listings.

```

package sis.ui;

import junit.framework.*;
import javax.swing.*;
import java.awt.*;
import sis.util.*;

public class StatusBarTest extends TestCase {
    private JTextField field1;
    private JTextField field2;
    private StatusBar statusBar;
    private JFrame frame;

    protected void setUp() {
        field1 = new JTextField(10);
        field2 = new JTextField(10);
        statusBar = new StatusBar();

        JPanel panel = SwingUtil.createPanel(field1, field2, statusBar);
        frame = SwingUtil.createFrame(panel);
    }

    protected void tearDown() {
        frame.dispose();
    }

    public void testMouseover() throws Exception {
        final String text1 = "text1";
        final String text2 = "text2";

        statusBar.setInfo(field1, text1);
        statusBar.setInfo(field2, text2);

        Robot robot = new Robot();

        Point field1Location = field1.getLocationOnScreen();

        robot.mouseMove(field1Location.x - 1, field1Location.y - 1);
        assertEquals("", statusBar.getText().trim());

        robot.mouseMove(field1Location.x + 1, field1Location.y + 1);
        assertEquals(text1, statusBar.getText());
        Point field2Location = field2.getLocationOnScreen();

        robot.mouseMove(field2Location.x + 1, field2Location.y + 1);
        assertEquals(text2, statusBar.getText());
    }
}

```

Feedback

Conceptually, providing status information is a common need for all of your application's windows. Instead of cluttering each window with additional code, you can encapsulate the status concept in a separate class.

StatusBar

A StatusBar is a JLabel with additional functionality. You can associate an information String with each text field by sending setInfo to a Status object.

The test extracts the location of the first field, then moves the mouse to an arbitrary position outside this field. The status bar should show nothing; the first assertion proves that. The test then moves the mouse over the field, then ensures that the status bar contains the expected text. The test finally asserts that the status bar text changes to TEXT2 when the mouse is over the second field.

The SwingUtil class extracts common code used to create a simple panel and a frame for testing:

```
package sis.util;

import javax.swing.*;

public class SwingUtil {
    public static JPanel createPanel(JComponent... components) {
        JPanel panel = new JPanel();
        for (JComponent component: components)
            panel.add(component);
        return panel;
    }

    public static JFrame createFrame(JPanel panel) {
        JFrame frame = new JFrame();
        frame.getContentPane().add(panel);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 300);
        frame.setVisible(true);
        return frame;
    }
}
```

Feedback

In StatusBar, the job of setInfo is to add a mouse listener to the text field. The listener reacts to mouse entry and mouse exit events. When a user moves the mouse atop the text, listener code displays the associated information. When a user moves the mouse away from the text field, listener code clears the status bar.

```
package sis.ui;

import java.awt.event.*;
import java.util.*;
import javax.swing.*;

public class StatusBar extends JLabel {
    private final static String EMPTY = " ";
    
```

```

public StatusBar() {
    super(EMPTY);
    setBorder(BorderFactory.createLoweredBevelBorder());
}

public void setInfo(final JTextField textField, final String text) {
    textField.addMouseListener(
        new MouseAdapter() {
            public void mouseEntered(MouseEvent event) {
                setText(text);
            }

            public void mouseExited(MouseEvent event) {
                setText(EMPTY);
            }
        });
}
}

```

The test for StatusBar passes. Now you must attach the status bar to CoursesPanel. How will you test this? Your test for CoursesPanel could use the robot again. But it would be easier to ensure that each text field has been attached to the status bar.

Update the test in CoursesPanelTest. The assertion declares a new intent: A StatusBar object should be able to return the informational text for a given text field. It also suggests that the informational text should come from the field spec object, obtained from the field catalog.

```

private void assertFields(String[] fieldNames) {
    StatusBar statusBar =
        (StatusBar)Util.getComponent(panel, StatusBar.NAME);

    FieldCatalog catalog = new FieldCatalog();
    for (String fieldName: fieldNames) {
        JTextField field = panel.getField(fieldName);
        Field fieldSpec = catalog.get(fieldName);

        assertEquals(fieldSpec.getInfo(), statusBar.getInfo(field));
        assertEquals(fieldSpec.getLabelName(), statusBar.getLabel(field));
    }
}

```

Feedback

This will not compile because you have not implemented getInfo. Note also that you will need to associate a component name with the status bar. Here are the changes to StatusBarTest and StatusBar:

```

// StatusBarTest
...
public void testInfo() {
    statusBar.setInfo(field1, "a");
}

```

```

        assertEquals("a", statusBar.getInfo(field1));
    }
    ...

// StatusBar
package sis.ui;

import java.awt.event.*;
import java.util.*;
import javax.swing.*;

public class StatusBar extends JLabel {
    public static final String NAME = "StatusBar";
    private final static String EMPTY = " ";
    private Map<JTextField, String> infos =
        new IdentityHashMap<JTextField, String>();

    public StatusBar() {
        super(EMPTY);
        setName(NAME);
        setBorder(BorderFactory.createLoweredBevelBorder());
    }

    public String getInfo(JTextField textField) {
        return infos.get(textField);
    }

    public void setInfo(final JTextField textField, String text) {
        infos.put(textField, text);
        textField.addMouseListener(
            new MouseAdapter() {
                public void mouseEntered(MouseEvent event) {
                    setText(getInfo(textField));
                }

                public void mouseExited(MouseEvent event) {
                    setText(EMPTY);
                }
            });
    }
}

```

Feedback

You must also add a field, getter, and setter to Field. You'll need to modify FieldCatalog to populate each field with a pertinent informational string:

```

...
static final String DEPARTMENT_FIELD_INFO =
    "Enter a 4-character department designation.";
static final String NUMBER_FIELD_INFO =
    "The department number should be 3 digits.";
static final String EFFECTIVE_DATE_FIELD_INFO =
    "Effective date should be in mm/dd/yy format.";
...

```

```

private void loadFields() {
    fields = new HashMap<String,Field>();

    Field fieldSpec = new Field(DEPARTMENT_FIELD_NAME);
    ...
    fieldSpec.setInfo(DEPARTMENT_FIELD_INFO);

    put(fieldSpec);

    fieldSpec = new Field(NUMBER_FIELD_NAME);
    ...
    fieldSpec.setInfo(NUMBER_FIELD_INFO);

    put(fieldSpec);

    fieldSpec = new Field(EFFECTIVE_DATE_FIELD_NAME);
    ...
    fieldSpec.setInfo(EFFECTIVE_DATE_FIELD_INFO);
    put(fieldSpec);
}

```

Finally, the code in CoursesPanel to make it all work for the student information system:

```

private Status status;

private void createLayout() {
    ...
    add(coursesScroll, BorderLayout.CENTER);
    add(createBottomPanel(), BorderLayout.SOUTH);
}

JPanel createBottomPanel() {
    JLabel statusBar = new JLabel(" ");
    statusBar.setBorder(BorderFactory.createLoweredBevelBorder());
    status = new Status(statusBar);

    JPanel panel = new JPanel();
    panel.setLayout(new BorderLayout());
    panel.add(statusBar, BorderLayout.SOUTH);
    panel.add(createInputPanel(), BorderLayout.CENTER);
    return panel;
}

JPanel createFieldsPanel() {
    GridBagLayout layout = new GridBagLayout();

    JPanel panel = new JPanel(layout);
    int i = 0;
    FieldCatalog catalog = new FieldCatalog();

```

Feedback

```
for (String fieldName: getFieldNames()) {  
    Field fieldSpec = catalog.get(fieldName);  
    JTextField textField = TextFieldFactory.create(fieldSpec);  
    status.addText(textField, fieldSpec.getLabel());  
    addField(panel, layout, i++,  
            createLabel(fieldSpec),  
            textField);  
}  
  
return panel;  
}
```

If you execute sis.ui.Sis as a stand-alone application, you should now see something like Figure 2. I rested the mouse pointer above the department field when I captured this screen image.

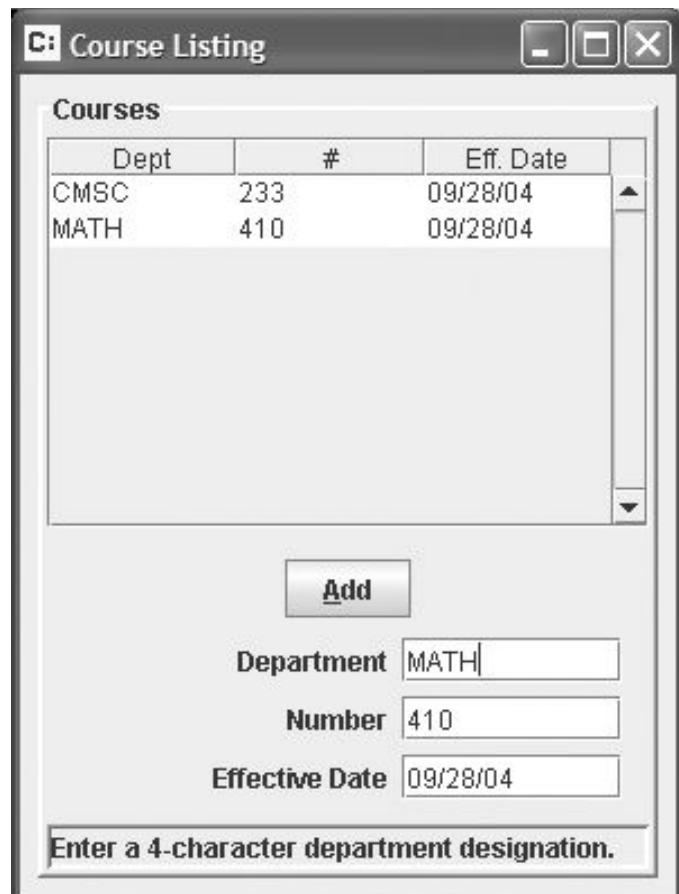


Figure 2 The Completed Look

Responsiveness

In the Sis method `addCourse`, insert a deliberate wait of three seconds. This wait might emulate the time required to verify and insert the `Course` object into the database.

```
private void addCourse() {
    Course course =
        new Course(
            panel.getText(FieldCatalog.DEPARTMENT_FIELD_NAME),
            panel.getText(FieldCatalog.NUMBER_FIELD_NAME));
    try { Thread.sleep(3000); } catch (InterruptedException e) {}
    JFormattedTextField effectiveDateField =
        (JFormattedTextField)panel.getField(
            FieldCatalog.EFFECTIVE_DATE_FIELD_NAME);
    Date date = (Date)effectiveDateField.getValue();
    course.setEffectiveDate(date);

    panel.addCourse(course);
}
```

Execute the application. Enter a department and course number and press Add. You should experience a 3-second delay. During that time, you cannot do anything else with the user interface! For an end user, this is a frustrating experience, since there is no feedback about why the application is not responding.

As an application developer, you can do two things with respect to responsiveness: First, provide feedback to the user that they should wait patiently for a short period. Second, ensure that the user is able to do other things while waiting.

Feedback comes in the form of a “wait” cursor. Windows represents a wait cursor using an hourglass. Some Unix desktops represent a wait cursor using a clock. You should provide an hourglass for any operation that does not immediately return control to the user.

```
private void addCourse() {
    frame.setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
    try {
        Course course =
            new Course(
                panel.getText(FieldCatalog.DEPARTMENT_FIELD_NAME),
                panel.getText(FieldCatalog.NUMBER_FIELD_NAME));
        try { Thread.sleep(3000); } catch (InterruptedException e) {}
        JFormattedTextField effectiveDateField =
            (JFormattedTextField)panel.getField(
                FieldCatalog.EFFECTIVE_DATE_FIELD_NAME);
```

Responsiveness

```

        Date date = (Date)effectiveDateField.getValue();
        course.setEffectiveDate(date);

        panel.addCourse(course);
    }
    finally {
        frame.setCursor(Cursor.getDefaultCursor());
    }
}

```

The `finally` block is essential! Otherwise, any unnoticed or abnormal return from `addCourse` will burden the user with an hourglass as a pointer.

Providing a wait cursor is an adequate and necessary response for any prolonged wait period. But the real solution is to ensure that the user does not have to wait. The threshold is a half a second: You should spawn a slow operation off in another thread if it is going to take any longer. In the example shown here, I've separated the behind-the-scenes operation of adding the course from the code that updates the user interface. I've also disabled the Add button until the thread completes.

```

// THIS IS AN INADEQUATE SOLUTION!
private void addCourse() {
    Thread thread = new Thread() {
        public void run() {
            panel.setEnabled(CoursesPanel.ADD_BUTTON_NAME, false);
            Course course = basicAddCourse();
            panel.addCourse(course);
            panel.setEnabled(CoursesPanel.ADD_BUTTON_NAME, true);
        }
    };
    thread.start();
}

private Course basicAddCourse() {
try { Thread.sleep(3000); } catch (InterruptedException e) {}
    Course course =
        new Course(
            panel.getText(FieldCatalog.DEPARTMENT_FIELD_NAME),
            panel.getText(FieldCatalog.NUMBER_FIELD_NAME));
    JFormattedTextField effectiveDateField =
        (JFormattedTextField)panel.getField(
            FieldCatalog.EFFECTIVE_DATE_FIELD_NAME);
    Date date = (Date)effectiveDateField.getValue();
    course.setEffectiveDate(date);
    return course;
}

```

Responsiveness

A subtle but real problem exists with this code. It is not thread-safe! Since Swing uses a separate thread known as an event dispatch thread, it is possible

for a user to click the Add button before the panel is completely updated. The user might see unexpected results.

You can rectify this by executing statements to update the user interface in the event dispatch thread. The class `javax.swing.SwingUtilities` contains two methods, `invokeLater` and `invokeAndWait`, to allow this. Each method takes a `Runnable` object that defines the code to execute in the event dispatch thread. You use `invokeLater` when you can allow the `run` method to execute asynchronously (when you don't need to "hold up" activities on the user interface). In our example, you want to use `invokeAndWait`, which results in the `run` method being executed synchronously.

Here's how `addCourse` might look using `invokeAndWait`:

```
private void addCourse() {
    Thread thread = new Thread() {
        public void run() {
            panel.setEnabled(CoursesPanel.ADD_BUTTON_NAME, false);
            try {
                final Course course = basicAddCourse();
                SwingUtilities.invokeAndWait(new Runnable() {
                    public void run() {
                        panel.addCourse(course);
                        panel.setEnabled(CoursesPanel.ADD_BUTTON_NAME, true);
                    }
                });
            } catch (Exception e) {}
        }
    };
    thread.start();
}
```

The big downside is that this change will break the `SisTest` method `testAddCourse`. The test presumes that clicking on Add will block until the course has been added to the panel. As a quick fix, you can have the test wait until elements appear in the panel's table.

```
public void testAddCourse() {
    ...
    JButton button = panel.getButton(CoursesPanel.ADD_BUTTON_NAME);
    button.doClick();

    while (panel.getCourseCount() == 0)
        ;
    Course course = panel.getCourse(0);
    assertEquals("MATH", course.getDepartment());
    assertEquals("300", course.getNumber());
    TestUtil.assertDateEquals(2006, 3, 17, course.getEffectiveDate());
}
```

Responsiveness

The change requires a small addition to CoursesPanel:

```
int getCourseCount() {  
    return coursesTableModel.getRowCount();  
}
```

Remaining Tasks

You've invested a considerable amount of code in this simple interface. Yet it's far from complete. Here is a list of some of the things you might consider coding to complete the interface:

- Clear the text fields when the user presses Add.
- Add a constraint that prevents the user from entering duplicate courses. You would code the logic to check for duplicates in CourseCatalog.
- Add delete button to remove courses. You might allow the user to select multiple rows for deletion.
- Add an update button to make edits to existing courses.
- Install the ability to sort the data in each column.
- Add a numeric filter to limit the user to entering only digits for the course number.
- Set each column width depending upon the average or maximum width of its contents.
- Add a hook that selects the contents of each field as the user tabs or clicks into it. This allows the user to replace the field's contents by simply typing.
- Add mouseover help. As the user moves the mouse over each field, show summary information in either the status bar or in a small pop-up "tool tip."
- Add keyboard help. Respond to the F1 key by popping up help. (Obviously this is more involved and requires an understanding of how to build a help subsystem.)
- Replace the department and/or course number fields with a drop-down list (JComboBox).

Remaining
Tasks

- Add interaction with the preferences subsystem (see Additional Lesson III) to allow the application to “remember” the last position of each window.

I've no doubt left a few features off this list. Building a robust, sophisticated user interface is a lot of work. The absence of any of these features will severely cripple the effectiveness of your application. However, rather than you as a developer trying to figure out what you need, you should treat each of these features as customer requirements. Your customer team needs a qualified expert to design and detail the specifications for the user interface.⁶

Final Notes

A large number of Swing books exist. Many are very thick, suggesting that there is quite a bit to learn about Swing. Indeed, there is. In these short two chapters you have only scratched the surface of Swing.

However, you have seen the basics of how to construct Swing applications. You would be able to build a decent interface with this small bit of information. No doubt you will want to learn about more complex Swing topics, such as tree widgets and drag & drop. A bit of searching should provide you with what you need to know. As always, the Java API documentation is the best place to start. Often the API documentation will lead you to a Sun tutorial on the topic.

I hope you've learned the more important lesson in these two Swing chapters: how to approach building a Swing application using TDD. Unit-testing Swing is often viewed as too difficult, and many shops choose to forgo it altogether. Don't succumb to the laziness: The frequent result of not testing Swing code is the tendency for Swing classes to bloat with easily testable application or model code.

Looking at the resultant view class, CoursesPanel, you should have noticed that it is very small and simple. Other than layout, there is little complexity in the class; it does almost nothing. TDD or not, that is always the

Final Notes

⁶This person can be a developer acting in the role of UI expert for the customer team. Do not, however, underestimate the importance of this role. Most developers, even those who have read a book or two on the topic, don't have a clue how to create an effective user experience.

design goal in a user-interface application: to keep the application and/or business logic out of the view.

Using TDD has pushed you toward this goal. The basic rule of TDD is to test everything that can't possibly break. One way to interpret this rule is "test everything that you can and redesign everything else so it can't possibly break." TDD has led you to create a small view class with very simple, discrete methods that you can easily test. You've also created dumb "reactive" methods that simply delegate their work to another trusted class. These methods can't break.

The shops that choose to not test their user interface classes always regret it. Business logic creeps into the application; application and business logic creeps into the view. The view code becomes a cluttered conglomerate of various responsibilities. Since tests aren't written for the view, a considerable amount of code goes untested. The defect rate rises.

Worse, the human tendency toward laziness takes over. Since no tests for the view exist, a developer often figures that the easiest way to get out of testing is to stuff the code into the view. "Yeah, creating a new class is a pain, and so is creating a new test class, so I'll just dump all the code in this mother Swing class." Doing so is a slippery slope that quickly degrades your application.

In these past two chapters, you've learned some additional guidelines and techniques for building Swing applications using TDD:

- Ensure that your design separates application, view, and domain logic.
- Break down design even further by considering the Single-Responsibility Principle.
- Eliminate the otherwise excessive redundancies in Swing (e.g., use common methods to create and extract fields).
- Use listeners to test abstract reactions of the view (e.g., to ensure that clicking a button results in some action being triggered).
- Use mock classes to avoid Swing rendering.
- Don't test layout.
- Use the Swing robot, but only as a last resort.

Swing was not designed with unit-level testing in mind. Figuring out how to test Swing is a problem-solving exercise. Dig through the various Swing classes to find what you need. Sometimes they'll supply the hooks to help you test; other times they won't. You may need to think outside the box to figure out how to test things!

Final Notes

Additional Lesson III

Java Miscellany

This chapter provides a handful of Java odds and ends. The primary goal of this chapter is twofold:

- It will present you with an overview of some very involved APIs that can take entire books to cover adequately.
- It will discuss a few core Java odds and ends that didn't fit elsewhere in *Agile Java*.

You will learn about:

- JARs
- the finalize method
- regular expressions
- JDBC
- internationalization
- call by reference versus call by value
- Java periphery: quick overviews of various other Java APIs, with pointers to more information

JARs

JARs

When you learned about the Java classpath in Lesson 1, you learned to add class folders (or directories) to your classpath. A class folder contains discrete class files that Java looks for when it needs to load a class.

Your classpath can include *JARs* in addition to, or instead of, class folders. A JAR (Java ARchive) is a single file that collects Java class files and other

resources needed at runtime. You can create JARs using the `jar` command-line utility executable supplied in your Java `bin` directory. You can also create JARs using Ant.

An understanding of JARs is absolutely essential to being able to work in any production Java environment. The usage of JARs falls under the category of deployment, so the discussion of JARs didn't have an appropriate place earlier in the book.

The `jar` utility combines class files and other resources (such as images or property files) into a single file with an extension of `.jar`. The utility creates JARs as ZIP files, a file format you are likely familiar with. A ZIP file uses a standard compression algorithm to reduce the overall space requirements.

There are many reasons to use JARs:

- Deployment speed. Transferring or installing a few files is considerably faster than transferring or installing thousands of files.
- Compression. Disk requirements are smaller; smaller files are downloaded faster.
- Security. When working in secure environments, you can attach a digital signature to a JAR.
- Pluggability. You can organize JARs around reuse interests. JARs can be components or complete API sets.
- Versioning. You can attach vendor and version information to JARs.
- Simplified execution. You can designate a JAR to be “Java executable.” A user of the JAR does not need to know an entry class name.

JARs are the basis for deploying classes to a web server or enterprise Java installation.

To create a JAR for the SIS application, first navigate to the `classes` directory. Then execute:

JARs

```
jar cvf sis.jar *
```

In the above `jar` command, the options `c`, `v`, and `f` tell the JAR command to create a new JAR, to provide verbose output (not necessary but useful), and to use a filename of `sis.jar`. The `*` at the end of the command tells the `jar` program to archive all files in the current directory and all subdirectories. You can execute the command `jar` with no arguments to display a brief summary of the command and its proper usage.

After executing the `jar cvf` command, you should have a file named `sis.jar`. Under Windows, you can open this JAR using WinZip. Under any platform, you can use the `jar` command to list the contents of the JAR.

```
jar tvf sis.jar
```

The only difference in the command is the option `t` (list table of contents) instead of `c`. You should see output similar to this:

```
0 Mon Aug 02 23:25:36 MDT 2004 META-INF/
74 Mon Aug 02 23:25:36 MDT 2004 META-INF/MANIFEST.MF
553 Sat Jul 24 10:41:28 MDT 2004 A$B.class
541 Sat Jul 24 10:41:28 MDT 2004 A.class
1377 Sat Jul 24 10:41:28 MDT 2004 AtomicLong.class
0 Sat Jul 24 10:41:28 MDT 2004 com/
0 Sat Jul 24 10:41:28 MDT 2004 com/jimbob/
0 Sat Jul 24 10:41:28 MDT 2004 com/jimbob/ach/
486 Sat Jul 24 10:41:28 MDT 2004 com/jimbob/ach/Ach.class
377 Sat Jul 24 10:41:28 MDT 2004 com/jimbob/ach/AchCredentials.class
516 Sat Jul 24 10:41:28 MDT 2004 com/jimbob/ach/AchResponse.class
1164 Sat Jul 24 10:41:28 MDT 2004 com/jimbob/ach/AchStatus.class
...
...
```

You can extract files from the JAR using the command option `x` (i.e., `jar xvf`).

Once you have added classes to a JAR, you can add the JAR to your class-path. Java will dig into the JAR file to look for classes it needs to load. For example, suppose you deploy `sis.jar` to the directory `/usr/sis`. You can execute the SIS application using the command:

```
java -cp /usr/sis/sis.jar sis.ui.Sis
```

In order for Java to find a class within a JAR, its path information must match the package information. To locate the class `com.jimbob.ach.Ach`, Java must be able to find an entry in the JAR with the complete pathname `com/jimbob/ach/Ach.class`. In the `jar tvf` listing, I've shown this entry in bold.

If you were a directory higher when you were creating the JAR, the path information in the JAR file might be:

```
486 Sat Jul 24 10:41:28 MDT 2004 sis/com/jimbob/ach/Ach.class
```

Java would not match up this entry with the class `com.jimbob.ach.Ach`.

The `jar` utility adds a manifest file, `META-INF/MANIFEST.MF`, to the ZIP file. The manifest file contains information about the contents of the JAR (i.e., meta-information).

One use for the manifest file is to indicate a “main” class. If you specify a main class, you can initiate an application by providing only the JAR name.

You can accomplish this by first creating a separate manifest file with the contents:

```
Main-Class: sis.ui.Sis
```

Make sure you include a blank line as the last line of this file! Otherwise Java may not recognize the manifest entry.

When you create the JAR, specify the manifest file using the command:

```
jar cvmf main.mf sis.jar *
```

The `m` option stands for `manifest`, of course. If you look at the file `MANIFEST.MF` in `sis.jar`, it should look something like this:

```
Manifest-Version: 1.0  
Created-By: 1.5.0 (Sun Microsystems Inc.)  
Main-Class: sis.ui.Sis
```

You can now initiate the SIS application using the simplified command:

```
java -jar sis.jar
```

You can manipulate JARs (and ZIP files) programmatically. Java supplies a complete API in the package `java.util.zip` to help you accomplish this. The `java.util.zip` package contains tools to both read and write JARs. This allows you to write unit tests against JAR operations about as simply as if you were testing against text files.

Regular Expressions

Regular Expressions

A regular expression, also known as a `regex` or `regexp`, is a string containing a search pattern. The regular expression language is a fairly standardized language for pattern-matching specifications. Other languages such as Perl and Ruby provide direct support for regular expressions. Programmers' editors such as TextPad and UltraEdit allow you to search file text using regular expressions. Java supplies a set of class libraries to allow you to take advantage of regular expressions.

You might have used a wildcard character ('`*`') to list files in a directory. Using the wildcard tells the `ls` or `dir` command to find all matching files, assuming that the `*` can expand into any character or sequence of characters. For example, the DOS command `dir *.java` lists every file with a `.java` extension.

The regular expression language is similar in concept but far more powerful. In this section, I'll introduce you to regular expressions using a few simple examples. I'll show you how to take advantage of regular expressions in

your Java code. I'll suggest how you might approach testing Java code that uses regular expressions. But a complete discussion of regular expressions is way out of the scope of this book. Refer to the end of this section for a few sites that provide regular expression tutorials and information.

Splitting Strings

Back in Lesson 7, you used the `String` method `split` to break a full name into discrete name parts. You passed a `String` containing a single blank character to split:

```
for (String name: fullName.split(" "))
```

The `split` method takes a regular expression as its sole argument.¹ The `split` method breaks the receiving string up when it encounters a match for the regular expression.

Suppose you try to split the string "Jeffrey Hyman"² with three spaces separating the first and last name. You want the results to be the strings "Jeffrey" and "Hyman", but a passing language test represents the actual results:

```
public void testSplit() {
    String source = "Jeffrey   Hyman";
    String expectedSplit[] = { "Jeffrey", "", "", "Hyman" };
    assertTrue(
        java.util.Arrays.equals(expectedSplit, source.split(" ")));
}
```

In other words, there are four substrings separated by spaces. Two of those substrings are the empty string (""): The first blank character separates "Jeffrey" from an empty string, the second blank character separates the first empty string from a second empty string, and the third blank character separates the second empty string from "Hyman".

You instead want to split the names around any *groupings* of whitespace characters. Using the Java API documentation for the class `java.util.regex.Pattern` as your guide, you should be able to figure out that the construct `\s` matches a single whitespace character (tab, new line, form feed, space, or carriage return). Under the section Greedy Quantifiers in the API doc, you'll note that `X*` means that a match is found when the construct X occurs at least one time.

Regular
Expressions

¹An overloaded version of `split` takes a limit on the number of times the pattern is applied.

²Aka Joey Ramone. Gabba gabba hey.

Combining the two ideas, the regular expression string `\s+` will match on sequences with one or more whitespace characters.

I've moved the name-splitting code from the `Student` class and into a class called `sis.studentinfo.Name`. You can find the complete code for `NameTest` and `Name` at <http://www.LangrSoft.com/agileJava/code>. The listings below of `NameTest` and `Name` show only relevant or new material.

```
// NameTest.java
public void testExtraneousSpaces() {
    final String fullName = "Jeffrey Hyman";
    Name name = createName(fullName);
    assertEquals("Jeffrey", name.getFirstName());
    assertEquals("Hyman", name.getLastName());
}

private Name createName(String fullName) {
    Name name = new Name(fullName);
    assertEquals(fullName, name.getFullName());
    return name;
}

// Name.java
private List<String> split(String fullName) {
    List<String> results = new ArrayList<String>();
    for (String name: fullName.split(" "))
        results.add(name);
    return results;
}
```

The test method `testExtraneousSpaces` will not pass given the current implementation in `Name.split`. Using the new pattern you learned about, you can update the `split` method to make the tests pass:

```
private List<String> split(String fullName) {
    List<String> results = new ArrayList<String>();
    for (String name: fullName.split("\\s+"))
        results.add(name);
    return results;
}
```

The backslash character ('\\') represents the start of an escape sequence in Java strings. Thus, the regular expression `\s+` appears as `\\s+` within the context of a String.

Since `\s` matches *any* whitespace character, change the test to demonstrate that additional whitespace characters are ignored.

```
public void testExtraneousWhitespace() {
    final String fullName = "Jeffrey \t\t\n \r\fHyman";
    Name name = createName(fullName);
    assertEquals("Jeffrey", name.getFirstName());
    assertEquals("Hyman", name.getLastName());
}
```

Replacing Expressions in Strings

Many applications must capture a phone number from the user. People enter phone numbers in myriad ways. The best place to start is to strip all nondigits (parentheses, hyphens, letters, whitespace, and so on) from the phone number. This is very easy to do using regular expressions. Using a U.S. 10-digit phone number as an example:

```
public void testStripPhoneNumber() {
    String input = "(719) 555-9353 (home)";
    assertEquals("7195559353",StringUtil.stripToDigits(input));
}
```

The corresponding production code:

```
public static String stripToDigits(String input) {
    return input.replaceAll("\\D+", "");
}
```

The `replaceAll` method takes two arguments: a regular expression to match on and a replacement string. The expression `\D` matches any *nondigit* character. The convention in the regular expression language is that the lowercase version of a construct character is used for positive matches and the uppercase version is used for negative matches. As another example, `\w` matches a “word” character: any letter, digit, or the underscore ('_') character. The uppercase construct, `\W`, matches any character that is not a word character.

The Pattern and Matcher Classes

You can use a `JTextPane` as the basis for a Swing-based text editor application. The `JTextPane` allows you to apply styles to sections of text. Many text editors provide the ability to search and highlight all strings matching a pattern. You’ll want to build a class that can manage text searches against the underlying text.

Here’s a test that demonstrates a small bit more of the regular expressions language.

```
package sis.util;
import junit.framework.TestCase;
import java.util.regex.*;

public class RegexTest extends TestCase {
    public void testSearch() {
        String[] textLines =
            { "public class Test {",
              "    public void testMethod() {}",
              "    public void testNotReally(int x) {}",
              "
```

**Regular
Expressions**

```

    "public void test() {}",
    "public String testNotReally() {}",
    "}";
String text = join(textLines);

String testMethodRegex =
    "public\\s+void\\s+test\\w*\\s*\\((\\s*\\)\\)\\s*\\{";
Pattern pattern = Pattern.compile(testMethodRegex);
Matcher matcher = pattern.matcher(text);
assertTrue(matcher.find());
assertEquals(text.indexOf(textLines[1]), matcher.start());
assertTrue(matcher.find());
assertEquals(text.indexOf(textLines[3]), matcher.start());
assertFalse(matcher.find());
}

private String join(String[] textLines) {
    StringBuilder builder = new StringBuilder();
    for (String line: textLines) {
        if (builder.length() > 0)
            builder.append("\n");
        builder.append(line);
    }
    return builder.toString();
}
}

```

The regular expression in `testMethodRegex` looks pretty tough. They can get much worse! The regular expression above doesn't handle the possibility of a static or abstract modifier.³ The double backslashes in the regular expression don't help matters.

Breaking the pattern down into its constructs makes it pretty straightforward. I'll step through each construct, left to right in the expression:

<code>public\\s+void\\s+test\\w*\\s*\\((\\s*\\)\\)\\s*\\{</code>	
match the text “public”	<code>public</code>
match one or more whitespace characters	<code>\\s+</code>
match the text “void”	<code>void</code>
match one or more whitespace characters	<code>\\s+</code>
match the text “test”	<code>test</code>

Regular Expressions

³Perhaps regex isn't the best tool for this job.

match zero or more word characters	<code>\w*</code>	An asterisk indicates that there may be any number of the preceding construct (including 0).
match zero or more whitespace characters	<code>\s*</code>	
match a left parenthesis	<code>\(</code>	You must escape parenthesis characters and brace characters.
match one or more whitespace characters	<code>\s*</code>	
match a right parenthesis	<code>\)</code>	
match zero or more whitespace characters	<code>\s*</code>	
match a left brace	<code>\{</code>	

A regular expression string is free-format, unproven text. You must compile it first using the `Pattern` class method `compile`. A successfully compiled regular expression string returns a `Pattern` object. From a `Pattern` object, you can obtain a `Matcher` for a given input `String`. Once you have a `Matcher` object, you can send it the `find` message to get it to locate the next matching *subsequence* (the substring of the input `String` that matches the regular expression). The `find` method returns `true` if a match was found, `false` otherwise.

The `Matcher` instance stores information about the last found subsequence. You can send the `Matcher` object the messages `start` and `end` to obtain the indexes that delineate the matched subsequence. Sending the message `group` to the `Matcher` returns the matching subsequence text.

In addition to triggering a `find`, you can send the message `matches` to the `Matcher`. This returns `true` if and only if the entire input string matches the regular expression. You can also send `lookingAt`, which returns `true` if the start of the input string (or the entire string) matches the regular expression.

Regular
Expressions

For More Information

As you've seen, testing regular expressions is extremely simple—the assertions compare only `String` objects. This brief section should have given you

enough information about regular expressions to be able to dig further into Java's support for it. The API documentation for the class `java.util.regex.Pattern` is the best place to start.

The regular expression language is a bit more involved. Knowing how patterns match against text is necessary to understanding how to write an appropriate regular expression.

The Sun tutorial on regular expressions at the web page <http://java.sun.com/docs/books/tutorial/extra/regex/> supplies information on a few additional regular expression topics. You may also want to visit

<http://www.regular-expressions.info/>

<http://www.javaregex.com/>

Cloning and Covariance

Java allows you the ability to *clone*, or make a copy of, an object. You might find it surprising that cloning is only rarely a useful thing to do. In my numerous years of professional Java development, I've only seen a few needs for cloning. Therefore, I'm relegating cloning to a brief overview in this final section.

Unfortunately, cloning is often misunderstood and poorly implemented. If you have more sophisticated cloning needs than this simple example demonstrates, I refer you *Effective Java*⁴ for an excellent discussion of the many issues related to cloning.

As an example, let's provide cloning capability for the simple class `Course`. The test is straightforward:

```
public void testClone() {
    final String department = "CHEM";
    final String number = "400";
    final Date now = new Date();
    Course course = new Course(department, number);
    course.setEffectiveDate(now);
    Course copy = course.clone();
    assertFalse(copy == course);
    assertEquals(department, copy.getDepartment());
    assertEquals(number, copy.getNumber());
    assertEquals(now, copy.getEffectiveDate());
}
```

Cloning and Covariance

⁴[Bloch2001].

Even though `clone` is defined on the class `Object`, the test won't compile. `Object` defines the `clone` method as `protected`. You must override the `clone` method in the `Course` subclass and make it `public`. The `clone` method is defined in `Object` as:

```
protected native Object clone() throws CloneNotSupportedException;
```

The `native` keyword in the signature indicates that the method is not implemented in Java but instead in the JVM.

To be able to clone an object, its class must implement the `Cloneable` interface. `Cloneable` is a marker interface—it doesn't define `clone`, as you might expect. Instead, its goal is to prevent clients from cloning a class that was not explicitly designed to be cloneable.

The implementation in `Course`:

```
package sis.studentinfo;
...
public class Course implements java.io.Serializable, Cloneable {
    private String department;
    private String number;
    private Date effectiveDate;
    ...
    @Override
    public Course clone() {
        Course copy = null;
        try {
            copy = (Course)super.clone();
        }
        catch (CloneNotSupportedException impossible) {
            throw new RuntimeException("unable to clone");
        }
        return copy;
    }
}
```

The first thing you might notice is that the `clone` method returns an object of the type `Course`—not `Object`, as defined in the superclass. This is a capability of Java known as *covariance*. Covariance is the ability of a subclass method to return an object that is a subclass of the superclass return type. Cloning is the canonical example for covariance in Java.

The core of the `clone` method is the call to the superclass `clone` method. You always want to call the superclass `clone` method. The `clone` method as implemented in `Object` (in the VM actually) performs a bitwise copy of the contents of the object. This means that it copies values for every field defined on the class. References are copied as well, but the objects they refer to are not copied. For example, for the `effectiveDate` field in `Course`, both the original and the `clone` refer to the same `Date` object.

Cloning and Covariance

The default clone behavior, in which references are copied but not the objects to which they point, is known as a *shallow clone*. If you want a *deep clone*—a copy where the contained objects are copied also—you must implement the details yourself in the `clone` method.

The call to `super.clone` can throw a `CloneNotSupportedException`. This is what requires your class to implement the `Cloneable` marker interface.

JDBC

The vast majority of business-oriented applications access relational databases such as Oracle, MySQL, Sybase, SQL Server, and DB2. Applications require information to be available and dependable from execution to execution—to be *persistent*. A relational database⁵ provides an application-independent means of storing and accessing that information.

A relational database is comprised of tables. A table is a matrix of rows and columns. A column is an attribute. A row is a collection of column values. For example, a student table might have columns ID, name, and city. A row exists in the database for each student. A representation of a student table appears in Table 1. In the table, there are two rows and thus two students: Schmoo, living in the city of Pueblo, and Caboose, living in the city of Laurel.

The language SQL (Structured Query Language) allows you to store and retrieve information from any relational database. SQL, for the most part, is standard across database implementations. An SQL statement to access student records from an Oracle database will work for accessing student records from Sybase (with perhaps minor modifications). In Java, you use an API called JDBC (Java DataBase Connectivity) to interact with the database.

Table 1 A Student Table with Two Rows

JDBC

ID	Name	City
221-44-4400	Schmoo	Pueblo
234-41-0001	Caboose	Laurel

⁵You may hear the term DBMS—database management system—to refer to the software that manages the database. An RDBMS is a relational DBMS.

JDBC allows you to establish connections to the database and execute SQL against it.

The need for database interaction is so commonplace that many Java products exist to simplify it. Enterprise Java Beans (EJBs), Hibernate, and JDO are three of the more popular attempts to simplify Java persistence. JDBC is the foundation of many of these tools. As is often the case, you are better off learning how the foundation works before considering use of a higher-level tool. Doing so will give you a better understanding of what the tools are doing. You may even find that a custom, highly refactored JDBC implementation is the better solution.

In this brief section, I'll demonstrate how to write simple JDBC tests and code to interact with a database. Just like Swing, entire books and many good web sites on JDBC exist. You will want to consult additional resources for more details on JDBC. You will also want to consult additional resources on SQL. I will provide only minimal explanation about the SQL in this section.

The examples in this section are written to interact with MySQL, a database freely available at <http://www.mysql.com>. Most of the examples are applicable to any other database. However, some of the code details for connecting to the database will differ. Also, a few of the “getting started” tests make presumptions about the availability of test structures within the database.

You will also need a JDBC *driver* for MySQL (see <http://dev.mysql.com/downloads/connector/j/3.0.html>). A driver is a Java library that meets Sun's JDBC interface specification. Your code always interacts with the same Sun-supplied JDBC interface methods, regardless of the database or database driver vendor. The database driver implementation adapts JDBC API calls to the specific requirements of the database itself.

Connecting to a Database

Perhaps the most difficult aspect of interacting with databases is establishing a connection to them. Installing MySQL always creates a database named test that you can access. Your first test will ensure that you can access the database test.

Make sure you have installed the MySQL JDBC driver and added it to your classpath. Note that you won't need the driver on the classpath to compile, but you will need it to execute.

```
package sis.db;  
  
import junit.framework.TestCase;  
import java.sql.*;
```

JDBC

```

public class JdbcAccessTest extends TestCase {
    public void testConnection() throws SQLException {
        JdbcAccess access = new JdbcAccess("test");
        Connection connection = null;
        try {
            connection = access.getConnection();
            assertFalse(connection.isClosed());
        }
        finally {
            connection.close();
        }
    }
}

```

The test is simple enough. Create a `JdbcAccess` instance using the database name `test`. Request a connection from the `access` object and verify that the connection is open (not closed). Make sure the connection gets closed by using a `finally` block.

A couple of comments: First, the test itself throws `SQLException`. You'll want to quickly encapsulate `SQLException`—you want as few classes as possible to even know that you are using JDBC.



Isolate knowledge of JDBC in your system to one class.

Second, you will not usually want client code to request its own connection object. A common problem in many systems is that the client code forgets to close the connection. Clients open more and more connections. Ultimately, the system crashes when it becomes unable to supply any more connections. For now, writing a test to prove the ability to establish a connection allows you to proceed incrementally. You probably won't want to publicize `getConnection`.

```

package sis.db;

import java.sql.*;

public class JdbcAccess {
    private String database;

    public JdbcAccess(String database) {
        this.database = database;
    }

    Connection getConnection() throws SQLException {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (Exception cause) {
            throw new SQLException(cause.getMessage());
        }
    }
}

```

JDBC

```

        String url = "jdbc:mysql://localhost/" + database;
        return DriverManager.getConnection(url);
    }
}

```

The implementation of `getConnection` in `JdbcAccess` involves two steps: first, load the JDBC driver class. Second, establish a connection by passing a database URL to the `DriverManager` class method `getConnection`.

The suggested technique for loading the driver is to use reflection. You pass a `String` representing the name of the class to the `Class.forName` class method. Some drivers require you to send `newInstance` to actually create a driver instance.⁶ But there's no reason you couldn't directly create a new instance of it:

```

Connection getConnection() throws SQLException {
    new com.mysql.jdbc.Driver();
    String url = "jdbc:mysql://localhost/" + database;
    return DriverManager.getConnection(url);
}

```

The original reason for suggesting the use of reflection is flexibility. Using `Class.forName`, you could load the driver class name from a property file or pass it in via a system property (see Properties in this chapter). You would then have the flexibility to change the driver at a later time without changing code. In reality, changing drivers doesn't happen that frequently, and you'll probably have to revisit your code anyway if you do.⁷

When you call `Class.forName` (or when you first reference and instantiate the class), the static initializer in the `Driver` class is invoked. Code in the static initializer is responsible for registering the driver with the `DriverManager`. The `DriverManager` maintains a list of all registered drivers and selects a suitable one based on the URL when `getConnection` gets called.

Another means of specifying one or more drivers is to supply class names using the system property `jdbc.drivers`. Refer to the Properties section later in this chapter for information on how to set this property.

The organization of information in the database URL is specific to the driver vendor. It usually follows a few conventions, such as starting with the word `jdbc` followed by a subprotocol string (`mysql` in this case; it's often the vendor name). You separate elements in the URL using colons (:). In MySQL, you follow the subprotocol with a reference to the server (localhost here—your machine) and the database name.

JDBC

⁶MySQL does not. Check your driver documentation. Or you can be lazy and always call `newInstance`.

⁷Of course, the need to support a second driver is reason enough to eliminate the hard-coded class reference.

The end result of calling `getConnection` is a `java.sql.Connection` object. You'll need a connection in order to execute any SQL statements.

Connections are scarce resources. You will be able to create only a limited number of connections. Further, establishing a new connection is costly from a performance standpoint. *Connection pools* allow you to maintain a number of always-open connections. A client requests a connection from the pool, uses it, then relinquishes the connection so it can return to the pool. Some JDBC implementations support connection pools directly; others do not. For purposes of these exercises and for simple, single-client access, connection pools are not necessary.

Executing Queries

The second test demonstrates executing SQL statements.

```
package sis.db;

import junit.framework.TestCase;
import java.sql.*;

public class JdbcAccessTest extends TestCase {
    private JdbcAccess access;

    protected void setUp() {
        access = new JdbcAccess("test");
    }
    ...
    public void testExecute() throws SQLException {
        access.execute("create table testExecute (fieldA char)");
        try {
            assertEquals("fieldA",
                access.getFirstRowFirstColumn("desc testExecute"));
        }
        finally {
            access.execute("drop table testExecute");
        }
    }
}
```

JDBC

You cannot assume the existence of any tables in the test database, so `testExecute` creates its own. The SQL statement "create table `testExecute` (`fieldA` char)" creates a table named `testExecute` with one `char` (character) column named `fieldA`.

The proof that the table was created is via a method named `getFirstRowFirstColumn`. The query `desc testExecute` returns a description of the `testExecute` table. The SQL command `desc` is short for `describe`. A `desc` command returns one row for each column that it describes. The first column in each row is the name of the column.

As a final measure, the test removes the `testExecute` table by using the SQL command `drop table`.

A slightly refactored `JdbcAccess` shows the new methods `execute` and `getFirstRowFirstColumn`. The refactoring eliminates some code duplication. It also eliminates processing duplication by ensuring that the driver instance only gets loaded once. Both the `execute` and `getFirstRowFirstColumn` methods obtain and relinquish their own connection. This alleviates the client from that responsibility.

```
package sis.db;

import java.sql.*;

public class JdbcAccess {
    private String url;
    ...
    public void execute(String sql) throws SQLException {
        Connection connection = getConnection();
        try {
            Statement statement = connection.createStatement();
            statement.execute(sql);
        }
        finally {
            close(connection);
        }
    }

    private void close(Connection connection) throws SQLException {
        if (connection != null)
            connection.close();
    }

    Connection getConnection() throws SQLException {
        if (url == null) {
            loadDriver();
            url = "jdbc:mysql://localhost/" + database;
        }
        return DriverManager.getConnection(url);
    }

    private void loadDriver() throws SQLException {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (Exception cause) {
            throw new SQLException(cause.getMessage());
        }
    }

    public String getFirstRowFirstColumn(String query)
        throws SQLException {
        Connection connection = getConnection();
```

```
try {
    Statement statement = connection.createStatement();
    ResultSet results = statement.executeQuery(query);
    results.next();
    return results.getString(1);
}
finally {
    close(connection);
}
}
```

You need a Statement object in order to execute SQL. You obtain a Statement object from a Connection by sending it `createStatement`. Once you have a Statement, you can either use the `execute` method to invoke SQL statements where you expect no results to come back or `executeQuery` to invoke SQL statements and subsequently obtain results. To `execute` or `executeQuery`, you pass the SQL string as an argument.

Results return in the form of a `ResultSet` object. A `ResultSet` is very similar to an `Iterator`. It maintains an internal pointer to the current row. You advance this pointer, which initially points to nothing, by sending the message `next` to the `ResultSet`. The method `next` returns `true` until the internal pointer moves past the last row. You access columns from the current row by either column name or column index. Various methods exist for obtaining column values, each differing by the type of data stored in each column. The `ResultSet` method `getString` returns character data, for example, and the method `getInt` returns an `int` value.

The code in `getFirstRowFirstColumn` advances the pointer to the first row returned from `executeQuery` by calling `results.next()`. The `getFirstRowFirstColumn` method returns the value of the first column by sending `getString` to the `ResultSet` using its index, 1, as an argument. Column indexes start at 1 in JDBC, not 0.

Prepared Statements

JDBC

Often you will want to execute the same statement numerous times in an application, different only perhaps by key information. For example, the SIS application requires rapid student lookups based on student identification (ID) number.

Since you pass an SQL statement in the form of a String, the statement must be compiled before the database can execute it. Compiling the statement ensures that it follows valid SQL syntax. Compilation takes enough execution time that it can cause performance problems. (Profile, as always, to be sure that you have poor performance.) To speed things up, you can create

a PreparedStatement object using an SQL string. A PreparedStatement compiles the SQL once and stores this compiled version for later, more rapid use.

You can insert placeholders in the SQL string in the form of question marks ('?'). Later, you bind values to these placeholders. An appropriate SQL string for the student lookup might be:

```
select id, name from testQueryBy where id = ?
```

For each student lookup, you bind a value to the question mark by using set methods. The method statement.setString(1, "boo") would bind the value 'boo' to the first (and only) question-mark argument.

```
public void testQueryBy() throws SQLException {
    drop("testQueryBy");
    access.execute(
        "create table testQueryBy (id varchar(10), name varchar(30))");
    PreparedStatement statement = null;

    try {
        access.execute("insert into testQueryBy values('123', 'schmoe')");
        access.execute(
            "insert into testQueryBy values('234', 'patella')");

        statement =
            access.prepare("select id, name from testQueryBy where id = ?");
        List<String> row = access.getUnique(statement, "234");
        assertEquals("234", row.get(0));
        assertEquals("patella", row.get(1));

        row = access.getUnique(statement, "123");
        assertEquals("123", row.get(0));
        assertEquals("schmoe", row.get(1));
    }
    finally {
        statement.close();
        drop("testQueryBy");
    }
}

private void drop(String tableName) {
    try {
        access.execute("drop table " + tableName);
    }
    catch (SQLException ignore) {
        // exception thrown if table doesn't exist; we don't care
    }
}
```

Since the point of using PreparedStatement is client efficiency, you do not want to close the connection each time you execute a query against the Pre-

paredStatement. In this case, you will have to trust that client code closes out the connection when it is done using the PreparedStatement.

The implementation:

```
public PreparedStatement prepare(String sql) throws SQLException {
    Connection connection = getConnection();
    return connection.prepareStatement(sql);
}

public List<String> getUnique(
    PreparedStatement statement, String... values)
    throws SQLException {
    int i = 1;
    for (String value: values)
        statement.setString(i++, value);
    ResultSet results = statement.executeQuery();
    results.next();

    List<String> row = new ArrayList<String>();
    ResultSetMetaData metadata = results.getMetaData();
    for (int column = 1; column <= metadata.getColumnCount(); column++)
        row.add(results.getString(column));
    return row;
}
```

From a ResultSet, you can extract metadata. The ResultSetMetaData object supplies useful information about the results returned: number of columns, data type for each column, value for each column, and so on. The implementation of getUnique obtains the number of columns using getColumnCount and uses this to iterate through all the column values.

Metadata is also available at the database level. You can send the message getMetaData to a Connection object. In return you receive a DatabaseMetaData object populated with a wealth of information about the database and drivers, including tables and columns, versions, driver limitations, and database limitations.

JDBC

JDBC Application Design

There are dozens of ways to implement persistence using JDBC in an application. As mentioned earlier, you can use a tool that sits atop JDBC or even supplants it. Or, following TDD, you can grow a similar tool using the philosophy of zero duplication. By being vigilant about squashing duplication, you can quickly build a tool to meet your needs. Unlike a third-party tool, yours will be flexible enough to change rapidly based on demand.

You can use the example above as a starting point for your own tool. The JdbcAccess class should end up being the only place in which you interact

with the JDBC API. Most of the remainder of the application is blissfully ignorant of the fact that JDBC is the persistence mechanism. If you choose to supplant JDBC with an object-oriented database down the road, replacement work is minimal.

You can create mapping objects to translate between database columns and domain attributes. You can accomplish this using code generation (which is preferable) or reflection. It is even possible to store everything in String format in the database and do type translation a layer up from JdbcAccess. This can help keep JdbcAccess simple: It provides access methods that return generic lists of lists of strings (rows of columns, each of which are strings).

SQL statements contain rampant duplication. It is fairly easy to code a SQL generator class that builds SQL statements using information available from the database (the database metadata).

Internationalization

A significant amount of software product gets deployed to all corners of the globe. No longer can you get away with developing an application for only your home language. Some localities, such as Quebec, require all software to execute in two or more languages. Success in the global marketplace hinges on the ability to sell your software in many countries using many different languages.

Building software so that it supports different languages and cultures is known as *internationalization*, or *i18n*⁸ in shorthand. Preparing software for delivery to a single locale is known as *localization*. A *locale* is a region. Geography, culture, or politics can define locales.

Retrofitting a large, existing application to support multiple languages can be expensive. This might lead you to believe that you must internationalize your application from the outset of development. Perhaps. Extreme adherence to the “no duplication” rule can leave you with a design that supports a rapid transition to internationalized software. Nonetheless, building a foundation for internationalization *can* make things easier. You can justify these building blocks because they do help eliminate duplication.



Use internationalization mechanisms to help eliminate duplication and position you to localize your application.

Internationalization

⁸Count the number of letters between the first and last letters of internationalization.

Resource Bundles

String literals embedded directly within code cause problems. Most significant, they create duplication. If a literal appears in production code and if you've been testing everything like you should, some test will duplicate that same literal. Also, the meaning of the literal is not always clear. In Agile Java, I had you extract String literals to class constants. Both the test and production code can refer to the same constant. The name of the constant imparts meaning.

Still, having to maintain a bunch of class constants is painful. Consider also the needs of the international application. The ideal solution involves extracting all literals to a common file. When you must deploy the software to a different locale, you provide a new file that contains the translated text.

Instances of the Java class `java.util.ResourceBundle` manage interaction with locale-specific resource files. To internationalize your software, you'll update all code to ask a `ResourceBundle` for a localized message String.

Using a `ResourceBundle` is pretty easy. You obtain a `ResourceBundle` by calling the creation method `getBundle`, passing it the name of the bundle. You then work with the `ResourceBundle` like a map—you pass it a key, it gives you a localized object in return.

Testing use of a `ResourceBundle` is another matter. You want to ensure that you can read a given property and its value from a `ResourceBundle`. But you can't presume that a key and certain value already exists in the resource file. The easiest solution would be to write out a small file with a known key and value. The problem is, you don't want to overwrite the resource file that the rest of your application requires.

```
package sis.util;
import junit.framework.TestCase;
import java.io.IOException;

public class BundleTest extends TestCase {
    private static final String KEY = "someKey";
    private static final String VALUE = "a value";
    private static final String TEST_APPEND = "test";
    private static final String FILENAME =
        "./classes/sis/util/" + Bundle.getName() + "Test.properties";
    private String existingBundleName;

    protected void setUp() {
        TestUtil.delete(FILENAME);
        existingBundleName = Bundle.getName();
        Bundle.setName(existingBundleName + TEST_APPEND);
    }

    protected void tearDown() {
        Bundle.setName(existingBundleName);
    }
}
```

Internationalization

```

        TestUtil.delete(FILENAME);
    }

    public void testMessage() throws IOException {
        writeBundle();
        assertEquals(VALUE, Bundle.get(KEY));
    }

    private void writeBundle() throws IOException {
        LineWriter writer = new LineWriter();
        String record = String.format("%s=%s", KEY, VALUE);
        writer.write(FILENAME, record);
    }
}

```

The test carefully manages a test resource file using `setUp` and `tearDown`. It interacts with a class you will create named `sis.util.Bundle` to retrieve the name of the existing bundle. The test stores the existing resource file base name, then tells `Bundle` to use a new base name. Only the test will recognize this base name. The test itself (`testMessage`) writes to the test resource file. Each entry in the resource file is a key-value pair separated by an equals sign (=).

The assertion in `testMessage` calls the `get` method on the `Bundle` class to retrieve a localized resource. Finally, the `tearDown` method resets `Bundle` to use the original base name.

```

package sis.util;

import java.util.ResourceBundle;

public class Bundle {
    private static String basePath = "Messages";
    private static ResourceBundle bundle;

    static String getName() {
        return basePath;
    }

    static void setName(String name) {
        basePath = name;
        bundle = null;
    }

    public static String get(String key) {
        if (bundle == null)
            loadBundle();
        return (String)bundle.getString(key);
    }

    private static void loadBundle() {
        bundle = ResourceBundle.getBundle("sis.util." + getName());
    }
}

```

Internationalization

The method `loadBundle` obtains a `ResourceBundle` instance by base name. The base name is similar to a fully qualified class name. Presume that you have a resource file with a fully qualified path name of `./classes/sis/util/Messages.properties`. Also presume that the directory `./classes` appears on the classpath. In order for `getBundle` to locate this resource file, the base name must be `"sis.util.Messages"`.

Once you have a `ResourceBundle` object, you can send it a few different messages to extract localized resources. The message `getString` returns localized text.

Once you've implemented the `Bundle` class, you would replace occurrences of literals in your application to use its `get` method. You will end up with a potentially large file of key-value pairs. If the file size makes management of the resource file unwieldy, consider partitioning it into multiple resource bundles.

Localization

Suppose you must deploy your application to Mexico. You would send your resource file to a translator for localization. The translator's job is to return a new file of key-value pairs, replacing the base value (perhaps English) with a Spanish translation.

(In reality, the translator will probably need to interact with you or someone else who understands the application. The translator will often require contextual information because of multiple word meanings. For example, your translator may need to know whether "File" is a verb or a noun.)

You must name the file that the translator returns to you according to the specific Locale. A locale is a language, a country (which is optional), and a variant (also optional; it is rarely used). For deployment to Mexico, the language is Spanish, indicated by `"es"` (`español`). The country is `"MX"` (`Mexico`). To come up with a resource file name, you append these bits of locale information to the base name, using underscores (`'_'`) as separators. The Mexican-localized file name for the base name `"Messages"` is `"Messages_es_MX.properties"`.

You can request a complete list of locales supported on your platform using the `Locale` class method `getAvailableLocales`. Code and execute this ugly bit of code:

```
for (Locale locale: Locale.getAvailableLocales())
    System.out.println(String.format("%s %s: use '_%s'", 
        locale.getDisplayLanguage(), locale.getDisplayCountry(),
        locale.getLanguage(),
        (locale.getCountry().equals("") ? "" : "_" + locale.getCountry())));
```

The output from executing this code snippet will give you appropriate extensions to the base name.

Here is BundleTest, modified to include a new test for Mexican localization:

```
package sis.util;

import junit.framework.TestCase;
import java.io.IOException;
import java.util.Locale;

public class BundleTest extends TestCase {
    private static final String KEY = "someKey";
    private static final String TEST_APPEND = "test";
    private String filename;
    private String existingBundleName;

    private void prepare() {
        TestUtil.delete(filename);
        existingBundleName = Bundle.getName();
        Bundle.setName(existingBundleName + TEST_APPEND);
    }

    protected void tearDown() {
        Bundle.setName(existingBundleName);
        TestUtil.delete(filename);
    }

    public void testMessage() throws IOException {
        filename = getFilename();
        prepare();
        final String value = "open the door";
        writeBundle(value);
        assertEquals(value, Bundle.get(KEY));
    }

    public void testLocalizedMessage() throws IOException {
        final String language = "es";
        final String country = "MX";
        filename = getFilename(language, country);
        prepare();

        Locale mexican = new Locale(language, country);
        Locale current = Locale.getDefault();
        try {
            Locale.setDefault(mexican);
            final String value = "abre la puerta";
            writeBundle(value);
            assertEquals(value, Bundle.get(KEY));
        }
        finally {
            Locale.setDefault(current);
        }
    }
}
```

Internationalization

```

    }

}

private void writeBundle(String value) throws IOException {
    LineWriter writer = new LineWriter();
    String record = String.format("%s=%s", KEY, value);
    writer.write(getFilename(), record);
}

private String getFilename(String language, String country) {
    StringBuilder builder = new StringBuilder();
    builder.append("./classes/sis/util/");
    builder.append(Bundle.DEFAULT_BASE_NAME);
    builder.append("Test");
    if (language.length() > 0)
        builder.append("_" + language);
    if (country.length() > 0)
        builder.append("_" + country);
    builder.append(".properties");
    return builder.toString();
}

private String getFilename() {
    return getFilename("", "");
}
}

```

BundleTest requires refactoring to support the changes to the test resource filename format. You cannot use the `setUp` method to delete the test resource file, since its name must change. I renamed `setUp` to `prepare` (an adequate name at best). The `prepare` method assumes that the field `filename` is populated with the appropriate test resource filename.

The new test, `testLocalizedMessage`, writes test data to a file using a name encoded with the language and country. The test creates a `Locale` object using the same language and country. It then obtains the current (default) `Locale` so that the test can reset the `Locale` to this default when it completes. As the final preparation before testing the `Bundle.get` method, the test sets the default `Locale` to the newly created Mexican `Locale` object.

Setting a `Locale` is a global change. Any code that might need to be internationalized can request the current `Locale` so that it knows what to do. Most of the time, this is handled for you. In the case of `ResourceBundle`, you don't need to do anything. To load a bundle, you supply only the base name. Code in `ResourceBundle` retrieves the default `Locale` and combines information from it with the base name to create a complete resource file name.

Formatted Messages

Entries in resource files may contain substitution placeholders. For example:

`dependentsMessage=You have {0} dependents, {1}. Is this correct?`

The value for `dependentsMessage` contains two *format elements*: `{0}` and `{1}`. Code that loads this string via a `ResourceBundle` would use a `MessageFormat` object to replace the format elements with appropriate values. The following language test demonstrates:

```
public void testMessageFormat() {
    String message = "You have {0} dependents, {1}. Is this correct?";

    MessageFormat formatter = new MessageFormat(message);
    assertEquals(
        "You have 5 dependents, Señor Wences. Is this correct?",
        formatter.format(message, 5, "Señor Wences"));
}
```

In older code using `MessageFormat`, you might see the arguments wrapped in an `Object` array:

```
formatter.format(new Object[] {new Integer(5), "Señor Wences"})
```

Why not just use the `java.util.Formatter` class? The reason is that the `MessageFormat` scheme predates the `Formatter` class, which Sun introduced in Java 5.0.

A more interesting reason for using the `MessageFormat` scheme is to take advantage of the `ChoiceFormat` class. This can eliminate the need to code additional `if/else` logic in order to supply format element values. The classic example shows how to manage formatting of messages that refer to numbers. We want to present a friendlier message to Señor Wences, and telling him that he has “one dependents” is unacceptable.⁹

Using `ChoiceFormat` is a bit of a bear. You first create a mapping of numeric ranges to corresponding text. You refer to the ranges, expressed using an array of doubles, as limits. You express the corresponding text, or formats, in an array of strings.

```
double[] dependentLimits = { 0, 1, 2 };
String[] dependentFormats =
    { "no dependents", "one dependent", "{0} dependents" };
```

The numbers in the limits array are ranges. The final element in the limits array represents the low end of the range—two dependents and up, in this case. For two dependents, the `ChoiceFormat` would apply the corresponding format string `"{0} dependents"`, substituting the actual number of dependents for `{0}`.

Internationalization

Using these limits and formats, you create a `ChoiceFormat` object:

```
ChoiceFormat formatter =
    new ChoiceFormat(dependentLimits, dependentFormats);
```

⁹No doubt he would have responded indignantly, “Tell it to the hand.”

A message string might have multiple format elements. Only some of the format elements might require a ChoiceFormat. You must create an array of Format objects (Format is the superclass of ChoiceFormat), substituting null where you have no need for a formatter.

```
Format[] formats = { formatter, null };
```

You can then create a MessageFormat object and set the array of formats into it:

```
MessageFormat messageFormatter = new MessageFormat(message);
messageFormatter.setFormats(formats);
```

You're now able to use the MessageFormat as before. The entire process is shown in the language test `testChoiceFormat`.

```
public void testChoiceFormat() {
    String message = "You have {0}, {1}. Is this correct?";
    double[] dependentLimits = { 0, 1, 2 };
    String[] dependentFormats =
        { "no dependents", "one dependent", "{0} dependents" };

    ChoiceFormat formatter =
        new ChoiceFormat(dependentLimits, dependentFormats);

    Format[] formats = { formatter, null };

    MessageFormat messageFormatter = new MessageFormat(message);
    messageFormatter.setFormats(formats);

    assertEquals(
        "You have one dependent, Señor Wences. Is this correct?",
        messageFormatter.format(new Object[] { 1, "Señor Wences" }));

    assertEquals(
        "You have 10 dependents, Señor Wences. Is this correct?",
        messageFormatter.format(new Object[] { 10, "Señor Wences" }));
}
```

Interaction with ChoiceFormat can be even more complex. Refer to the Java API documentation for the class.

Note that the format strings should come from your ResourceBundle.

Other Areas to Internationalize

The formatting of numbers, dates, and currencies can vary depending upon the locale. Display numbers using separators every three digits in order to make them more readable. The separators are commas (,) in the United States, while some other localities use periods (.). Dates in the United States

often appear in month-day-year order, separated by slashes or hyphens. Dates elsewhere may appear in a different order, such as year-month-day.

The classes `java.text.NumberFormat`, `java.util.Calendar`, and `java.text.DateFormat` and their subclasses, support the use of `Locale` objects. Your code that works with these classes should request instances using factory methods. Your code should not directly instantiate these classes, otherwise you will not get the default `Locale` support. For example, to obtain a `SimpleDateFormat` instance with support for the default `Locale`:

```
DateFormat formatter = SimpleDateFormat.getDateInstance();
```

The `Calendar` class also allows you to manage time zones.

Various layout managers in Java supply support for different axis orientation. Some non-Western cultures orient text so that it reads from top to bottom instead of left to right. For example, the `BoxLayout` class allows you to organize components within a container around a `LINE_AXIS`. Each `Locale` contains a `java.awt.ComponentOrientation` object; this object defines whether text reads from right to left or left to right (Western) and whether lines are horizontal (Western) or vertical. At runtime, then, the `BoxLayout` obtains the `ComponentOrientation` in order to determine whether `LINE_AXIS` corresponds to the `x` axis or the `y` axis.

Other languages require the use of different character sets. Since Java uses Unicode, you generally don't have to concern yourself with the different character sets. However, if you need to sort text elements, you will have to work with collation sequences. The class `java.text.RuleBasedCollator` provides the basis for this support. The API documentation for this class contains a good overview of collation in Java.

Call by Reference versus Call by Value

Much ado is made over whether Java uses “call by reference” or “call by value.” These classic software development phrases describe how a language manages the passing of arguments to functions (perhaps in C) or methods. The terminology (or people's interpretations of the terminology) is not necessarily important. What's important is that you understand how Java manages the passing of arguments.

**Call by
Reference
versus Call
by Value**

Call by value means that the called function¹⁰ makes a copy of an argument behind the scenes. Code in the function operates on the copy. The implication is that any changes made to the argument are discarded when the

¹⁰I use the term “function” in these general definitions to imply that this is not necessarily how Java does it.

function completes execution. The reason is because changes are not actually made to the argument but to its local copy. The copy of the argument has only method scope.

Call by reference means that the function works on the same physical argument that was passed in. Any changes to the argument are retained.

Java is entirely call by value. If you pass an `int` to a method, the method operates on a copy of the `int`. If you pass an object reference to a method, the method operates on a copy of the *reference—not a copy of the object itself*. I'll demonstrate both of these statements in code.

Understanding call by value with respect to primitives is easy enough. Here's a test to show you how the method's copy of the primitive gets discarded.

```
public void testCallByValuePrimitives() {
    int count = 1;
    increment(count);
    assertEquals(1, count);
}

private void increment(int count) {
    count++;
}
```

Even though the `increment` method changes the value of `count`, the change is to a local copy. The local copy disappears upon exit of the `increment` method. Code in `testCallByValuePrimitives` remains oblivious to any of these shenanigans.

Remember that a reference is a pointer—an address of an object's location in memory. The called method copies this address, creating a new pointer that refers to the same memory location. If you assign a new memory address (i.e., a different object) to the reference within the called method, this new address is discarded when the method completes.

Code in the called method can send messages to the object pointed at by the reference. These message sends *can* effect permanent changes in the object.

Call by Reference versus Call by Value

In `testCallByValueReferences`, shown following this paragraph, code creates a customer with an ID of 1. The test then calls the method `incrementId`. The first line of code in `incrementId` grabs the existing ID from the customer, adds 1 to this number, and sets the sum back into the customer. The second line in `incrementId` creates a new `Customer` object and assigns its address to the `customer` argument reference. You don't want to do this!¹¹ The assignment is discarded when `incrementId` completes. The test finally verifies that the customer ID is 2, not 22.

¹¹You might consider declaring all your arguments as `final`. Attempts to assign to the argument will result in compile failure. My ingrained habits prevent me from doing this consistently.

```

public void testCallByValueReferences() {
    Customer customer = new Customer(1);
    incrementId(customer);
    assertEquals(2, customer.getId());
}

private void incrementId(Customer customer) {
    customer.setId(customer.getId() + 1);
    customer = new Customer(22); // don't do this
}

class Customer {
    private int id;
    Customer(int id) { this.id = id; }
    void setId(int id) { this.id = id; }
    public int getId() { return id; }
}

```

Java Periphery

Java gives you a number of mechanisms to help your application interact with the local operating environment.

Properties

A small hash table of system properties is always available from any point in your application. This table contains useful information, including things about your Java environment and your operating system. You can obtain the entire set of properties using the `java.lang.System` method `getProperties`:

```
Properties existingProperties = System.getProperties();
```

The `java.util.Properties` class extends the old `java.util.Hashtable` class. It adds a number of management methods and utility methods. The management methods help you do things such as load a set of properties from a file. The utility methods simplify access to the hash table.

The API documentation for the `java.lang.System` class method `getProperties` shows the list of predefined properties. Some of the more useful properties include:

<code>java.class.path</code>	value of the Java classpath ¹²
<code>java.io.tmpdir</code>	the default temp file path

Java Periphery

¹²You can set this property to a new value, but it has no effect on the current Java execution. The Java VM does not reread this value. If you think you need to change the classpath, you may need to build a custom class loader instead.

file.separator	the file separator ('/' under Unix; '\' under Windows)
path.separator	the separator between entries in a path (':' under Unix; ';' under Windows)
line.separator	the line separator ('\n' under Unix; '\r\n' under Windows)
user.home	the current user's home directory
user.dir	the current working directory

Using these properties, you can code your application to execute on different operating systems without change.

You retrieve the value of a single system property using the `System` class method `getProperty`:

```
assertEquals("1.5.0-beta3", System.getProperty("java.version"));
```

System properties are global. You can access them from anywhere in your code.

You can add your own global properties to the list of system properties. Generally, a database is a better place for this sort of information. Your main reason to use properties should be to dynamically alter small pieces of information from one application execution to the next. For example, your deployed application may need to execute in “kiosk” mode most of the time, looping and occupying the full screen without borders or a title bar. During development, you may want to be able to switch rapidly back and forth between execution modes.

You can accomplish this dynamism in a number of ways. You could put a mode setting flag into a database, but having to modify data in a database can slow you down. You could pass the information as command line arguments and manage it in your `main` method. But you might not need the information until later in your application; what do you do with it until then?

You can add your property to the system properties programmatically:

```
System.setProperty("kioskMode", "loopFullscreen");
```

Or, as you saw with logging in Lesson 8, you can set the property as a VM argument:

```
java -DkioskMode=loopFullscreen sis.Sis
```

If a property is not set, `getProperty` returns `null`. You can provide a default value to `getProperty` to be returned if the property is not set. A language test shows how this works:

```

public void testGetPropertyWithDefault() {
    Properties existingProperties = System.getProperties();
    try {
        System.setProperties(new Properties()); // removes all properties
        assertEquals(
            "default", System.getProperty("noSuchProperty", "default"));
    }
    finally {
        System.setProperties(existingProperties);
    }
}

```

Property Files

One technique you will encounter in many shops and many applications is the use of property files. A property file is similar to a resource file; it is a collection of key-value pairs in the form `key=value`. Developers often use property files as a quick-and-dirty catchall for configurable values. Things such as class names of drivers, color schemes, and boolean flags end up in property files.

One driving force is the need to change certain values as software moves from a production environment into a test environment, an integration environment, or a production environment. For example, your code might need to connect to a web server. The web server name you use for development will differ from the web server name for production.

I recommend that you minimize your use of property files or at least minimize the entries that go in them. Property file entries are disjointed from your application; they represent a bunch of global variables arbitrarily clustered together. You should prefer instead to carefully associate configurable values with the classes they impact. The better way to do this is through the use of a database.



Keep any property files small and simple or, better yet, eliminate them. Resist adding new entries.

If you must use property files, the `Properties` class contains methods for loading from an input stream that is either in key-value text format or in XML format. Do not try to dynamically update this file from your executing application. Instead, consider the Preferences API, discussed in the next section.

Java Periphery

Preferences

For critical information that you must retain from application execution to execution, you should use a database or other trustworthy persistence mechanism. However, you often want the ability to quickly and simply store

various bits of noncritical information. Usually this is in the area of user trivialities: preferred colors, window positioning, recently visited web sites, and so on. Were any of this information lost, it would be a nuisance to a user but would not prevent him or her from accurately completing work.

The Java Preferences API supplies a simple local persistence scheme for this data. PersistentFrameTest (shown after the next paragraph) demonstrates. The first test, `testCreate`, shows that the first time a frame is created, it uses default window positions. The test `testMove` moves a frame, then closes it. The test shows that subsequent frames use the last position of any PersistentFrame.

In order for PersistentFrameTest to work more than once, the `setUp` method sends the message `clearPreferences` to the frame. This method ensures that the `PersistentFrame` object uses default window placement settings.

```
package sis.ui;

import junit.framework.*;
import java.awt.*;
import java.util.prefs.BackingStoreException;

import static sis.ui.PersistentFrame.*;

public class PersistentFrameTest extends TestCase {
    private PersistentFrame frame;

    protected void setUp() throws BackingStoreException {
        frame = new PersistentFrame();
        frame.clearPreferences();
        frame.initialize();
        frame.setVisible(true);
    }

    protected void tearDown() {
        frame.dispose();
    }

    public void testCreate() {
        assertEquals(
            new Rectangle(
                DEFAULT_X, DEFAULT_Y, DEFAULT_WIDTH, DEFAULT_HEIGHT),
            frame.getBounds());
    }

    public void testMove() {
        int x = 600;
        int y = 650;
        int width = 150;
        int height = 160;
        frame.setBounds(x, y, width, height);
    }
}
```

Java Periphery

```

        frame.dispose();

        PersistentFrame frame2 = new PersistentFrame();
        frame2.initialize();
        frame2.setVisible(true);
        assertEquals(
            new Rectangle(x, y, width, height), frame2.getBounds());
    }
}

```

The implementation in PersistentFrame:

```

package sis.ui;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.prefs.*;

public class PersistentFrame extends JFrame {
    static final int DEFAULT_X = 100;
    static final int DEFAULT_Y = 101;
    static final int DEFAULT_WIDTH = 300;
    static final int DEFAULT_HEIGHT = 400;

    private static final String X = "x";
    private static final String Y = "y";
    private static final String WIDTH = "width";
    private static final String HEIGHT = "height";

    private Preferences preferences =
        Preferences.userNodeForPackage(this.getClass());           // 1

    public void initialize() {
        int x = preferences.getInt(X, DEFAULT_X);                // 2
        int y = preferences.getInt(Y, DEFAULT_Y);
        int width = preferences.getInt(WIDTH, DEFAULT_WIDTH);
        int height = preferences.getInt(HEIGHT, DEFAULT_HEIGHT);

        setBounds(x, y, width, height);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        addWindowListener(new WindowAdapter() {
            public void windowClosed(WindowEvent e) {
                saveWindowPosition();
            }
        });
    }

    private void saveWindowPosition() {
        Rectangle bounds = getBounds();                           // 3

        preferences.putInt(X, bounds.x);
        preferences.putInt(Y, bounds.y);
    }
}

```

Java Periphery

```

preferences.putInt(WIDTH, bounds.width);
preferences.putInt(HEIGHT, bounds.height);
try {
    preferences.flush();                                // 4
}
catch (BackingStoreException e) {
    // not crucial; log message
}
}

// for testing
void clearPreferences() throws BackingStoreException {
    preferences.clear();
    preferences.flush();
}
}

```

You create a Preference object by passing a class name to one of two Preferences factory methods, `userNodeForPackage` or `systemNodeForPackage`, corresponding to the user and system trees of preference data. You might use the system preferences tree for application-specific information, such as the location of a properties file. You might use the user preferences tree for user-specific information, such as color choices.

Java stores preferences in trees that are analogous to file system directory tree structures. Depending on the platform (and specific implementation), the actual storage location of preferences could be an operating system registry, the file system, a directory server, or a database.

The `PersistentFrame` class uses the user preferences tree (see line 1). The frame initialization code attempts to read the preferences for the frame's bounds¹³ (starting at line 2). Each call to `getInt` can contain a default value to use if no corresponding preference data exists.

When the frame gets closed (by client code that invokes the `dispose` method or by other means), code in `saveWindowPosition` stores the current bounds of the frame using the Preferences API. You must `flush` the preferences object (line 4) to force a write to the persistent store. In addition to `.putInt` (line 3) the Preferences class supplies other methods to support storing various types of data: `String`, `boolean`, `byte[]`, `double`, `float`, and `long`.

The Preferences API provides methods for writing data to and reading data from a file in XML format.

¹³A `Rectangle` object specifies the bounds of a window. The (x, y) coordinate represent its origin, or upper-left corner. The height and width represent the extent of the window in pixels.

System Environment

On rare occasions, you'll need to know information about your operating environment that the predefined system properties does not provide. Unix and Windows operating systems have environment variables, yet again a key-value pair scheme. From a command-line prompt, you view the environment variables using the command `set`.¹⁴ From within Java, the `System` class provides two `getEnv` methods to help you retrieve these environment variables and their values.

```
Map<String, String> env = System.getenv();
for (Map.Entry entry: env.entrySet())
    System.out.printf("%s->%s%n", entry.getKey(), entry.getValue());

System.out.println(System.getenv("SystemRoot")); // Unix: try "SHELL"
```

Avoid depending on system environment variables. It will make it more difficult to port your application to another platform. Almost no environment variables are common between Windows and Unix. If you feel compelled to create your own system-wide environment variables, try to use the Java properties mechanism instead.

Executing Other Applications

You may on rare occasion need to kick off another operating system process. For example, you may want your application to bring up the system calculator application (`calc.exe` under Windows). Java will allow you to execute separate processes, but doing so is something you should avoid. It will usually tie you to a single operating system. In most cases you will then need to maintain conditional logic if you must handle multiple platforms.

There are two ways to create and execute separate processes under Java. The older technique requires you to obtain the singleton¹⁵ instance of `java.lang.Runtime` using its class method `getRuntime`. You can then pass a command-line string along with optional arguments to the `exec` method. J2SE 5.0 introduced a `ProcessBuilder` class that replaces this technique. You create a `ProcessBuilder` using the command-line string, then initiate the process by sending `start` to the `ProcessBuilder` object.

Java Periphery

¹⁴This works under Windows and under many Unix shells.

¹⁵The `Runtime` class is designed so that there can only be a *single* instance of `Runtime` in an executing Java application, therefore `Runtime` is an example of the *singleton* design pattern.

You will not see a console window when executing the new process. It is possible, but slightly tricky, to capture output from or siphon input to the new process. Java redirects the three standard IO streams (`stdin`, `stdout`, and `stderr`) to Java streams so that you can work with them. Usually you'll only need to capture output; that's what we'll concentrate on here.

The trickiness to capturing output comes from the fact that the buffer size of the operating system may be small. You must promptly manage data coming from each output stream. If you do not, your application is likely to hang as it executes.

```
package sis.util;

import junit.framework.TestCase;
import java.util.*;

public class CommandTest extends TestCase {
    private static final String TEST_SINGLE_LINE = "testOneLine";
    private static final String TEST_MULTIPLE_LINES = "testManyLines";
    private static final String TEST_LOTS_OF_LINES = "testLotsLines";
    private static Map<String, String> output =
        new HashMap<String, String>();
    private static final String COMMAND =           // 1
        "java " +
        "-classpath \\" + System.getProperty("java.class.path") + "\\" +
        "sis.util.CommandTest %s";
    private Command command;

    static {                                     // 2
        output.put(TEST_SINGLE_LINE, "a short line of text");
        output.put(TEST_MULTIPLE_LINES, "line 1\\nline 2\\n");
        output.put(TEST_LOTS_OF_LINES, lotsOfLines());
    }

    static String lotsOfLines() {
        final int lots = 1024;
        StringBuilder lotsBuffer = new StringBuilder();
        for (int i = 0; i < lots; i++)
            lotsBuffer.append("" + i);
        return lotsBuffer.toString();
    }

    public static void main(String[] args) {           // 3
        String methodName = args[0];
        String text = output.get(methodName);
        // redirected output:
        System.out.println(text);
        System.err.println(text);
    }

    public void testSingleLine() throws Exception {
        verifyExecuteCommand(TEST_SINGLE_LINE);
    }
}
```

Java Periphery

```

public void testMultipleLines() throws Exception {
    verifyExecuteCommand(TEST_MULTIPLE_LINES);
}

public void testLotsOfLines() throws Exception {
    verifyExecuteCommand(TEST_LOTS_OF_LINES);
}

private void verifyExecuteCommand(String text) throws Exception {
    command = new Command(String.format(COMMAND, text));
    command.execute();
    assertEquals(output.get(text), command.getOutput());
}
}

```

CommandTest uses itself as the very command to test. Line 1 constructs a java command line to execute sis.util.CommandTest as a stand-alone application. It obtains the appropriate classpath by querying the system property `java.class.path`. The `main` method (line 3) supplies the code for the application.

The `main` method simply writes lines of output to `stderr` (`System.err`) and `stdout` (`System.out`). It uses a single argument, an arbitrary string that corresponds to a test method name, to determine what lines to write. The lines come from a map populated using a static initializer (line 2).

The `Command` class appears next. In line 1, you create a `ProcessBuilder` object using the command string. The `ProcessBuilder start` method kicks off the corresponding operating system process. Lines 4 and 6 show how you can extract the streams corresponding to `stderr` and `stdout`. Once you've obtained these streams, you can use them to capture output using a `BufferedReader` (see the `collectOutput` method at line 7). The trick to making all this work is to ensure that the process of collecting output executes in simultaneous threads, otherwise you run the risk of blocking or losing output. Thus, the code in `collectErrorOutput` (line 3) and `collectOutput` (line 5) creates and starts new `Thread` objects. After initiating the threads, code in `execute` waits until the process completes, using the `Process` method `waitFor` (line 2).

```

package sis.util;

import java.io.*;

public class Command {
    private String command;
    private Process process;
    private StringBuilder output = new StringBuilder();
    private StringBuilder errorOutput = new StringBuilder();

    public Command(String command) {
        this.command = command;
    }

    public void execute() throws Exception {

```

Java Periphery

```

process = new ProcessBuilder(command).start();           // 1
collectOutput();
collectErrorOutput();
process.waitFor();                                     // 2
}

private void collectErrorOutput() {                      // 3
    Runnable runnable = new Runnable() {
        public void run() {
            try {
                collectOutput(process.getErrorStream(),   // 4
                               errorOutput);
            } catch (IOException e) {
                errorOutput.append(e.getMessage());
            }
        }
    };
    new Thread(runnable).start();
}

private void collectOutput() {                          // 5
    Runnable runnable = new Runnable() {
        public void run() {
            try {
                collectOutput(process.getInputStream(), // 6
                               output);
            } catch (IOException e) {
                output.append(e.getMessage());
            }
        }
    };
    new Thread(runnable).start();
}

private void collectOutput(                         // 7
    InputStream inputStream, StringBuilder collector)
    throws IOException {
    BufferedReader reader = null;
    try {
        reader =
            new BufferedReader(new InputStreamReader(inputStream));
        String line;
        while ((line = reader.readLine()) != null)
            collector.append(line);
    }
    finally {
        reader.close();
    }
}

public String getOutput() throws IOException {
    return output.toString();
}

```

```

public String getErrorOutput() throws IOException {
    return output.toString();
}

public String toString() {
    return command;
}
}

```

Avoid use of command-line redirection (> and <) or piping (|)—these will be considered part of the command input string and will not be appropriately handled by the operating system.

The `ProcessBuilder` class allows you to pass new environment variables and values to the new process. You can also change the current working directory under which the process executes.

One thing that `ProcessBuilder` provides that the `Runtime.exec` technique does not is the ability to merge the `stdout` and `stderr` streams. Console applications often interweave these: You might get a prompt or other output message that comes from `stdout`, then an error message from `stderr`, then a `stdout` prompt, and so on. If `stdout` and `stderr` are separated into two streams, there is no way to determine the original chronological sequencing of the combined output. See the API documentation for details on how to control this feature.



Avoid use of things such as `getEnv` and `ProcessBuilder` that introduce dependencies on your operating environment.

What Else Is There?

Java contains many more advanced topics that most developers will rarely have a need for. There are also dozens upon dozens of additional APIs that you may need to consider. This section briefly mentions some of these topics and APIs.

Custom Class Loaders

When you access a class for the first time, the Java VM uses one of three default class loaders to load the class for use. The first class loader Java uses is the bootstrap class loader. The bootstrap class loader looks in the system JAR files `rt.jar` and `i18n.jar` to find Java classes from the core Java API library. If Java does not find the class, it then looks in the Java extension directory, `./lib/ext` under the JRE installation directory. If necessary, Java finally uses the directories on your classpath to load the class.

What Else Is There?

You can create additional custom class loaders. A custom *class loader* can load a class from an alternate location. It might load a class from a database, from a web site, from an FTP site, or from dynamically created byte codes.

For further information, refer to the article “Create a Custom Java 1.2-Style Classloader” by Ken McCrary at <http://www.javaworld.com/javaworld/jw-03-2000/jw-03-classload.html>. For more-up-to-date information, start with the API documentation for the class `java.lang.ClassLoader`.

Weak References

Java’s garbage collection scheme manages reclaiming memory for objects that you’re done with. But what if you want to write an application that monitors memory use? In order to monitor an object, you must hold a reference to it. As long as your monitor application retains such a reference, the garbage collector cannot reclaim the object in question.

You might also want to implement a caching scheme. A cache loads frequently accessed objects. But since the cache data structure must (by definition) refer to cached objects, the garbage collector can never consider them for reclamation. You must write complex additional code to manage removing objects from the cache from time to time. Otherwise your cache will continue to grow until you get an out-of-memory error.

To help with these and other problems, Java lets you use *weak references*. A weak reference to an object does not count for purposes of garbage collection. The garbage collector will reclaim any object with weak references but no strong references.

Java supplies three levels of weak references, in order from weakest to strongest:¹⁶ phantom, weak, and soft. A phantom reference can be used for special cleanup processing after object finalization. A weak reference results in the referred object being removed when garbage is collected. A soft reference results in the referred object being removed only when the garbage collector determines that memory is really needed.

The class `java.util.WeakHashMap` is a Map implementation with weak keys. Refer to its API documentation for further details.

Refer to the Java API documentation for the package `java.lang.ref` for more information on the various weak reference types.

What Else Is There?

The finalize Method

In Lesson 4, you learned a bit about garbage collection. Java automatically collects unused objects—ones that no other objects refer to. Once in a while, you may find the need to accomplish something when an object is garbage collected. To do so, you might consider defining a `finalize` method:

¹⁶[JavaGloss2004b].

```
protected void finalize() throws Throwable {
    try {
        // do some cleanup work
    }
    finally {
        super.finalize();
    }
}
```

The garbage collector calls the `finalize` method against any objects that it collects.

The problem is that you cannot guarantee that `finalize` ever gets called. Even when the Java VM shuts down—when your application exits—`finalize` might not get called. This means that you should never write code in `finalize` that must execute. Closing file or database connections in a `finalize` method is not a good idea—chances are that these resources will never be released.

If you think you need a `finalize` method, try to find a way to redesign your code. Or consider the use of phantom references as an alternative. If you do implement `finalize`, it is a good idea to ensure that you always call the superclass `finalize` method (as shown in the above bit of code).

Instrumentation

Using Java instrumentation capabilities, you can add informational byte codes to Java class files. This information can be used to aid in logging messages, profiling method execution times, or building code coverage tools. The goal is to allow you to insert such information in a manner that does not alter the functionality of the existing application.¹⁷

To use instrumentation, you implement the `ClassFileTransformer` interface. You register this implementation at the `java` command line using the `-javaagent` switch. As the class loader attempts to load a class, it calls the `transform` method in your `ClassFileTransformer` implementation. The goal of the `transform` method is to return a byte array representing the replacement class file.

Refer to the Java API documentation for the package `java.lang.instrument` for more information.

Management

The Management API allows you to monitor and manage the Java VM. It also allows for some management of the OS under which the VM is executing. The Management API allows you to do things such as compare perfor-

What Else Is
There?

¹⁷Java's instrumentation API is an example of a scheme known as *aspect-oriented programming*.

mance characteristics of the garbage collector, determine the number of processors available, monitor memory usage, monitor threads, or to monitor class loading from an external perspective.

Refer to the API documentation for the package `java.lang.management` for more information.

Networking

The `java.net` package supplies several classes to help you build network applications. This package provides support for low-level communications, based on the universal concept of a bidirectional communication mechanism known as a socket. Sockets allow you access to the TCP/IP protocol. Sockets are the underpinning of most host-to-host communication.

The `java.net` package also provides a number of higher-level APIs centered on web programming and URLs (Universal Resource Locators).

For more information, refer to the API documentation for the package `java.net`.

NIO

Java NIO (“New I/O”) is an advanced, high-performance facility for input-output operations. If you need to boost your performance for mass data transfers, you may want to consider NIO. While Sun has retrofitted many of the `java.io` streams to use NIO, you may want to directly interact with the NIO API for the fastest possible data transfers.

NIO is not stream based. You instead use channels, which are similar to streams in that they are sources and sinks for data. A buffer contains chunks of data that channels read from and write to. The main speed gain in NIO comes from the use of direct buffers. Normally, data is copied between Java arrays and VM buffers. Direct buffers are allocated directly within the VM and allow your code to directly access them. This avoids the need for expensive copy operations.¹⁸

NIO is nonblocking. Normally when you execute an I/O operation, calling code is blocked—it must wait until the operation completes. Using nonblocking I/O, your code can continue to execute even if a read or write operation is taking place. You can use nonblocking socket channels as the foundation for socket servers. This increases the scalability of the socket server and simplifies the management of incoming connections and requests. Blocking I/O

What Else Is
There?

¹⁸[Travis2002], pp. 2–3.

requires the judicious use of multithreading, while NIO allows you to manage all incoming requests in a single thread.

Conceptually, NIO is more complex than the standard `java.io` library. The tradeoff is worth it if performance is paramount. In the case of socket servers, NIO is definitely the way to go. For most other needs, the standard I/O library will suffice.

For more information, refer to the Sun NIO document at <http://java.sun.com/j2se/1.4.2/docs/guide/nio/index.html>.

JNI

The Java Native Interface (JNI) is your hook to the outside world. You might need to interact with a hardware API written in C, or perhaps you might need to call operating system routines that go beyond the capabilities of Java. Using JNI, you can call libraries (DLLs in Windows and `sos` in Unix) written in other languages including C++ and C.

For more information, the Sun tutorial is a good place to start (but it may be somewhat out of date): <http://java.sun.com/docs/books/tutorial/native1.1/stepbystep/index.html>.

As with any Java technology that allows you to interact with external resources, minimize your use of JNI in order to maximize your potential for cross-platform deployment.

RMI

Remote Method Invocation (RMI) is an API that allows you to call methods on Java objects that execute in the context of another Java VM. The other VM can be on the same machine or on a remote machine. RMI is the basis for the distributed component-based computing used in EJBs (Enterprise Java Beans).

RMI uses the proxy design pattern. A client object can interact with a server method located on another machine by working through its public interface. In reality, the client interacts with a client-side stub. The stub takes a Java message and translates it into serialized objects that can be sent across the wire. The stub interacts with a server-side skeleton, the job of which is to take those object streams and translate them into a method call against the actual server class.

You generate the stub and skeleton Java source using the RMI compiler `rmic`, located in your Java `bin` directory.

Using RMI keeps your clients from having to build remote communication code. As far as your client code knows, it is interacting with just another Java

What Else Is
There?

object in the same virtual machine. Network limitations will inherently slow your application, however, so you must still design your application with the high overhead of RMI in mind. The primary design goal for distributed applications is to minimize the amount of distributed processing—don't distribute your objects unless you must!¹⁹

Beans

You use JavaBeans, not to be confused with Enterprise Java Beans (EJBs), for building pluggable GUI components. You might build a new Java stoplight GUI control that you want to sell to Java GUI developers. The stoplight Java-Bean is a Java class like any other. But in order for developers to load and use your stoplight control in tools such as JBuilder, it must adhere to JavaBeans specifications.

To be a visual JavaBean, a class must inherit from `java.awt.Component` and must be serializable. It must expose its information using accessor methods that follow the standard Java naming convention (e.g. `getName` to return the value of the field `name`). The ability of tools to gather bean information based on knowledge of these conventions is known as introspection. A bean has a related `BeanInfo` class that provides relevant bean metadata. Beans communicate with other beans using an event mechanism.

For more information, the best place to start is the Sun tutorial at <http://java.sun.com/docs/books/tutorial/javabeans/>.

Security

Security in Java is easily a topic for a whole book. The J2SE security API, located in various packages starting with the name `java.security`, includes classes for things that include certificate management, key store management, policy files, encryption/decryption algorithms, and access control lists.

The Java security model is based on the concept of a *sandbox*, a customizable virtual place. In the sandbox, Java programs can execute without adverse impact to the underlying system or its users.

The core J2SE security API grows with each new release of Java. For full information on what's available, refer to the Java API and release documentation.

What Else Is
There?

J2EE

The Java 2 Platform Extended Edition (J2EE) is a collection of various enterprise-level APIs. J2EE can allow you to create scalable component-based applications.

¹⁹[Fowler 2003a], p. 89.

Often when people refer to J2EE, they specifically mean Enterprise Java Beans (EJBs). EJBs are components. The goal of EJBs is to eliminate the need for an application developer to re-create deployment, security, transaction, and persistence services. An EJB application server such as BEA's WebSphere, IBM's WebLogic, or JBoss is a container that provides these services for EJBs.

J2EE is heavily weighted toward XML as the common data language. Four APIs center on XML: the Java API for XML Processing (JAXP), the Java API for XML-based RPC (remote procedure calls) (JAX-RPC), SOAP with Attachments API for Java (SAAJ), and the Java API for XML Registries (JAXR).

Another set of J2EE APIs centers on Web development. The Java Servlet API is the basis for web applications. JavaServer Pages (JSPs) simplify the presentation layer for web applications. The JSP Standard Tag Library (JSTL) provides common custom tags to simplify writing JSPs. JavaServer Faces gives you a framework for simplifying the development of the model and application facets of a web application.

Various other J2EE APIs support transactions, resource connections, and security. The Java Message Service (JMS) provides an interface for asynchronous message needs.

Scads of books exist on J2EE. The J2EE tutorial available at <http://java.sun.com/j2ee/download.html> is as good a place as any to start. Before you leap into the extremely complex world of J2EE, ensure that you have justified your decision from the standpoints of functionality need, cost of development, complexity requirements, performance need, and scalability needs. Many shops invested heavily in J2EE, only to find that they didn't need most of the things it provides. Often you will find that your own custom implementation provides all you need, is vastly simpler, and is a far more flexible solution.

More?

Dozens more Java APIs exist. They support various things such as image manipulation, speech, applets, cryptography, shared data, telephony, accessibility, and automatic web installation.

Java is an entrenched technology that will not go away anytime soon. The evidence is the fact that for virtually every computing need, a Java API for it exists. The best place to start is Sun's Java site, <http://java.sun.com>. Beyond that, the world is a haystack with many needles. Use Google to find them!



Go fish!

What Else Is
There?

This page intentionally left blank

Appendix A

An Agile Java Glossary

This appendix lists key terms. Most terms that appear in the text of *Agile Java* in *italicized form* appear here.

Each definition is brief but should provide you with a good idea what a given term means. Refer to the index in order to locate occurrences of the term within *Agile Java*. The context should provide you with a more robust understanding of a specific term.

abstract representing concepts as opposed to concrete, specific details

acceptance test a test that demonstrates functional behavior; also known as a functional test or customer test

access modifier a keyword that identifies the level of exposure of a Java element to other code

action method a method that effects an action, usually altering the state of an object

activation a UML construct that represents the lifetime of a method in a sequence diagram

actual value in an `assertEquals` statement, the expression or value that you are testing. The actual value immediately follows the expected value.

agile a term for a body of lightweight processes; an agile process centers around the idea that requirements will change as a project progresses

annotation a construct that allows you to embed descriptive information for Java elements in source code. An annotation is verified against an interface specification by the compiler.

anonymous inner class an unnamed class that is dynamically defined within a method body

API application programming interface

application a computer-based system that provides value to humans or other systems

**An Agile Java
Glossary**

application programming interface the published specification for other classes with which your code can interact

argument an element that is passed to a method. Informal use of the term can indicate either one of the list of actual arguments passed in a method call or the formal parameters listed in the method's signature

argument list the list of values or references passed to a method

array initializer an initialization construct that allows provision of a set of initial values for elements in an array

assignment the act of storing the result of an expression in a field or variable

assignment operator the equals sign (=); used as part of an assignment

atomic an operation that executes as a single indivisible unit

attribute a general object-oriented term for a characteristic of an object. You define attributes on a class.

autoboxing automatically enclosing a primitive with its corresponding wrapper type

autounboxing automatically extracting a primitive from its corresponding wrapper type

bind to supply a type for a parameterized class

boolean a type that represents two possible values: true and false

bound having supplied a type for a parameterized class

block (n.) a section of code enclosed between braces; (v.) to wait until some other operation has completed

border a buffer area between the edges of a container and any of its components

byte codes the programming operations and other relevant information contained in a class file

callback the technique of passing a function as a parameter so that the receiver can message to it

camel case the mixed-case naming scheme preferred for Java identifiers

cast to tell the compiler that you want to convert the type of a reference; to narrow a numeric value to a smaller type

catch to trap a thrown exception

checked exception an exception that your code must explicitly acknowledge

checked wrapper a collection wrapper that ensures that objects stored in the collection are of the appropriate type

class a template for creating objects; a class defines behavior and attributes

class constant a reference to a fixed value that can be accessed irrespective of creating object instances

class diagram a UML model that shows class detail and the static relationship between classes

class file a file containing code that has been compiled from source code

class library a collection of useful classes provided by a vendor or another developer as an API

class literal a value that represents the sole instance of a class

class loader code that dynamically reads a stream of bytes representing a Java compilation unit from some source location and converts it to a usable class

class variable a variable defined on a class that can be accessed without using an instance of the class

classpath a list of directories or JAR files that Java uses for compilation and execution

client a class, or code in general, that interacts with a specific target class. The target class may be referred to as a server class.

clone to create a copy of an object

close to design a class so that it needs no further modification despite future changes

coding the act of writing programs

collective code ownership a team philosophy that allows anyone to modify any portion of the system

collision when two elements hash to the same location in a hash table

compiler an application that transforms source code into a binary form that a platform can understand and execute

command object an object that captures information in order to be able to effect specific behavior at a later time

comment a free-form annotation of code that is ignored by the compiler

complex annotation an annotation that includes another annotation

compound assignment a shortcut form of assignment that presumes that the variable being assigned to is also the target of the first operation in the right-hand expression

An Agile Java
Glossary

composition an association between classes whereby objects of one class are composed of objects of another class

concatenate append

connection pool a group of shared database connections

constructor a block of code that allows you to provide statements to initialize an object. Invoking a constructor using the `new` operator results in the creation of an object.

constructor chaining calling one constructor from another

container a component that can contain other components

controller a class responsible for managing input from the end user

cooperative multitasking a multithreading scheme in which the individual threads are responsible for yielding time to allow other threads to execute

couple to increase dependencies between classes

covariance the ability for a subclass to vary the return type of an overridden method

critical section see “mutual exclusion”

customer test see :acceptance test

daemon thread a secondary thread; an application can terminate regardless of whether or not daemon threads are active

declaration the point at which something is defined

decouple to minimize dependencies between classes

decrement to subtract (usually one) from a value

decrement operator `--`. A unary operator that subtracts 1 from the operand

default package the unnamed package; classes that are defined without a package statement end up in the default package

dependency a relationship between two classes (or any two entities) where one requires the existence of the other

dependency inversion designing class associations so that classes are dependent upon abstractions instead of concrete implementations

deprecated Java code intended for eventual removal from a class library

dereference to navigate to a memory location based on the memory address stored in a reference

derived class see “subclass”

deserialize to “reconstitute” an object from a sequence of bytes

design by contract a design technique that requires programmers to create a contract for how their software is expected to behave. See precondition, post-condition, and invariant.

driver with respect to JDBC, an API that meets Sun's specification for interfacing with a database

dynamic proxy class a class that allows you to implement one or more interfaces at runtime

element a Java construct that an annotation can possibly annotate: a class, enum, interface, method, field, parameter, constructor, or local variable

encapsulation hiding implementation specifics from clients; a core object-oriented concept

erasure the scheme chosen by Sun to implement parameterized types; it refers to the fact that parameterized type information is erased to its upper bounds

escape sequence a sequence of characters within a String or character literal that maps to a single character. An escape sequence begins with the character \\. Escape sequences allow you to represent characters that you may not be able to type directly.

exception an object that encapsulates error information

expected value in an assertEquals statement, the value you expect to receive when comparing to an actual result. The expected value immediately precedes the actual value.

factory method a method responsible for constructing and returning objects

field the Java implementation of an attribute. A field minimally specifies a type and name for a characteristic of a class

field initializer a code expression used to provide an initial value for a field or local variable

formal parameter a named argument listed in a method or constructor signature

format elements substitution parameters in a resource defined in a resource bundle

format specifier a placeholder for substitution values within a format string; defines the transformation from value to output string

frame a top-level window

fully qualified indicates that a class name contains its complete package information

An Agile Java
Glossary

garbage collection the VM process of reclaiming memory that is no longer needed

garbage collector the code that is responsible for garbage collection

generic method a method that can be parameterized with one or more type parameters

generics parameterized types

global available to any interested client

guard clause a code construct that defines an early return from a method in the event of an exceptional condition

hacking informal: creating software with abandon. Originally, hacking was a positive term; modern use of the term (including its use in *Agile Java*) is usually pejorative

hash code an integer generated by a hashing algorithm that is used as the key for locating elements in a hash table

hash table a fixed-size collection that determines the location of its elements using a hash code

heavyweight refers to a process that emphasizes written, up-front documentation and discourages deviation from an initial plan

i18n Internationalization

identifier the name given to a Java element, such as a field, method, or class

immutable unchangeable. A class is immutable if its attributes cannot be altered. An immutable object is also known as a value object

implement to provide code details. When you implement an interface, you supply concrete definitions for the methods it declares

increment to add (usually 1) to a value

increment operator `++` A unary operator that adds 1 to the operand

inherit to extend from a base class

inheritance a relationship between two classes where one class (known as the subclass or derived class) specializes the behavior of another class (known as the superclass or base class)

initial value the value that a field or local variable has at the time of declaration

inline to embed previously externalized code into a method body; i.e., to eliminate the need for a method call by moving the target method's code into the calling method

instance variable see field

instantiate to create an object

interface general: the point of communication between two things, such as two systems or two classes; Java: a construct that allows you to define method specifications, but no implementation details

interface adapter class a class that transforms an externally defined interface into one that is more useful in the context of the current system

internationalize to code a system such that it can support deployment to different localities

interpreter an application that reads and executes compiled code; see also virtual machine

invariant an assertion that must always hold true while executing code; see also design by contract

invert to turn upside-down; used in the context of dependency inversion

jagged array an array of arrays where each subarray can vary in size

JAR Java ARchive

Java archive a class container; implemented using the ZIP file format

layout the arrangement of components in a user interface

layout manager a class that can aid in arranging components in a user interface

lazy initialization a programming technique whereby you do not provide an initial value for a field until it is used for the first time

lightweight refers to a process that emphasizes adaptation over adherence to plan

linked list a serial data structure that adds new elements by dynamically allocating memory. Elements of a linked list must refer to other elements by storing one or more references to them.

local variable a variable defined within the body of a method

locale a region that is defined by perhaps geography, culture, or politics

localize to prepare an internationalized application for delivery to a locale

log to record

logical operator an operator that returns a boolean value

lower bound a constraint that requires a wildcard to be a supertype of the specified class

marker annotation an annotation that provides no arguments

An Agile Java
Glossary

marker interface an interface with no method declarations; allows programmers to signal intent for a class

member a constructor, field, or method defined on the instance side of a class

member-value pair a key and corresponding value supplied in an annotation

memory equivalence when two object references are equal—i.e., they point to the same object in memory

metadata data about data

method a unit of code that helps define the behavior of a class. A return type, name, and argument list uniquely define a method. A method consists of a number of statements

methodology a process for building software

mnemonic a single-letter shortcut for activating a button

mock object an object used for creating context to aid in verifying assertions against target code. Loosely used, a mock is any object that substitutes for production code to aid in testing

model a class that contains application or business logic

modulus an operator that returns the remainder of a division operation

multiline comment a comment that can span multiple source lines; you start a multiline comment with /* and end it with */

multithreaded having the ability to execute certain code sections at the same time as other sections

mutual exclusion a form of synchronization in which a section of code has exclusive attention from the Java VM while it executes

naked type variable a use of a type parameter symbol within a type declaration

namespace a namespace implies the ability to define unique names, such that entities with the same base name do not clash with one another. In Java, this is accomplished through the use of package names

nested embedded within

nested class a class defined within another class

nondeterministic having the potential to return different results each time when given the same inputs and initial state

numeric literal a code instance of a number, such as an integer or a floating-point number

object an entity that has identity, type, and state; objects are created from classes

object-oriented based on the concepts of classes and objects

operator a special token or keyword recognized by the compiler. An operator takes action against one or more values or expressions

overload to supply multiple definitions for a method or constructor. You vary the definitions by their argument list (or possibly their return value; see covariance)

overloaded operator an operator that has different meanings depending on the context in which it is used

override to provide a replacement definition for a method already defined in a superclass

package an arbitrary collection of classes, defined either for deployment or organizational purposes

package import a form of the import statement that indicates that any of the classes in the specified package might be used in the class

package structure the current state of package organization

pair programming a development technique in which two developers actively program at a single workstation

parameter informally, another name for an argument

parameterized type a class to which you can bind one or more classes; doing so supplies additional constraints on the types of object that the class can interact with

persistent existing from application execution to execution

platform an underlying system on which applications can be run

polymorphism the ability of objects to supply variant behaviors, in response to a message, based on their type. Seen from the opposite direction, the ability of clients to send a message to an object without an awareness or concern for which object actually receives the message

postcondition an assertion that must hold true after executing code; see also design by contract

postfix operator a unary operator that appears after the target it operates on

precondition an assertion that must hold true prior to executing code; see also design by contract

An Agile Java
Glossary

preemptive multitasking a multithreading scheme in which the operating system (or virtual machine) is responsible for interrupting and scheduling threads

prefix operator a unary operator that appears before the target it operates on

primitive type non-object types in Java. Numeric types and the boolean type are primitive types

process framework a foundation for creating and customizing a process tailored to your needs

process instance a specific customization of a process framework

program the body of code that comprises an application; often used interchangeably with the term “application”

programmer test see “unit test”

programming see “coding”

programming by intention a programming technique whereby you first specify your intent for a solution (i.e., the general steps involved) before implementing it (i.e., providing the detailed code)

project the related things that make up an effort

proxy a stand-in. A proxy object is an object that might filter or control access to the object it stands in for. To a client, a proxy is indistinguishable from the real thing

public interface the methods of a class that are exposed to any other client class

query method a method that returns information to a client but does not alter the state of the object

queue a serial data structure that returns the eldest object when a client requests an object from it. Also known as a first-in-first-out (FIFO) data structure

random access file a logical interpretation of a file that allows you to quickly seek to specific positions in the file and read from or write to that position

raw type an unbound parameterized type

realizes implements

receiver the object to which a message is sent, directly or indirectly, as the result of a method call

recursive calling oneself; sending a message to the same method that is currently executing

refactor to transform code; by definition, refactoring should not change the behavior of code

refactoring (n.) a code transformation; (v.) the act of applying transforms to code

reference a memory address; a variable that points to an object

reflective of or dealing with reflection

reflection the capability in Java that allows you to dynamically derive information about types and their definitions at runtime

regression test a test that ensures that the existing application still works after changes have been applied

regular expression a set of syntactical elements and symbols that is used to match text

resource bundle a collection of resources such as message strings; allows an application to dynamically load said resources by name. A resource bundle is usually an external file.

rethrow to catch and subsequently throw the same or another exception object

return type indicates the class of object that a method must provide to the caller

runtime refers to the scope of execution of a program; i.e., when it is running

sandbox a customizable virtual Java place that defines boundaries for security purposes

scale the number of digits to the right of the decimal point in a decimal number

scope the lifetime of an object or variable; the sphere of influence it has

seed a number used as the basis for determining a pseudorandom number sequence

semantic equality when two objects are considered the same based on arbitrary criteria as supplied by a programmer. In Java, you use the `equals` method to define semantic equality

serializable a class whose objects can be serialized; arbitrarily designated by a programmer

serialize to convert an object to a sequence of bytes

An Agile Java
Glossary

server a class or group of classes that provides a service to clients

singleton a class that is coded to disallow creation of more than one instance of itself

signature the information that uniquely identifies a method: its return type, name, and list of argument types

simple design an approach to design that values testing, elimination of duplication, and code expressiveness. Simple design rules result in positive emergent behavior in code.

single-line comment a comment that begins with //; an end-of-line character terminates the comment

software development kit a bundle of applications and utilities designed to help developers build and execute custom applications

source file a file containing code as typed by a programmer

spurious wakeup an unexpected activation of an idling thread

stack trace the execution summary that an exception generates. A stack trace shows, in reverse order, the chain of message sends leading up to a problem

stack walkback see “stack trace”

state a snapshot of what an object looks like at a current point in time, represented by the values in its attributes

statement a line of Java code, terminated by a semicolon

static import an import form that allows you to use class methods or variables defined in a different class as if they were defined locally

static initialization block a block of code that executes when a class is first loaded by the VM

static method a method you can invoke without first instantiating the class on which it is defined

static scope having existence for the lifetime of a class

story an informally stated requirement; a promise of further conversation

strategy an algorithm; a variant way of accomplishing a task

stream a sequence of data that you can write to or read from

string literal a code instance of a String object, represented by text enclosed in double quotation marks

strongly typed a characteristic of a programming language whereby variables and constants must be associated with a specific data type

subclass the specialized class in an inheritance relationship

- subscript** an index representing a position within an array
- suite** a collection of unit tests
- synchronization** the coordination of simultaneously executing threads to avoid data contention
- target** the subject of desire; the object being operated on; in Ant, a process you want to execute
- TDTT** test-driven truth table
- temp variable** see “local variable”
- temporary variable** see “local variable”
- ternary operator** an operator that allows you to represent an if-else statement as an expression
- test-driven** designed by virtue of programming specification tests prior to coding the corresponding implementation
- test-driven truth table (TDTT)** a series of assertions to demonstrate the results for all possible combinations of bitwise and logical operators
- token** an individual text component delineated by specially designated text characters
- tolerance** the acceptable margin of error when comparing two floating-point quantities
- thread** a Java unit of execution. A Java process may contain multiple threads, each of which can be executing simultaneously with the others
- thread pool** a group of reusable threads
- throw** to generate an exception; a transfer control mechanism
- trace statements** code inserted in your application to allow you to monitor its behavior as it executes
- two's complement** a scheme used for internal representation of negative integers. To obtain the two's complement representation of a number, take its positive binary representation, invert all binary digits, and add 1
- type parameter list** in a parameterized type, the list of arguments that you must bind to
- uncaught exception handler** a hook that allows you to trap executions thrown from a thread
- upper camel case** a Java identifier naming scheme; camel case in which the first letter of the identifier is upper case
- unary operator** an operator that has a single target

An Agile Java
Glossary

unchecked exception an exception that does not require code to explicitly acknowledge it

unit test a body of code that verifies the behavior of a target class (i.e., a unit); also known as a programmer test

upper bound the constraint attached to a type parameter

user interface the means with which a human interacts with an application

user thread the main thread in an executing application; the existence of an active user thread sustains application execution

varargs a variable number of arguments

view a class responsible for displaying things to an end user

VM virtual machine

virtual machine an application that executes and manages custom programs. A virtual machine interacts with an underlying operating system as necessary; also referred to as an interpreter

walkback see “stack trace”

weak reference a reference that does not count for purposes of garbage collection

whitespace space, tab, new line, form feed, and carriage return characters (collectively)

wildcard a substitution character (?) in a type argument (for a parameterized type) that represents any possible type

wrapper an object that contains another object or value. In Java, wrapper objects are used to contain primitives so that they can be used as objects

Appendix B

Java Operator Precedence Rules

The table in this appendix lists the complete set of operators.¹ Operators grouped on the same line have the same precedence. The table orders precedence groups from highest to lowest. In contrast with the binary operators + and -, the unary operators + and - are used in conjunction with a numeric literal or variable to express whether it is positive or negative.

postfix	[] . () ++ -
unary	++ - + -
creation, cast	new (class)reference
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > >= <= instanceof
equality	== !=
and	&
exclusive or	^
inclusive or	
conditional and	&&
conditional or	
ternary (conditional)	?:
assignment	= += -= *= /= %= >>= <<= >>>= &= ^= =

¹[Arnold 2000].

This page intentionally left blank

Appendix C

Getting Started with IDEA

IDEA

This appendix demonstrates how to build and execute the “Hello World” application using IntelliJ IDEA. Before you attempt to use a sophisticated IDE such as IDEA, you should first learn how to do command-line compilation, execution, and testing of Java code. Learning these fundamentals will ensure that you can use Java on any platform you encounter.

IDEA

IDEA is a Java IDE built by a Czech company called JetBrains. IDEA has been one of the driving forces behind Java-specific intelligent features in an IDE. Many of the valuable features sported by other Java IDEs appeared first in IDEA. IDEA is also an open tool, meaning that third-party vendors and individual developers can enhance its capabilities by writing plug-ins for it.

I’ve used IDEA for several years, alternating between it and Eclipse as my primary IDEs (the choice my client often makes). One thing I’ve noted about IDEA is that it promotes the notion of least surprise: Generally, if you want to do something, IDEA does it and in a way you might expect or hope for.

IDEA also happened to support all of the new features of J2SE 5.0 well before Eclipse, in time for me to write this appendix before my deadline.

You can obtain the latest version of IDEA from the JetBrains web site at <http://www.jetbrains.com>. The version I used at the time of writing this appendix was IntelliJ IDEA 4.5.1. Follow the instructions at the site for downloading and installing IDEA. You should have already installed Java before bothering to install IDEA.

The installation program walks you through a number of questions regarding how you want IDEA configured. In general, you can go along with the defaults and just hit the Next button.

The following instructions are intended to be a brief introduction to getting started with IDEA, not a comprehensive user’s guide. If you need more infor-

mation, refer to the help supplied with IDEA or contact JetBrains support. Expect that the instructions shown in this appendix should not change dramatically in the near future, barring any significant changes from JetBrains.

The Hello Project

The Hello Project

Start IDEA once you have installed it. IDEA will show you the dialog shown in Figure C.1.

The “Hello World” project will be throwaway. Change the text `untitled` to `hello` in the **Name** field. As you type the project name, note that the directory specified in the **Project file location** field changes automatically. You may want to make a note of the location so that you are aware of where it is in your file system.¹ Click the **Next** button. You will see the dialog shown in Figure C.2.

If you see nothing listed in the **Project JDK** field or if you do not see a version of Java that is 1.5 (5.0) or later, click the **Configure** button. Use the **Configure JDK** to add (+) new JDKs as necessary. You will need to know the installation location of J2SE 5.0.

Ensure you have selected a J2SE 5.0 JDK and click **Next**. Continue to click **Next**, accepting the defaults for each panel. Click the **Finish** button when IDEA first makes it available to you.

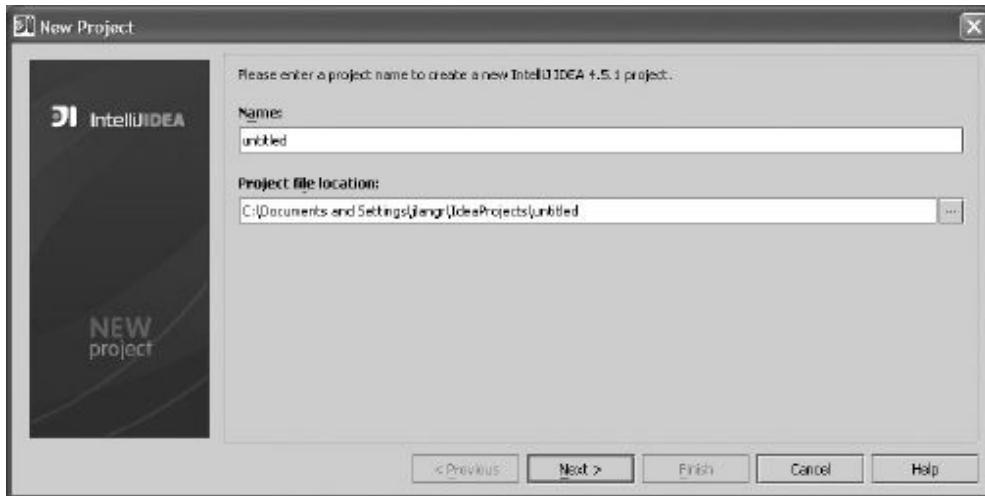
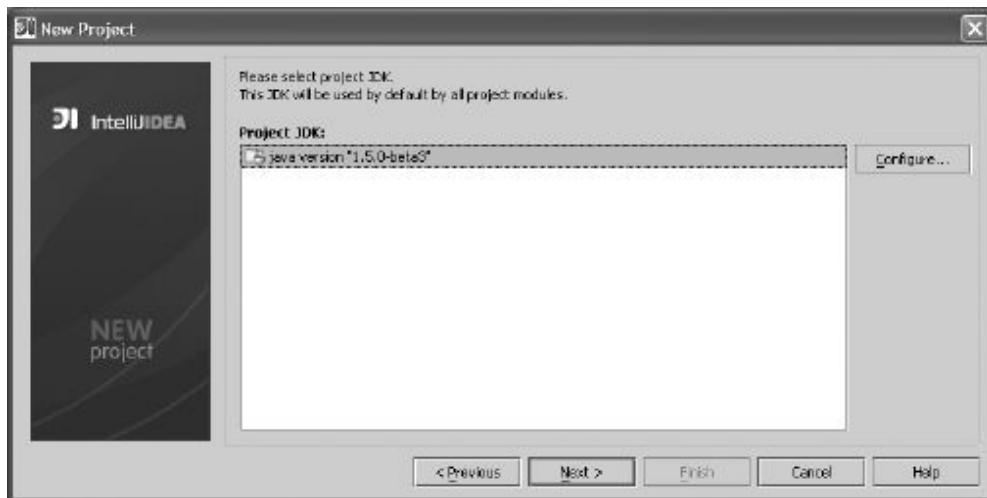


Figure C.1

¹IDEA saves information on recently opened projects, so usually you will not need to worry about the location.



The Hello Project

Figure C.2

The initial project screen, shown in Figure C.3, is spartan. A tab marked Project appears in the upper left corner, oriented vertically. Click this tab to open the Project tool window (see Figure C.4).

Select the top entry in the hierarchy tree—the first entry named **hello**. This is the project. The next entry below, also **hello**, is the module. Simple projects you build usually will need to contain only one module.

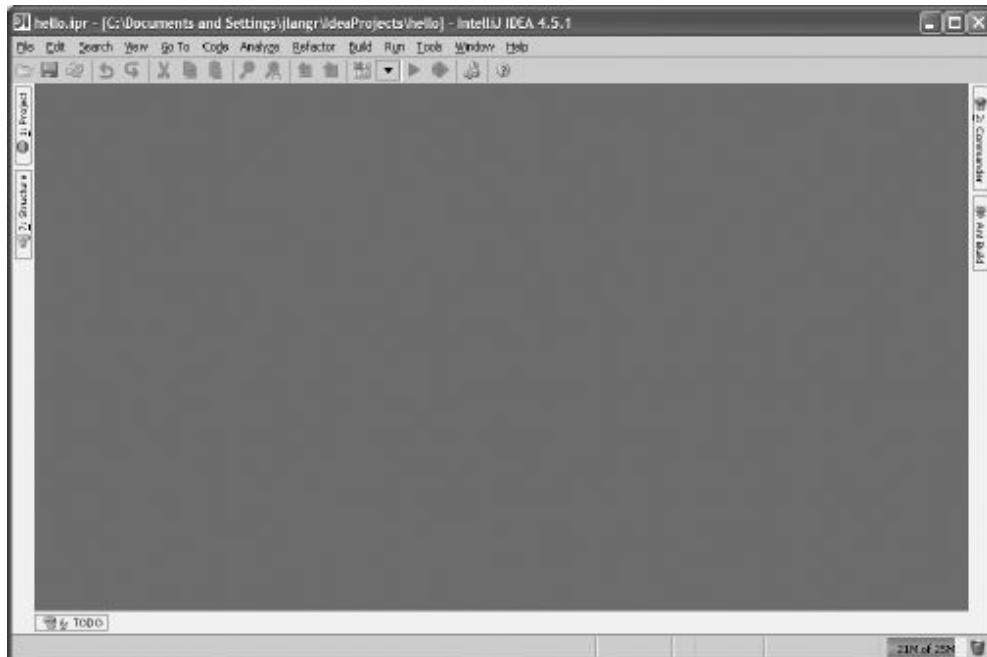


Figure C.3

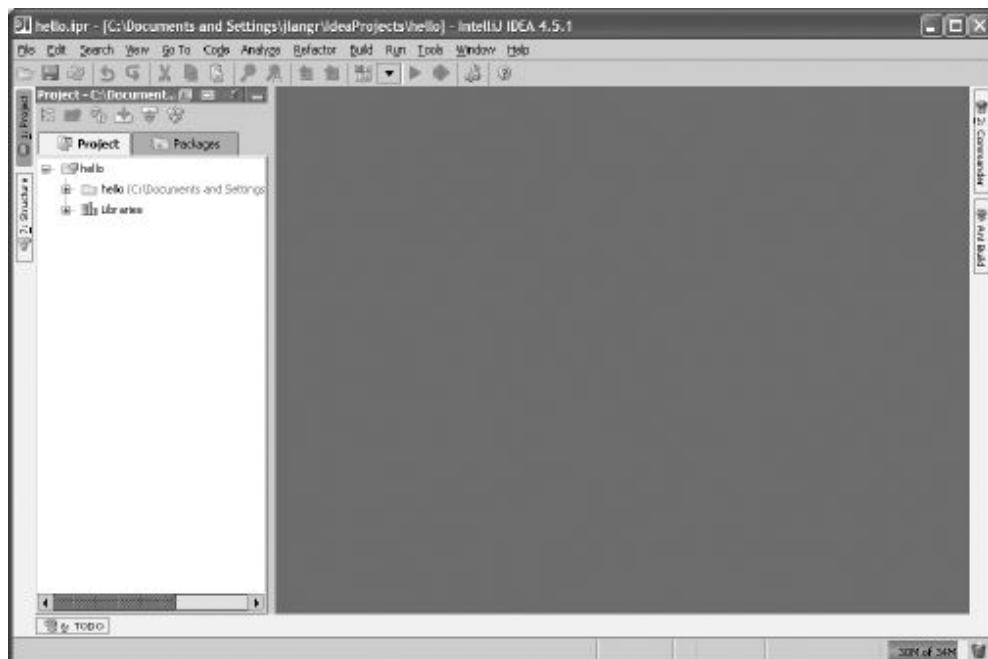
The Hello Project

Figure C.4

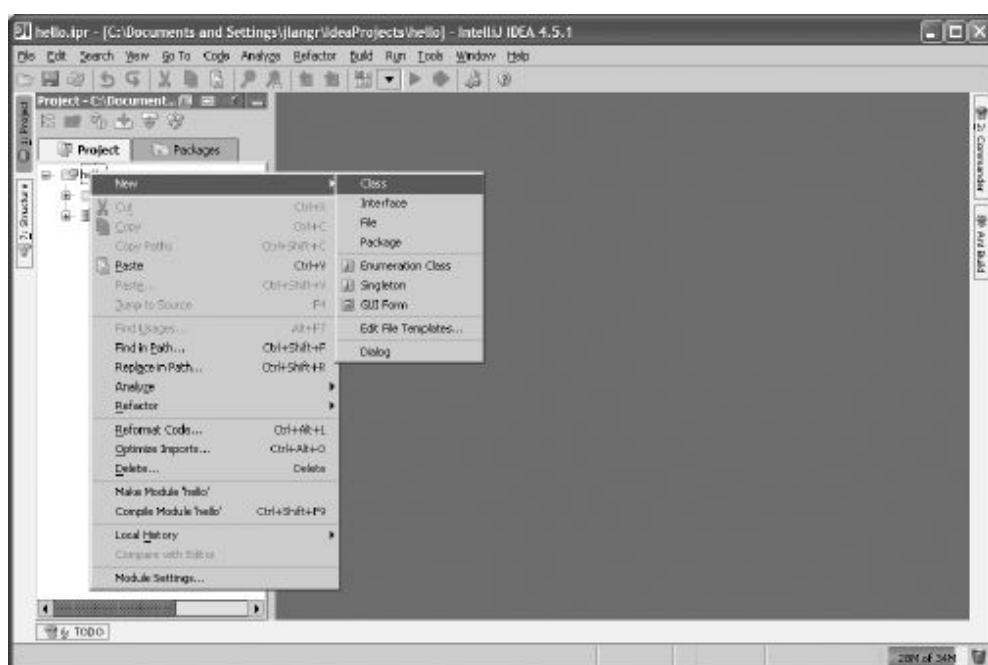


Figure C.5

Click your right mouse button after selecting the hello project in order to bring up the project context menu. The first entry in this menu is New; select it. You should see a screen that looks like Figure C.5.

Click the menu item Class. When prompted for a class name, enter Hello. Make sure you enter the class name with the correct case! You should see an editor with a starting definition for the class Hello (see Figure C.6).

In the Hello.java editor window, change the code to reflect the Hello code as originally presented in Chapter II, Setting Up:

```
class Hello {  
    public static void main(String[] args) {  
        System.out.println("hello world");  
    }  
}
```

The Hello Project

The Project tool window to the left now shows the expanded project hierarchy. Beneath the src entry, you see your new class **Hello** listed. Right-click this class entry to bring up the context menu. See Figure C.7.

Click the menu entry Run "Hello.main()", third from the bottom of the menu. Note that you can press the key combination Ctrl+Shift+F10 as an alternative to using the menu. You need not save your source files before attempting to run programs—IDEA saves them for you automatically.

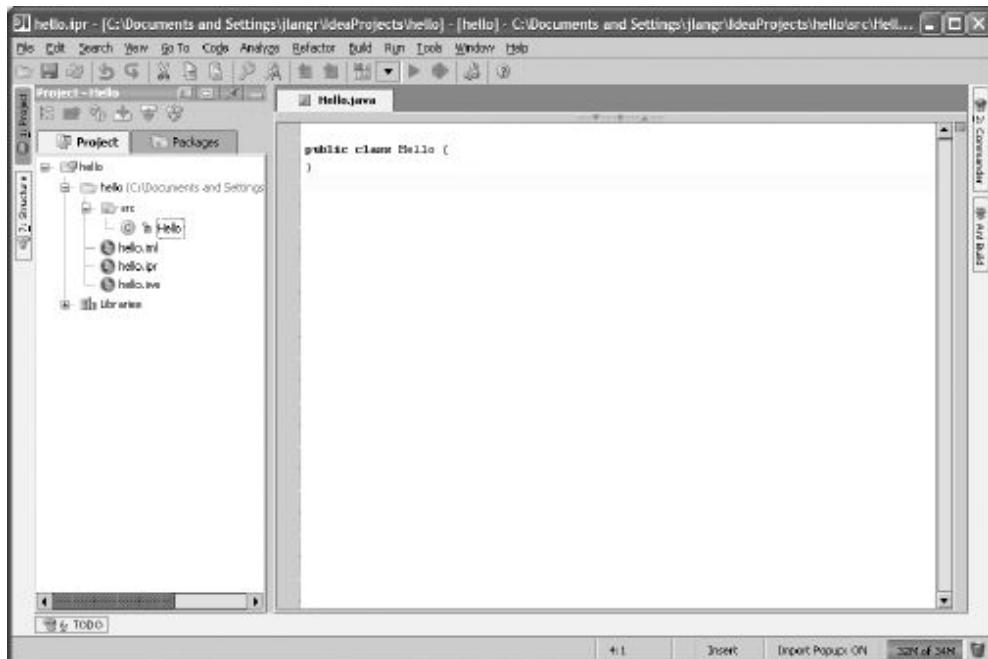


Figure C.6

The Hello Project

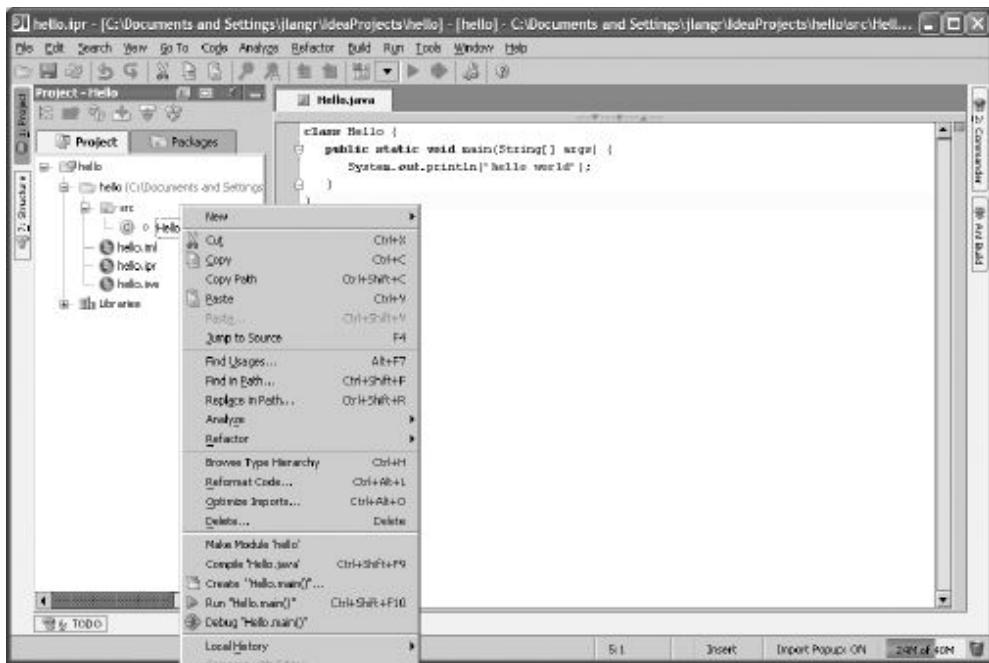


Figure C.7

IDEA might present you with a dialog asking whether or not you want to create an output directory. If so, simply click Yes.

A Compile Progress dialog should appear briefly. Before IDEA can execute your program, it must compile it to ensure there are no problems. The first time you compile, it may take a little longer; subsequent runs should begin more quickly.² Presuming that you have no compile errors, you should see an output window at the bottom of IDEA (see Figure C.8).

Success!

Next, you'll change the source code to deliberately contain errors in order to see how IDEA deals with them.

Change the source code in the Hello.java editor to something that should generate an error. Here's one possibility:

```
class Hello {
    public static void main(String[] args) {
        xxxSystem.out.println("hello world");
    }
}
```

²The slow initial compilation appears to occur each time you start IDEA.

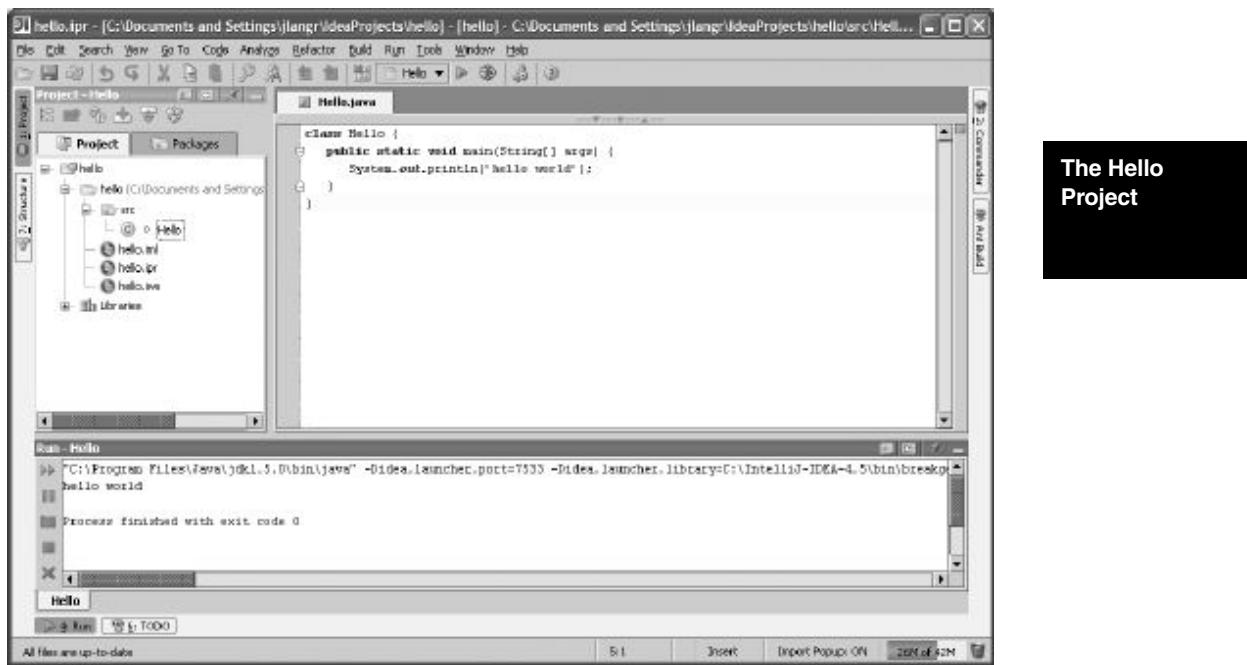


Figure C.8

Observe that off to the right, there are a couple of visual indicators telling you that the code has problems. In the upper right corner of the editor a small square will appear; it will be red. It was green until you mucked up the code. Below this square, you will see one or more thin red lines. These lines indicate the problem spots in the code. You can hold your mouse over a line to see the error message. You can click on the line to navigate to the problem spot.

Try running Hello again and see what happens. Since you've already executed Hello one time, you can quickly execute it again using one of at least two ways: either press the Shift-F10 key combination or click the Run arrow (a green arrow pointing right, like the play button on a CD player). The Run arrow is located on the toolbar beneath the menu and above the Hello.java editor.

Instead of seeing the Run tool window with the correct output, you see the Messages tool window (Figure C.9). This window presents you with a complete list of error messages. Off to the left are various icons that help you manage the messages. Hold your mouse over any one of the icons for a second or so to see a tool tip that explains the icon's use.

If you double-click a specific error message in the Messages tool window, IDEA will put the text cursor on the specific location in Hello.java that contains the error. Correct the error and re-run Hello.

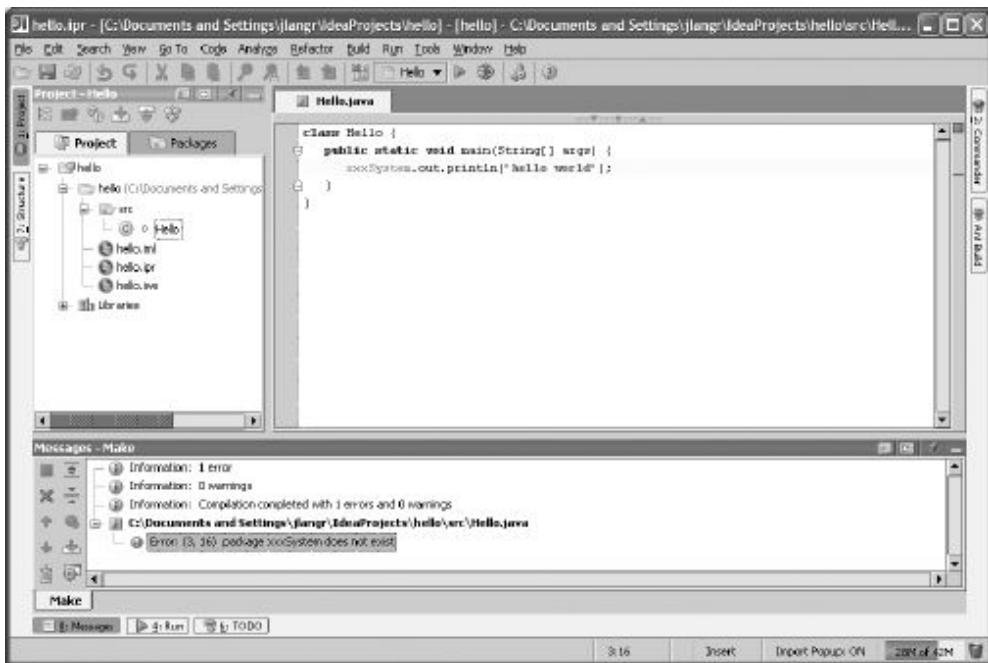
Running Tests

Figure C.9

Running Tests

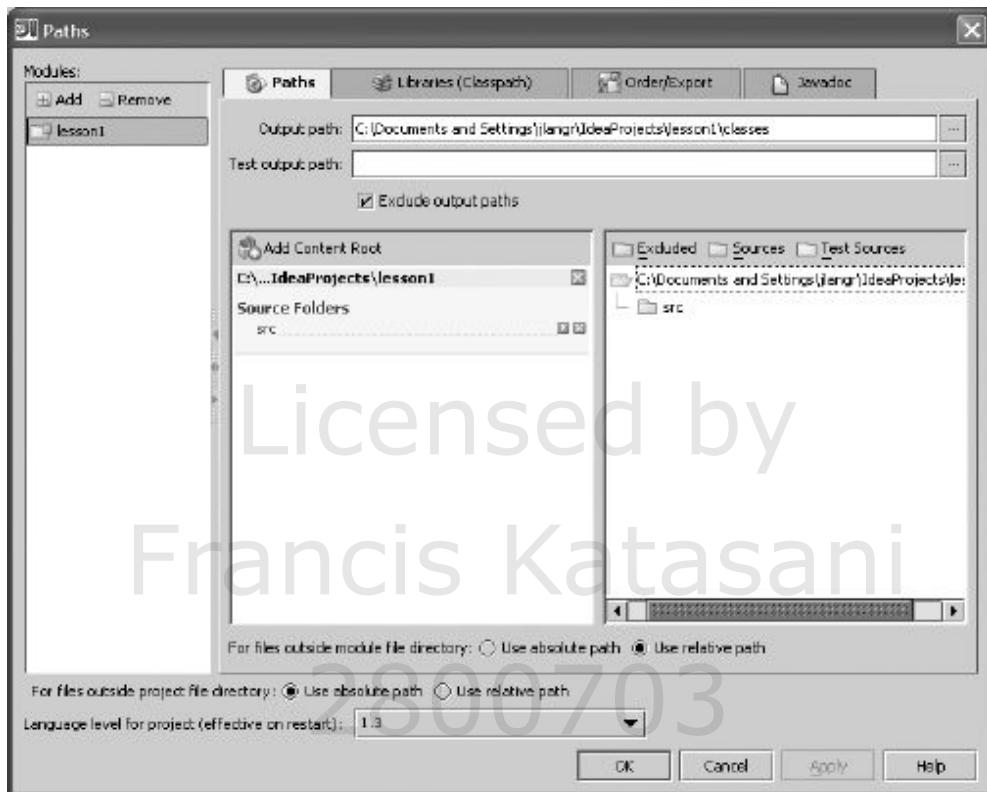
This section walks you through the first part of Lesson 1, in which you create a `StudentTest` class and a corresponding `Student` class. The goal of this section is to show you how to effectively do test-driven development in IDEA.

Start by creating a new project. Click on the File menu, then click New Project. IDEA will present you with the same series of dialogs as when you created the `hello` project. This time, type the name of the project as `lesson1`. Then follow the same steps as when you created the `hello` project—keep clicking Next. When you click Finish, IDEA gives you the opportunity to open the project in a new frame. Select Yes.

Before you enter code, you will need to change some project settings. From the Project tools window, select the Project (`lesson1`). Right-click to bring up the context menu and select Module Settings.

You should see the Paths dialog, shown in Figure C.10.

First, you must configure the `lesson1` module to recognize the J2SE 5.0 language level (if it doesn't already). Near the bottom of the dialog is a drop-down list marked Language level for project (effective on restart). The paren-



Running Tests

Figure C.10

thesized comment suggests that you'll need to exit IDEA and restart it for changes to be effective.³ You'll do this shortly. Now, click on the drop-down list to expose the selection list. Choose the entry that indicates version 5.0.

Next, you must specify where the module can find the JUnit JAR file. (If you're not sure what this means, refer to Lesson 1 as well as Chapter II on setting up.) Click on the tab marked Libraries (Classpath) (it appears at the top of the Paths dialog). Your Paths dialog should look like the screen snapshot shown in Figure C.11.

Currently the Paths dialog shows that you have no libraries (JAR files) on your classpath. Click the button Add Jar/Directory. You will see a Select Path dialog. Navigate to the directory in which you installed JUnit, select *junit.jar*, and click OK. You should be back at the Paths dialog.

Click OK to close the Paths dialog. Exit IDEA, then restart IDEA. Make sure you completely exit IDEA—close the hello project frame if it remains open.

³While some of the new features of J2SE 5.0 will work, others will not do so until you restart.

Running Tests

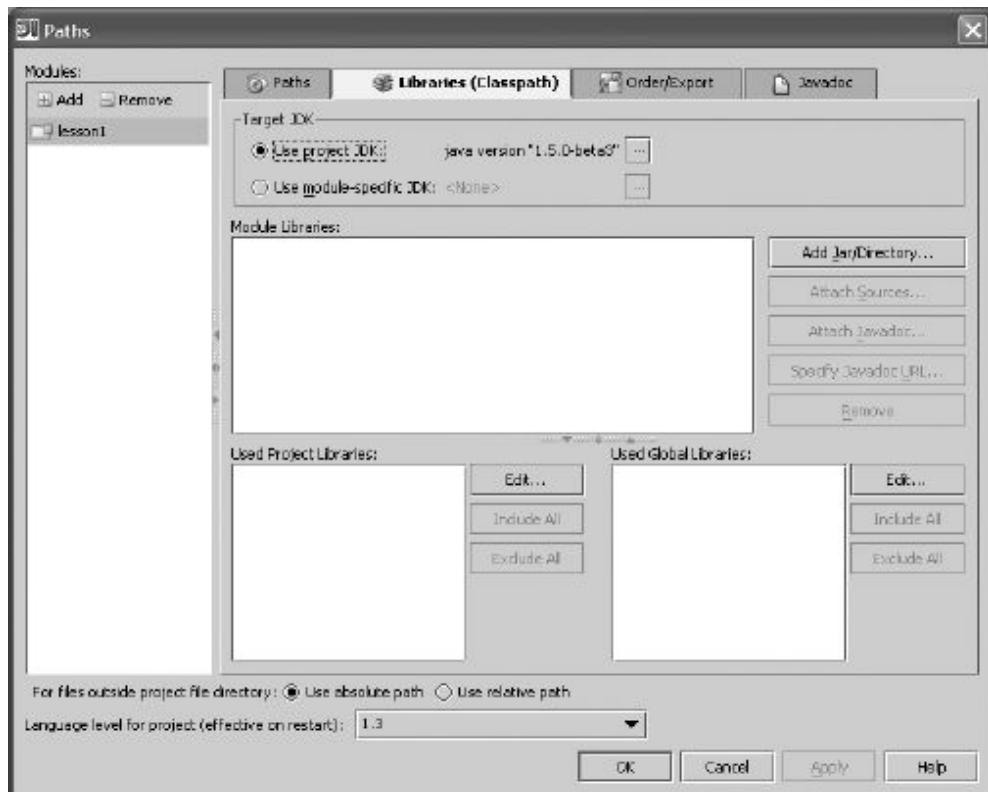


Figure C.11

IDEA will probably reopen with the hello project, not the **lesson1** project. If it does so, select File→Close Project from the menu. Then click File→Reopen and select the **lesson1** project from the list of recently opened projects. If the **lesson1** project does not appear, click File→Open Project. You will then need to navigate to the directory in which you created the project. If you accepted the defaults, under Windows this directory is probably **C:\Documents and Settings\userName\IdeaProjects\lesson1**, where **userName** is your Windows login name.

Once you have reopened **lesson1**, right-click in the Project tool window. Select New→Class from the context menu. For the class name, enter **StudentTest**. In the **StudentTest.java** editor, enter the initial code from Lesson 1 for the **StudentTest** class:

```
public class StudentTest extends junit.framework.TestCase {  
}
```

From the Project tool window tree, right-click and select Run “**StudentTest**”. Create an output directory if asked to do so. You should see something like the window in Figure C.12.

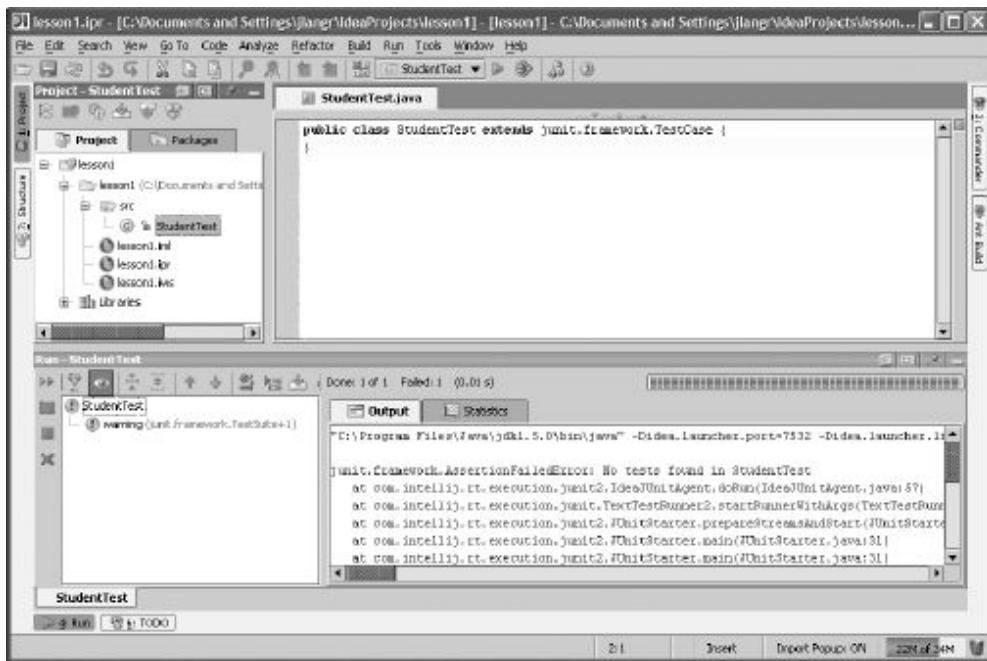


Figure C.12

The pane in the lower right quadrant shows output from the test run. This is JUnit output—IDEA has integrated JUnit directly into the IDE.

If you can't see enough of the output, drag the slider bar up. The slider bar appears just above the Run tool window and separates the top half of IDEA from the bottom half. Move the mouse slowly over the slider until you see a double-headed north-south arrow, then click and drag.

Make sure you're following along with the directions in Lesson 1 as you proceed. As expected, the Output window shows a failure, since you have defined no tests in StudentTest. Modify the StudentTest code:

```
public class StudentTest extends junit.framework.TestCase {
    public void testCreate() {
    }
}
```

Rerun (Shift-F10). Now, instead of a red bar and an error in the Output window, you see a green bar and two lines of output. The first line shows the actual java command passed to the operating system. The second line says that the “Process finished with exit code 0.” Good. You demonstrated failure, corrected the problem, then demonstrated success.

Next, the fun part. Modify the StudentTest code again:

```
public class StudentTest extends junit.framework.TestCase {
    public void testCreate() {
        new Student("Jane Doe");
    }
}
```

Running Tests

You should see red indicators to the right. The class name `Student` appears in red. You can hold the mouse over any red for more information. If you click on the class name `Student`, you will see a little light bulb to the left of the line after a short delay. Click on the light bulb. See Figure C.13.

The light bulb represents the “intention actions” tool. It presents you with a list of actions that you can effect. As an alternative to clicking the light bulb, you can bring up the list of actions using the Alt-Enter key combination. IDEA builds the intention actions list based on what it thinks you want to do. In this case you can select the only entry in the list, Create Class “`Student`”. When asked for a package, just click OK.

Figure C.14 shows that you now have two editors, `StudentTest.java` and `Student.java`. You can bring up either one by clicking on its tab.

The intention actions tool is one of many ways in which IDEA reduces the amount of code you must type. You have a `Student` class with a bunch of the relevant code already filled in. The generated class code is currently in “template” mode—you can press tab to move through various relevant

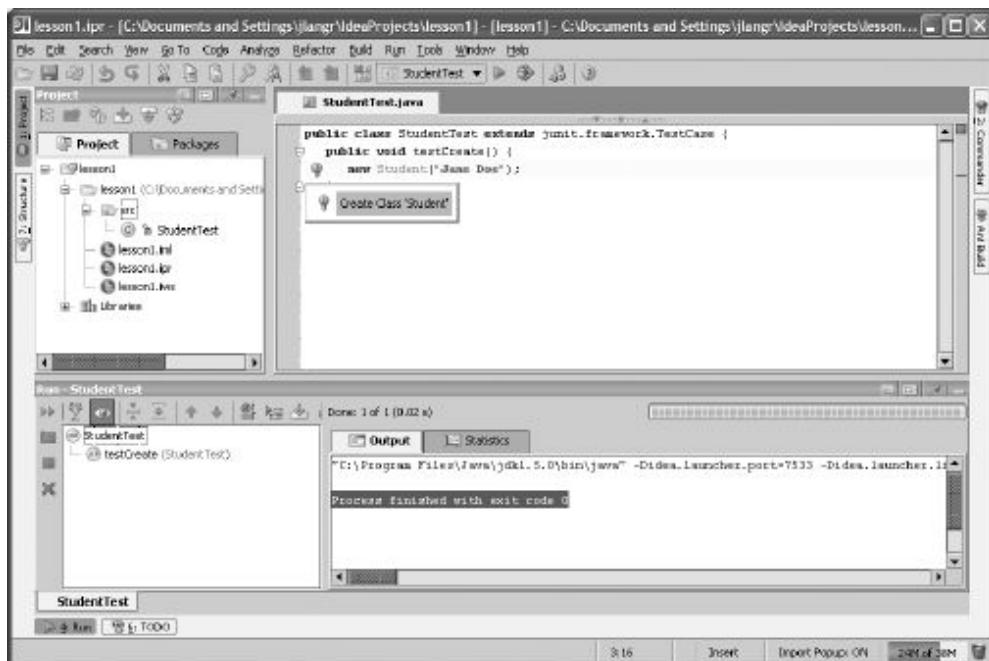


Figure C.13

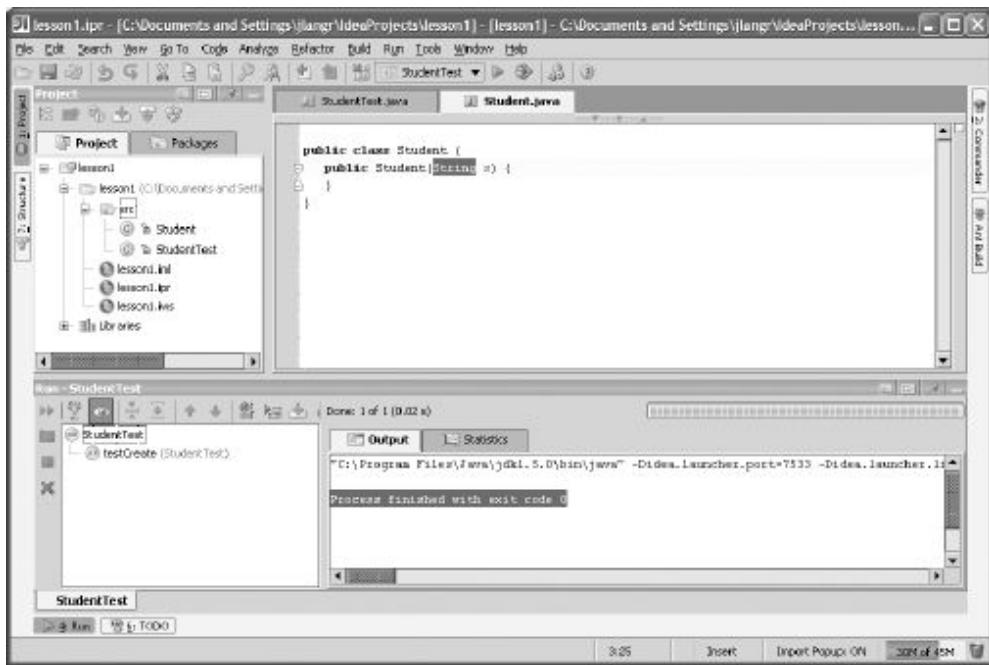


Figure C.14

fields. This gives you the opportunity to replace template code with your specifics. Tab to the argument field and replace the argument name *s* with *name*. Tab again or press the Esc key to exit template mode.

You should see no red indicators. Rerun and ensure you see green results from JUnit.

Continue with the remainder of the Lesson 1 exercise. Take advantage of the various intention actions as IDEA presents them to you. Experiment with alternate ways of effecting things. If you've been leaning toward mouse-directed options, try the key combinations to see if they help you go faster. And vice versa! Dig through the menus available and see if you can effect actions from there.

Taking Advantage of IDEA

IDEA, like most modern IDEs, is full of many time-saving features. It behooves you to learn how to master your IDE. The difference between a novice using only the basic features versus an expert taking full advantage of the IDE can be measured in weeks over the lifetime of a one-year project.

Code Completion

One valuable feature is code completion, which I'll also refer to as auto-completion. When you begin typing things that IDEA can recognize, such as class names, package names, and keywords, click Ctrl-Space. IDEA will do one of three things: It may say "No suggestions," it may automatically complete the text you began typing, or it may bring up a list of selections to choose from. You can navigate to the selection using the cursor (arrow) keys and press Enter or you can click on the desired selection. IDEA will finish your typing based on this selection.

Pressing the Enter key results in the new text being inserted at the current location. If you instead press the Tab key, the new text will replace any text immediately to the right.

Get in the habit of pressing Ctrl-Space after typing two or three significant characters. I press it all the time. You may be surprised at what IDEA is able to figure out about your intentions.

IDEA provides two additional kinds of code completion: Smart Type and Class Name. These are more sophisticated tools triggered by alternate key-strokes. Refer to the IDEA help system for more information (Help→Help Topics).

Navigation

One of the most significant strengths of using an IDE such as IDEA is the ease with which you can navigate through Java code. Click on the Go To menu. The size of this menu suggests how flexible this feature is. Click on the Class selection from the Go To menu. Type the three letters `stu`. Pause. You will see a drop-down list of appropriate classes to choose from.

Class navigation is basic but important. More valuable is the ability to navigate from directly within code. Open the `StudentTest` class. From the line that reads `new Student("Jane Doe");`, click on the class name `Student`. Then press the Ctrl-b key combination (b for "browse," I suppose). Pressing Ctrl-b is equivalent to selecting the menu option Go To→Declaration.

IDEA will navigate directly into the `Student` constructor. You can return to your previous edit location by using the key combination Ctrl-Shift-Backspace. Learning all these navigation shortcuts will save you considerable time. I've watched programmers, even recently, use inferior tools. They waste several minutes every hour in manually navigating through a list of classes and scrolling through large amounts of code.

Searching is another form of navigation. IDEA understands Java and how it all fits together, making searches far more effective. Suppose you want to

find all code that invokes a certain method. Using a programmer's editor, you can do only text searches against code. Suppose you're trying to find all client classes that call a method named `getId`. There might be other classes that implement `getId`; the text searches will report uses against these classes as well.

But IDEA can use its Java smarts to help you find only the usages you're interested in. As an example: In the `Student` class, click on the constructor. Right-click and select `Find Usages` from the context menu. Click `OK` when presented with the `Find Usages` dialog. The `Find` tool window will appear at the bottom of IDEA. As with the `Messages` tool window, you can double-click an entry in the find results to navigate directly to it.

Taking
Advantage of
IDEA

Refactoring

Another important capability is the built-in refactoring support. IDEA automates a number of refactorings that allow you to quickly and safely transform your code. Take a look at all the entries in the `Refactor` menu to get an idea of the things you can do.

The simplest and perhaps most valuable refactoring appears first—`Rename`, also effected by Shift-F6. In a few keystrokes, you can safely change the name of a method. This means that IDEA also changes any other code that calls the method to reflect the new name. It also shows you the proposed rename before it applies it to your code. This allows you to back out if you recognize a problem with the potential refactoring.

I use the `Rename` refactoring all the time. Rather than waste a lot of time coming up with the “perfect” method or class name, I’ll often go with a name I know to be inferior. This allows me to immediately get moving. After coding a little, a better name usually suggests itself. A quick Shift-F6 and I’m on my way. Sometimes I might change a method name more than a couple of times before I’m satisfied.

Code Analysis

IDEA supplies you with a code analysis feature known as the code inspector (`Analyze→Inspect Code`). You execute code inspection on demand against a single source file or the entire project. The inspector looks for problem spots in your code, including things such as unused methods or variables or empty catch blocks. The list of options includes over 300 categorized entries. You pick and choose what you’re interested in: individual entries or groups of related entries or both.

The inspection can take a bit of time to execute. When an inspection completes, the inspector presents you with a tree showing all the results. Each result is a detailed explanation of the problem or potential problem in your code. From a detailed explanation, you can click to navigate to the corresponding trouble spot. When possible, IDEA will give you the option to automatically correct the problem.

IDEA may report dozens or hundreds of entries that you may not think are problems. It's up to you and your team to decide the items that are relevant. Some items are not problems at all. But virtually everything on the list can point to the potential for trouble in certain environments or circumstances.

The practice of TDD can mitigate most of these concerns. In *Agile Java*, you've learned to build things incrementally and not always to an overly robust expectation. You can "get away" with things that would be deemed unacceptable in the absence of good testing.

Relax. Some others might consider the style in *Agile Java* to be fast and loose. Certainly there is always room for improvement in my code. But I don't worry about it as much when doing TDD.

I consider the bulk of code I come across difficult to understand and maintain. Blindly adhering to standards can seem like a good idea, but often it can lead you to waste considerable time for little gain. If you instead look to the simple design goals of testability, elimination of duplication, and code expressiveness, you will rarely go wrong.

Agile Java References

- [Arnold2000] Arnold, K.; Gosling, J.; Holmes, D. *The Java Programming Language (3e)*. Sun Microsystems, 2000.
- [Astels2003] Astels, Dave. *Test-Driven Development: A Practical Guide*. Pearson Education, 2003.
- [Astels2004] Astels, Dave. “One Assertion Per Test.” <http://www.artima.com/weblogs/viewpost.jsp?thread=35578>.
- [Beck1998] Beck, Kent; Gamma, Erich. “*Test Infected: Programmers Love Writing Tests.*” <http://members.pingnet.ch/gamma/junit.htm>.
- [Bloch2001] *Effective Java Programming Language Guide*. Addison-Wesley, 2001.
- [Fowler2000] Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.
- [Fowler2003] Fowler, Martin. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3rd ed. Addison-Wesley, 2003.
- [Fowler2003a] Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.
- [Gamma1995] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. *Design Patterns*. Addison-Wesley, 1995.
- [George2002] George, Eric. “Testing Interface Compliance with Abstract Test” <http://www.placebosoft.com/abstract-test.html>.
- [Hatcher2002] Hatcher, Eric; Loughran, Steve. *Java Development with Ant*. Manning Publications Company, August 2002.
- [Heller1961] Heller, Joseph. *Catch-22*. Dell Publishing, 1961.
- [JavaGloss2004a] Green, Roedy. “Java Glossary: float.” <http://mindprod.com/jgloss/floatingpoint.html>.
- [JavaGloss2004b] Green, Roedy, “Java Glossary: weak references.” <http://mindprod.com/jgloss/weak.html>.
- [Jeffries2001] Jeffries, R.; Anderson, A.; Hendrickson, C. *Extreme Programming Installed*. Addison-Wesley, 2001.
- [Kerievsky2004] Kerievsky, Joshua. *Refactoring to Patterns*. Addison-Wesley, 2004.
- [Langr2000] Langr, Jeff. *Essential Java Style*. Prentice Hall PTR, 2000.

- [Langr2001] Langr, Jeff. "Evolution of Test and Code Via Test-First Design." <http://www.objectmentor.com/resources/articles/tfd.pdf>.
- [Langr2003] Langr, Jeff. "Don't Mock Me." <http://www.LangrSoft.com/articles/mockng.html>.
- [Lavender1996] Lavender, R. Greg; Schmidt, Douglas C. "An Object Behavioral Pattern for Concurrent Programming." <http://citeseer.ist.psu.edu/lavender96active.html>.
- [Link2003] Link, Johannes. *Unit Testing in Java: How Tests Drive the Code*. Morgan Kaufmann, 2003.
- [Martin2003] Martin, Robert. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.
- [Massol2004] Massol, Vincent. *JUnit in Action*. Manning Publications, 2004.
- [McBreen2000] McBreen, Pete. *Software Craftmanship*. Addison-Wesley, 2001.
- [Rainsberger2005] Rainsberger, J. B. *JUnit Recipes*. Manning Publications, 2005.
- [Sun2004] [Java] Reference Glossary. <http://java.sun.com/docs/glossary.html>.
- [Travis2002] Travis, Gregory M. *JDK 1.4 Tutorial*. Manning Publications, 2002.
- [Venners2003] Venners, Bill; Eckel, Bruce. "The Trouble with Checked Exceptions: A Conversation With Anders Hejlsberg, Part II." <http://www.artima.com/intv/handcuffs.html>.
- [Vermeulen2000] Vermeulen, Allan, et al. *The Elements of Java Style*. Cambridge University Press, 2000.
- [WhatIs2004] "platform." <http://www.whatis.com>.
- [Wiki2004] "CodeSmell." <http://c2.com/cgi/wiki?CodeSmell>.
- [Wiki2004a] "EmptyCatchClause." <http://c2.com/cgi/wiki?EmptyCatchClause>.
- [Wiki2004b] "SimpleDesign" and "XpSimplicityRules," <http://c2.com/cgi/wiki?SimpleDesign> and <http://c2.com/cgi/wiki?XpSimplicityRules>.
- [Wikipedia2004] http://en.wikipedia.org/wiki/Java_programming_language.

Index

A

^ (exclusive-or operator), 312–313
^ (xor (exclusive-or) operator), 362
| (bit-or operator), 362, 364
| (non-short-circuited or operator),
 313
|| (logical or operator), 312
~ (logical negation operator), 362,
 365
! (not operator), 312
% (modulus operator), 355
& (bit-and operator), 362, 364
&& (and operator), 312
+ (add operator), 67
+ (String concatenation), 106, 344
++ (increment operator), 144–145
— (decrement operator), 144
; (statement terminator), 39
<< (bit shift left operator), 367
= (assignment operator), 42–43
== (reference comparison operator),
 346
>> (bit shift right operator), 367
>>> (unsigned bit shift right opera-
 tor), 367
? (wildcard character), 519
?: (ternary operator), 245–246
\n (line feed escape sequence), 105,
 109
^ (bit-xor operator), 362, 365–367
^ (logical exclusive or operator),
 312
@deprecated, 537–538
@Override annotation 216, 217,
 325, 345, 538

@param, 99
@Retention annotation, 545
@return, 99
@Target annotation, 546–547
abbreviations, 60
abstract, 204–205
abstract classes, 204–205
abstract test pattern, 221–223
abstraction, 13, 188
AbstractTableModel
 (javax.swing.table), 637
access modifiers, 122–124,
 212–213
action methods, 133
ActionListener (java.awt.event), 587
actionPerformed (java.awt.event.Ac-
 tionListener method), 587
activation, 476
active object pattern, 462
actual value, 48
adapter, 435–438 , 590–591
additional bounds, parameterized
 types, 525–526
advanced streams, 399
agile, 9–11
algorithms, hash code, 333–336
American Standard Code for Infor-
 mation Interchange
 (ASCII), 105
and operator (&&), 312
annotations, 537–538
 @Retention, 545
 @Target, 546–547
 @TestMethod, 543–545, 548
array parameters, 553–554
compatibility, 561

- annotations (*cont.*)
 - complex, 558
 - default values, 557
 - member-value pairs, 555
 - multiple parameter sup-port, 555–556
 - package, 559–560
 - single-value, 550–552
 - types, 558–559
- anonymous inner classes, 433–440, 471
- Ant, 25, 41, 128–131, 527, 542–543
- API (application programming interface), 3, 11, 426
 - documentation, 72
 - Java API Library, 368
- application, 11
 - exceptions 276
- application programming interface.
 - See* API
- argument, 36, 38
- argument list, 39
- arithmetic
 - BigDecimal (java.math), 350–353
 - bit manipulation, 360–367
 - expression evaluation
 - order, 356–357
 - functions, 368–369
 - infinity, 358–359
 - integers, 354
 - NaN, 357–358
 - numeric casting, 355–356
 - numeric overflow, 359
 - random numbers, 371–374
- ArithmException, 358
- array initialization, 258–259, 262
- ArrayList (java.util), 71–73, 189, 246, 510–511
- arrays, 255–262, 264–266
 - annotations, 553–554
 - parameterized types, 530
 - references, 263
- Arrays.equals, 263
- ASCII, 105
- asList (java.util.Arrays method), 266
- Assert (junit.framework), 421, 535, 537
- assert keyword, 537
- assertEquals (junit.framework.Assert method), 46–48, 175, 324, 537
- assertFalse (junit.framework.Assert method), 150
- AssertionError, 536
- assertions, 46–48, 535–536
 - failure messages, 152
 - JUnit, 537
- assertTrue (junit.framework.Assert method), 336
- assignment, 42–43
- atomic variables, 501–502
- atomic wrappers, 506
- attributes, 15, 51, 65
- autoboxing, 254
- autounboxing, 255
- avoiding numeric overflow, 359
- await (java.util.concurrent.locks.Condition method), 493
- AWT, 35, 566, 572

B

- background threads, 474
- base class, 203
- base class references, 220
- batch files, 40
- beans. *See* JavaBeans
- Beck, Kent, 19
- BigDecimal (java.math), 350–353

binary numbers, bit manipulation, 360–361
 binding parameterized types, 72
 additional bounds, 525–526
 arrays, 530
 checked collections, 528, 530
 generic methods, 522
 limitations of, 531
 raw types, 526–527
 reflection, 532
 support, 514–515
 upper bound constraints, 516–518
 wildcards, 518–524
 bit manipulation, 360–367
 bit shift left operator. *See <<*
 bit shift right operator. *See >>*
 bit shifting, 367–368
 bit-and operator. *See &*
 bit-or operator. *See |*
 bit-xor operator. *See ^*
 BitSet (java.util), 246, 368
 bitwise multiplication, 362
 Bloch, Joshua, 505
 BlockingQueue (java.util.concurrent), 483–484
 blocks
 catch
 exceptions, 271
 rethrowing exceptions, 282–284
 finally, 285–287
 try-catch exceptions, 271
 Boolean, 252
 boolean expressions, 312. *See also*
 logical operators
 boolean type, 150–154, 312, 362
 border, 608
 BorderFactory (javax.swing), 602, 609

BorderLayout (java.awt), 597–599
 bounds
 arrays, 530
 parameterized types, 525–526
 BoxLayout (javax.swing), 601, 681
 branch, 172
 break statements, 197–198, 242, 243, 244–245
 bridge icon, 8, 53, 54, 85, 91, 95, 96, 112, 117, 120, 122, 134, 141, 149, 155, 168, 183, 187, 207, 234, 260, 279, 324, 330, 346, 420, 426, 447, 459, 666, 673, 693, 699
 BufferedReader (java.io), 387, 392, 394, 691
 BufferedWriter (java.io), 387, 394
 build targets, 131
 build.xml, 128–131
 byte type, 354
 Byte, 252
 byte codes, 12
 byte streams, 380
 ByteArrayInputStream, 391, 416
 ByteArrayOutputStream, 392, 394, 416

C

C language, 2, 232, 471
 Calendar (java.util), 87–90, 681
 call by reference, 681–683
 call by value, 681–683
 callbacks, 471–472, 587
 calling superclass constructors, 211–215
 camel case, 59
 case labels, 196–198

casting
 numeric, 355–356
 references, 230–231
catch blocks, 271, 282–284
catches, exceptions, 270, 280–282
causes, exception, 283
char type, 103–106, 353–354
character, 103–105, 252
 literals, 104–105
 streams, 380–385
 strings, 106. *See also* strings
 system properties, 109–110
checked collections, parameterized
 types, 528–530
checked exceptions, 273–278
checked wrappers, 528
chess, 62
ChoiceFormat (`java.text`),
 679–690
class, 14, 73
 constants, 84–86, 284
 diagram, 16
 file, 12, 27
 library, 11–12
 loader, 456, 693–694
 method, 133–136, 138–141
 variables, 137, 138–141
Class class, 442–444, 453, 667
 modifiers, 447
class keyword, 33, 167
ClassCastException, 327–328, 530,
 584
ClassNotFoundException, 401,
 459
classpath attribute, 131, 441–448
ClassPathTestCollector (`junit.runner`), 441–442
client, 83
clone (`Object` method), 663–664
Cloneable, 663, 664
CloneNotSupportedException, 664
cloning, 662–664
closing a solution. *See open-closed principle*
code smells, 94
coding standards, 61
Collection, 339, 482, 509–510
Collections Framework, 73, 510
Collections.sort, 164–166
collective code ownership, 98
collisions, 332–333
command objects, 462
comment, 96–100
comment, multiline, 97
comment, single-line, 96
Comparable, 166–168, 169–171
Comparator objects, sorting, 342
compareTo (`Comparable` method),
 167, 169–171
compilation, 81
 Ant, 128–131
compile errors, 35, 40, 41, 45, 76,
 117
compiler, 11, 26
compiler warnings, 91–92
complex annotations, 558
Component (`java.awt`), 574, 578,
 617, 698
ComponentOrientation (`java.awt`),
 681
composition, 445
compound assignment, 144,
 362
concatenation of strings,
 106–107
Condition (`java.util.concurrent.locks`), 493
conditional branching, 172. *See also*
 if statement
conditional expressions, 245
conditionals, boolean values,
 153

Connection (`java.sql`), 668
connection pool, 668
console, 118
`ConsoleHandler` (`java.util.logging`),
 294, 298
class variables, 137–141
classes, 84–86
 enum, 364–365
constraints, upper bound,
 515–518
constructors, 41–42, 576
 assertions, 46–48
 chaining, 170
 default, 69–70
 enums, 209
 exceptions, 300. *See also* exceptions
 fields (initializing), 68–69
 inheritance, 218–219
 overloaded, 86–87
 superclasses, 211–214
`Container` (`java.awt`), 573
continue statement, 243–244
contract, design by, 220–221
contract, equality, 326
control statements, 242
 break statement, 243
 continue statement, 243–244
 labeled break statement,
 244–245
 labeled continue statement,
 244–245
 return statement, 45–46, 173,
 178
controller, 571
conventions used in book, 6
conventions, naming, 59–60
cooperative multitasking, 477
covariance, 663
creation method, 146
creation test, 64
critical section, 481
`currentThread` (`Thread` method),
 494, 502
custom exceptions, 276–278

D

daemon thread, 502
data streams (IO), 395–399
 character streams, 380–385
 management of, 379–380
 writing to files, 385–387
`DataInputStream` (`java.io`), 398
`DataOutputStream` (`java.io`),
 395–397
`Date` (`java.util`), 86–91
`DateFormat` (`java.text`), 681
dates, 86–91
deadlocks, 495
debugger, 22
debugging (`toString` method),
 343–344
`decode` (`Integer` method), 371
decrement operator, 144
deep clone, 664
default access, 122
default constructor, 69–70, 218–219
default package, 79–80
default value, annotations, 557
`DefaultListModel` (`javax.swing`),
 590
`DefaultTableModel`
 (`javax.swing.table`), 637
dependency, 16, 188, 426
dependency inversion principle, 188
deployment (Ant), 128–131
deprecated, 87–88
deprecation warnings, 91–92
design, 10, 32, 53, 115, 147
 synchronization, 506

Design Patterns, 268, 448
do keyword, 239
do loop, 239–240, 241
doClick (javax.swing.JButton), 588
Document (javax.swing.text), 620
DocumentFilter (javax.swing.text), 620–621
domain map, 639
double type, 173–174, 177, 355–356
Double, 252, 357, 358
downloading code, 6
driver, JDBC, 665
DriverManager (java.sql), 667
duplication, 53, 85, 111, 113–118, 147, 168, 185, 195–196, 440, 673
dynamic proxy class, 448–449, 455–458

E

Eclipse, 21, 35, 717
EJBs, 3, 400, 449, 665, 697, 699
element types, 546–547
empty catch clause, 279, 301
encapsulation, 14, 83–84, 140, 272
enhanced for loop, 111
enterprise application development, 3
enum keyword, 179–181, 209–210
enumerated types, 179–181
Enumeration (java.util), 248–249
EnumMap (java.util), 199–201, 342
EnumSet (java.util), 342
environment. *See* system environment
epoch, 86
equality, 323–330
String, 345–346

equals (java.util.Arrays method), 263
equals (Object method), 324–330, 335, 346
erasure, 514–515, 531
Error, 276
escape sequence, 105
Exception, 276
exceptions, 73, 158–159, 270–287
causes, 283
checked, 273–278
finally block, 285–287
hierarchies, 275–276
logging, 290–292, 294–305
messages, 279–280
multiple, 280–282
refactoring, 287–289
rethrowing, 282–284
stack traces, 285
types, 276–278
exclusive-or (xor) operator (^), 312–313
exercises, 6, 62, 100, 131, 160
exit (System method), 503–504, 552
expected value, 48
expression evaluation order, 356–357
expressions, boolean, 150–154, 312
extending methods, 206–207
extends keyword, 33, 202, 516
extreme programming, 9

F

factory, 450
factory method, 93–94, 145–146
Feathers, Michael, 576
Fibonacci sequence, 239, 242

field, 51
 access modifiers, 122–127
 edits, 619–620
 initializers, 67
 primitive type initialization, 159–160
 FIFO (first-in, first-out), 474
 File (java.io), 388–389
 files
 logging, 298–299
 random access, 407–410
 redirecting, 118–119
 writing to (IO), 385–387
 FileHandler (java.util.logging), 294, 298–299, 302
 FileInputStream (java.io), 398
 FileOutputStream (java.io), 397
 FileReader (java.io), 387
 FileWriter (java.io), 387
 filter, field, 620–626
 filtered streams, 395
 final keyword, 54, 84, 438–439
 finalize method, 694–695
 finally blocks, 285–287, 399
 float type, 173–174, 355–356
 Float, 252, 357, 358
 floating point numbers. *See* floats
 floats, 173–175, 350
 FlowLayout (java.awt), 595
 flyweight pattern, 345
 for loop, 236–239, 241
 for-each loop, 111
 iterators, 249–250
 testing, 340
 formal parameter, 56–57
 format (String method), 287–289
 format specifier, 287–288
 Formatter (java.util), 679
 Formatter (java.util.logging), 299
 forName (Class method), 667
 Fowler, Martin, 16–17, 94, 322

Frame (java.awt), 610
 frame, 567
 fully qualified class name, 78

G

garbage collection, 149, 694–695
 generic methods, 522
 generics. *See* parameterized types
 getClass (Object method), 284, 328
 getEnv (System method), 689
 getProperties (System method), 683–684
 getRuntime (Runtime method), 689
 getter method, 65
 global, 134
 Goldeberg, Adele, 2
 green bar, 38–39
 GregorianCalendar (java.util), 86, 95
 GridBagConstraints (java.awt), 603–605
 GridBagLayout (java.awt), 602–605
 GridLayout (java.awt), 595–596
 guard clause, 177, 328

H

hacking, 10
 Handler (java.util.logging), 294–296
 hash code, 330–337
 hash tables, 313–319, 321–331
 algorithms, 333–336
 collisions, 332–333
 HashMap class, 337–341
 implementation, 341–343
 hashCode (Object method), 331–337

- HashMap (`java.util`), 314–315, 330, 332, 333, 339, 511
 HashSet (`java.util`), 335, 339
 Hashtable (`java.util`), 246–247, 482, 683
 heavyweight process, 10
 hello world, 2, 26–27, 718
 hexadecimal, 43, 353
 Hibernate, 665
 hourglass, 647
- I**
- i18n. *See* internationalization
 icon, 610
 IDE, 21–22
 IDEA. *See* IntelliJ IDEA
 identifier, 59
 IdentityHashMap (`java.util`), 343
 if statement, 172–173, 176–177, 311–312
 ignoring test methods, 557
 IllegalArgumentException, 277
 Image (`java.awt`), 610
 ImageIcon (`javax.swing`), 612
 immutable, 58, 107
 implementing interfaces, 167, 169–171, 184, 205
 implements keyword, 168, 202
 import statement, 78–81, 93
 static, 141–143
 increment operator. *See* `++`
 incremental refactoring, 75
 incrementing, 67–68, 144–145
 infinite loop, 239
 infinity, 358–359
 inherit, 33
 inheritance, 17–18, 74, 201–204
 initial value, 67
 initialization, 68–69, 157–160, 434, 468
 inlining code, 186
 inner classes, 412–413, 438
 anonymous, 433–435, 440
 InputStream (`java.io`), 389, 391
 InputStreamReader (`java.io`), 389–390, 392, 394
 instance initializers, 434
 instance variable. *See* field
 instanceof operator, 328–329, 574–575, 600
 instantiate, 16
 instrumentation, 695
 int type, 65–68, 354
 Integer, 252, 359, 370
 integers
 bit manipulation, 360–367
 math, 354–355
 integrated development environment. *See* IDE
 IntelliJ IDEA, 21, 22, 717–732
 interface keyword, 167
 interface references, 187–188
 interfaces, 73, 166–169, 184
 InternalError, 276
 internationalization, 85, 673–681
 interpreter, 12
 interrupt (Thread method), 486
 InterruptedException, 472, 486
 inverting dependences, 188
 InvocationHandler (`java.lang.reflect`), 451, 456, 458
 InvocationTargetException (`java.lang.reflect`), 454
 invokeAndWait
 (`javax.swing.SwingUtilities` method), 649
 invokeLater (`javax.swing.SwingUtilities` method), 649

IO (input-output), 379
 advanced streams, 399
 application testing, 393–395
 byte streams, 389
 character streams, 380–385
 data streams, 395–399
 files
 File (java.io), 388–389
 writing to, 385–387
 functionality, 422
 interfaces, 390–393
 management, 379–380
 object streams, 399–406
 random access files, 407–410
 IOException (java.io), 382
 isAlive (Thread method), 485
 Iterable (java.util), 249–250
 iteration, map, 337–341
 Iterator (java.util), 247–249
 iterator, 247–249

J

J2EE, 3, 698
 J2ME, 3
 J2SE, 3
 jar command, 654–655
 JAR, 34, 653–656
 java command, 23–24, 27–28, 655, 656
 Java, 2, 11–13
 API documentation, 72–73
 language specification, 4
 platform, 11
 SDK, 23
 version, 12, 23–24
 virtual machine. *See* virtual machine
 Java Archive. *See* JAR

java.awt package, 572
 java.io package, 379–380, 389
 java.lang.instrument package, 695
 java.lang.management package, 696
 java.lang.ref package, 694
 java.net package, 696
 java.security package, 698
 java.util package, 73
 java.util.concurrent package, 483
 java.util.concurrent.atomic package, 506
 JAVA_HOME, 23
 JavaBeans, 698
 javac command, 26–27, 35, 41, 91, 527
 javadoc command, 99
 javadoc comment, 97–100
 JavaRanch, 29
 JavaServer Faces, 699
 javax.swing package, 572
 JAX_RPC, 699
 JAXP, 699
 JAXR, 699
 JButton (javax.swing), 581, 587
 JComponent (javax.swing), 573
 JDBC, 3, 495, 664–673
 JDO, 665
 Jeff’s Rule of Statics, 149–150
 Jeffries, Ron, 576
 JetBrains, 21, 717
 JFormattedTextField (javax.swing), 620, 626–628
 JFrame (javax.swing), 567–570, 572, 594, 610
 JLabel, 573
 JList (javax.swing), 581–582, 589–590, 635
 JLS. *See* Java language specification
 JMS, 3, 699
 JNI, 697

join (Thread method), 485
 JPanel, 572–574
 JRE, 23
 JScrollPane (javax.swing), 608
 JSPs, 699
 JSTL, 699
 JTable (javax.swing), 635, 640
 JTextField (javax.swing), 581, 620
 JTextPane (javax.swing), 659
 JUnit, 25, 33–38, 47–49, 50, 329,
 336, 343, 440–446, 537, 548
 JUnit setUp method, 81–82
 JUnit suite, 70–71
 JUnitPerf, 336
 JVM. *See* VM

K

Kerievsky, Joshua, 146
 KeyAdapter (java.awt.event), 616
 KeyListener (java.awt.event), 616

L

labeled break, 244–245
 labeled continue, 244–245
 layout managers, 594
 lazy initialization, 201
 leaks, memory, 43
 length (array field), 262
 Level (java.util.logging), 293
 lightweight process, 10
 line feed. *See* \n
 line.separator system property, 109
 LinkedBlockingQueue (java.util.concurrent), 484–486
 LinkedHashMap (java.util), 343
 LinkedHashSet (java.util), 343
 LinkedList (java.util), 474

List (java.util), 189, 249, 510–511
 listener, 471
 ListModel (javax.swing), 590
 literals
 boolean, 150–154
 characters, 105
 classes, 70, 84–86
 numeric, 65
 strings, 39
 local variable, 42–44
 Locale (java.util), 676, 678, 681
 locale, 673
 localization, 673, 676–678
 Lock (java.util.concurrent.locks),
 492–493
 locking monitors, 481–482
 Log4J, 291
 Logger (java.util.logging), 292–293,
 304
 logging, 290–305
 logging handlers, 294
 logging hierarchies, 304
 logging levels, 302–304
 logical bit operators, 361–367
 logical negation. *See* ~
 logical operators, 311–313
 long type, 354
 Long, 232
 loops
 control statements, 242
 break statements, 243
 continue statements, 243–244
 labeled break statements,
 244–245
 labeled continue statements,
 244–245
 constructs, 232–233
 comparing, 240–241
 do loops, 239–240
 for loops, 236–239
 for-each loops, 249–250

while loop, 234–235
for-each, 111, 340
lower bounds, 523–524

M

main method, 261, 393–395, 568
make command, 41
`MalformedURLException (java.net)`,
 273–274
manifest, JAR, 655–656
`Map (java.util)`, 199–201, 341, 343,
 511
`map`, 199–201
`Map.Entry (java.util)`, 340
marker annotation, 550
marker interfaces, 401, 663
masks, defining, 363
`Matcher (java.util.regex)`, 661
`Math`, 135, 368–370
mathematics
 `BigDecimal` class, 350–353
 bit manipulation, 360–367
 expression evaluation order,
 356–357
 functions, 368–369
 infinity, 358–359
 integers, 354
 `NaN`, 357–358
 numeric casting, 355–356
 numeric overflow, 359
 primitive numerics, 353–356
 random numbers, 371–374
 wrapper classes, 370–371
member, 140
member-value pairs, 555
memory leak, 43–44
merge sort, 165
`MessageFormat (java.text)`, 679,
 680

`Method (java.lang.reflect)`, 453,
 454, 458
`method`, 37–38
 static, 71
methodology, 9–11
mixed case, 59
mnemonic, button, 614–615
mock objects. *See* mocking
mocking, 372–373, 425–426,
 429–430, 504–505
model, 571
model-view-controller, 571
`Modifier (java.lang.reflect)`,
 447–448
modifiers, class, 446–448
modulus operator. *See* %
monitor, 481–482
mouse listener, 642
multi-line comment, 97
multidimensional arrays, 261–262
multiple exceptions, catching,
 280–281
multiple return statements, 173
multiplication, bitwise, 362
multithreading, 109, 462
 atomic variables, 501
 atomic wrappers, 506
 `BlockingQueue` interface,
 483–484
 cooperative/preemptive multi-
 tasking, 477
 deadlocks, 495
 groups, 505
 objects, 492–494
 priority levels, 494
 `Runnable` interface, 480
 shutting down, 502–504
 stopping, 485–486
 synchronization, 478–483, 506
 `ThreadLocal` class, 495–498
 `Timer` class, 499–500

mutual exclusion, 481
MySQL, 665, 667

N

naked type variable, 513–514
namespaces, 78–81
naming conventions, 59–61
NaN, 357–358
natural sort order, 169
navigable association, 16
nested class, 373, 412–413
networking, 696
new operator, 39–40, 42
NIO, 696–697
non-short-circuited logical operators, 313
not operator (!), 312
notify (Object method), 491–492
notifyAll (Object method), 489, 491
NotSerializableException, 402
NullPointerException, 82, 159, 257, 327, 357, 599
NumberFormat (java.text), 681
numbers, 66
 BigDecimal class, 350–353
 characters, 103–105
 dates, 86–91
 floating-point, 173–174
 random, 371–374
 sorting, 171–172
 strings, 370
numeric casting, 355–356
numeric literal, 65
numeric overflow, 359
numeric wrapper classes, 370–371
numerics, primitive numeric types, 353–356

O

Object, 73–74
Object Mentor, 1
object streams, 399–407
object-oriented, 2, 12, 13–17
objects, mock, 425–432
ObjectInputStream (java.io), 400, 416
ObjectOutputStream (java.io), 400, 416, 419
octal, 105, 354
one-to-many, 75, 294
open-closed principle, 182–183, 186, 450
operating system, 11
operators, 107
 and (&&), 312
 bit-and (&), 362
 bit-or (||), 362
 exclusive-or (^), 312–313
 increment, 144
 instanceof, 328
 logical, 311–313
 logical negation (~), 362
 new, 39
 not (!), 312
 or (||), 312
 postfix, 145
 prefix, 144
 ternary, 245
 xor (^), 362
or operator (||), 312
OutOfMemoryError, 276
OutputStream (java.io), 389
OutputStreamWriter (java.io), 389–390
OutputStreamWriter (java.io), 394
overflow, numeric, 359–360
overloaded constructor, 86–87

overloaded operator, 107
overriding, 216

P

package (keyword), 72, 78–81
package access, 122–123
package import, 93
package structure, 121
packages
 java.awt, 572
 java.io, 379–380, 389
 java.lang.instrument, 695
 java.lang.management, 696
 java.lang.ref, 694
 java.net, 696
 java.security, 698
 java.util, 73
 java.util.concurrent, 483
 java.util.concurrent.atomic, 506
pair programming, 72
parameter, 38, 56–57
parameterized types, 71–72, 74–75,
 509–510
 additional bounds, 525–526
 arrays, 530
 checked collections, 528–530
 Collections Framework,
 510–511
 creating, 511–513
 generic methods, 542
 limitations, 531
 raw types, 526–527
 reflection, 532
 specifying, 509
 support, 514–515
 upper bound constraints,
 516–518
 wildcards, 518–524

parameters
 annotations, 555–556
 arrays, 553–554
 interface references, 187–189
parity checking, 365
parseInt (Integer method), 370–371
path, 24
Pattern (java.util.regex), 657,
 661–662
performance testing, 336–337
PermissionException (java.lang.re-
flect), 454, 457
persistence, 664
phantom references, 694
pi, 368
piped streams, 399
platform, 11
Point (java.awt), 617
polymorphism, 14, 181–186, 199,
 220
postcondition, 220
postfix, 145
precedence rules, 715
preemptive multitasking, 477
Preference (java.util.prefs), 688
preferences, 685–689
prefix operator, 144–145
PreparedStatement (java.sql),
 670–672
primitive types, 66
 casting, 251
 field initialization, 159–160
 wrapper classes, 252–255
printStackTrace (Throwable
 method), 285
PrintStream (java.io), 389, 394
PrintWriter (java.io), 387
priority levels, threads, 494
private keyword, 57–59, 122
private constructor, 135

Process, 691
 process framework, 10
 process instance, 10
 ProcessBuilder, 689, 691–693
 processes, creating, 689
 profiling, Java, 345
 program, 11
 programmer’s editor, 21–22
 programming, 11
 programming by intention, 213–214, 292
 project, Ant, 129
 Properties (java.util), 683
 properties, 667, 683
 Ant, 130
 system, 109–110
 property files, 685
 property, setting on command line, 684
 protected keyword, 213–215
 proxy, 448–449, 697
 public keyword, 33, 122–124, 184
 public interface, 66
 pushback streams (java.io), 399

Q

query method, 133
 Queue (java.util), 484
 queue, 462, 474, 483–484

R

Random (java.util), 371–374
 random access file, 407
 random numbers, 371–375
 RandomAccessFile (java.util), 415–416
 raw types, 527

Reader (java.io), 390
 ReadWriteLock (java.util.concurrent.locks), 492
 realizes association, UML, 183
 receiver, 14, 44
 recursion, 229–230, 242
 red bar, 36–37
 ReentrantLock (java.util.concurrent.locks), 493
 refactoring, 53, 94–95. *See also* duplication
 references, 42
 interface, 187–189
 reflection, 284, 440–459, 532, 542, 667
 regression test suite, 301
 regular expressions, 266, 656–662
 relational database, 664
 remote method invocation. *See* RMI
 replaceAll (String method), 659
 required fields, 615
 requirements. *See* stories
 resource bundle, 85, 674–676
 ResourceBundle (java.util), 674, 676, 678, 679
 ResultSet (java.sql), 670, 672
 ResultSetMetaData (java.sql), 672
 rethrowing exceptions, 282–283
 return statement, 45–46, 173, 178, 287
 return type, 38, 45
 annotations, 558–559
 return within finally, 287
 RMI, 400, 449, 697–698
 Robot (java.awt), 617
 rounding, BigDecimal, 352
 RuleBasedCollator (java.text), 681
 run (Thread method), 473, 477, 485, 490, 504, 649
 Runnable, 473, 480, 659

Runtime, 503, 689

RuntimeException, 276

RUP, 9, 10

S

SAAJ, 699

SBCS. *See* single-byte character set

scaling (BigDecimal class), 352

scope, static 137

Scrum, 9

SDK, 11, 23

SecureRandom (java.util), 374

sending messages, 14

sequence diagram, 476

SequenceInputStream (java.io),
399

Serializable (java.io), 401, 402

serialization, 399–407

serialver command, 404

serialVersionUID, 404

Set (java.util), 339

setDaemon (Thread method),
503

setProperty (System method),
694

setUp (junit.framework.TestCase
method), 81–82

shallow clone, 664

Short, 252

short type, 354

short-circuited logical operators,
313

shutting down threads, 502–504

signal (java.util.concurrent.locks.
Condition method), 493

signalAll (java.util.concurrent.locks.
Condition method), 493

signature, 166

simple design, 147, 577

SimpleFormatter (java.util.logging),
299–300

Single Responsibility Principle, 60,
112, 115, 450, 463, 509–510,
525, 572

single-byte character set, 105

single-line comment, 96

small steps, 5

soft references, 694

software development kit, 11

sorting, 163–172, 263

source file, 12

special characters, 105

split (String method), 265–266, 347,
657

SpringLayout (javax.swing), 606

spurious wakeup, 491

SQL, 664–665, 668–671, 673

SQLException, 666

SRP. *See* Single Responsibility Prin-
ciple

Stack (java.util), 246

stack trace, 48, 159, 285

stack walkback, 159

start (Thread method), 477

state, 67, 133–134

Statement (java.sql), 670

statement, 39

static import, 141–143

static initialization block, 136

static keyword, 109., 135, 137

static method, 71, 127

static nested class, 412–413

static scope, 137

statics, use of, 147–150

stderr, 118, 680, 691

stdin, 389, 690

stdout, 118, 389, 690, 691

stereotype, 136

stop (Thread method), 485

story, 33

strategy, 183–184
 stream, 379–380
 stream unique identifier. *See serial VersionUID*
StreamTokenizer (`java.io`), 399
StrictMath, 368
String, 39, 168
 string concatenation, 106
 string constant, 54
 string literal, 39, 54
StringBuffer, 109
StringBuilder, 107–109
 strings, 103
 concatenation, 106–107
 literals, 39
 splitting, 264–266
StringTokenizer, 264
 strongly typed, 74
 student information system, 6, 31
 student information system stories, 32
 style
 naming conventions, 60
 whitespace, 61
 subclass, 33, 90, 202
 subcontracting, 220–228, 537
 subdirectories
 classes, 81
 creating, 121
 subscripting, array, 257
 suite, 70–71
 Sun, 28
 super keyword, 207, 212
 superclass, 203
 Swing, 3, 5, 35, 504, 565–567
 Swing layout, 594
SwingUtilities (`javax.swing`), 649
 switch statement, 195–199
 synchronization, 478–480, 506
 synchronization wrappers, 482–483

synchronized keyword, 481–482, 488–489, 502
System, 109, 389–390, 503, 689
 system environment, 689
 system properties, 109
System.err, 118
System.out, 118, 119–120

T

tags. *See annotations*
 target, Ant build, 129
 TDD. *See test-driven development*
 TDTT (test-driven truth table), 312
tearDown (`junit.framework.TestCase` method), 485
 temp variable. *See local variable*
 template method, 217
 temporary variable. *See local variable*
 ternary operator, 245–246
 test class, 32
 test package, 126
 test suite, 70–71
 test-driven development, 1–2, 4–5, 11, 18–19, 31–32, 154–157, 651–652
 test-driven development cycle, 55
TestCase (`junit.framework`), 33–34, 46–47, 421
TestRunner (`junit.awtui`, `junit.swingui`, `junit.textui`), 35–36
 tests as documentation, 154–155
TextPad, 21, 656
 this keyword, 55–57, 170
 Thread, 472, 473
 thread groups, 505–506

thread pool, 491–492, 496
 thread priorities, 494
 ThreadLocal, 495–499
 throw statement, 278
 Throwable, 276
 throws clause, 274–275
 Timer (`java.util`), 500–501
 TimerTask (`java.util`), 500
 timestamps, 86–91, 95
 title bar, 609–610
 tokens, 264
 tolerance, float, 175
`toString` (`Object` method), 107,
 343–345, 590
 trace statement, 119
 transient (keyword), 402–403,
 405
 trapping exceptions, 271
 TreeMap (`java.util`), 342–343
 TreeSet (`java.util`), 342–343
 trim (`String` method), 243
 try (keyword), 271
 try-catch block, 271–272
 two's complement, 361
 type parameter list, 511

U

UML, 16–17
 abstract test, 223
 activation, 476
 activity diagram, 476
 class box, 15
 class dependency, 16
 composition, 445
 inheritance, 18, 34, 204
 interface, 167, 184, 211
 message send, 14
 multiplicity, 294

one-to-many relationship, 75
 protected methods, 217
 realizes, 183
 sequence diagram, 475–476
 stereotype, 136
 strategy pattern, 183
 UncaughtExceptionHandler, 505,
 506
 unchecked exception, 274, 276,
 278–279
 unchecked, javac lint switch, 527
 Unicode, 103–105
 Unified Modeling Language. *See*
 UML
 unit test, 18–19
 Unix, 11, 24, 40–41, 50, 109, 299,
 477, 689
 unsigned bit shift right operator.
 See >>>
 upper bounds, 515, 516–518,
 519
 upper camel case, 60
 URL (`java.net`), 273, 465
 URL, 273, 612, 696
 URL, database, 667
 URLConnection (`java.net`), 465
 user interface, 121
 user thread, 502, 504
 utility class, 136
 utility method, 134

V

varargs, 260–261
 variables
 atomic, 501
 boolean, 150–154
 classes, 137–143
 enums, 209

- variables (*cont.*)
 - incrementing, 67
 - instance, 49–52, 187–189
 - local, 42–43
 - naked type, 513, 531
 - numeric overflow, 359
 - settings, 364
 - statics
 - applying, 148
 - rules, 149–150
 - troubleshooting, 147
 - switch statements, 196
 - Vector (java.util), 246–247, 482
 - view, 571
 - virtual machine. *See* VM
 - VM, 11, 23
 - void keyword, 38, 45, 68
 - volatile keyword, 502
- W**
- wait (Object method), 489, 491–492
 - wait cursor, 647–648
 - wait/notify, 472, 486–492
 - walkback, 48
 - warnings, deprecation, 91–92
 - waterfall, 9
 - weak references, 694
 - WeakHashMap (java.util), 694
- X**
- XML, 319, 719. *See also* Ant
 - XMLFormatter (java.util.logging), 300
 - xor (^) operator, 312–313, 362–365
 - XP, 9, 10
- Y**
- yield (Thread method), 475, 477
- Z**
- ZIP file, 654

This page intentionally left blank

Wouldn't it be great

if the world's leading technical publishers joined forces to deliver their best tech books in a common digital reference platform?

They have. Introducing
InformIT Online Books
powered by Safari.

■ Specific answers to specific questions.

InformIT Online Books' powerful search engine gives you relevance-ranked results in a matter of seconds.

■ Immediate results.

With InformIT Online Books, you can select the book you want and view the chapter or section you need immediately.

■ Cut, paste and annotate.

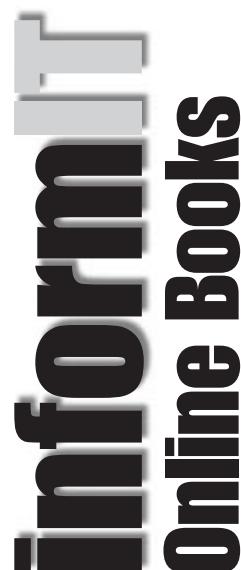
Paste code to save time and eliminate typographical errors. Make notes on the material you find useful and choose whether or not to share them with your work group.

■ Customized for your enterprise.

Customize a library for you, your department or your entire organization. You only pay for what you need.

Get your first 14 days FREE!

For a limited time, InformIT Online Books is offering its members a 10 book subscription risk-free for 14 days. Visit <http://www.informit.com/online-books> for details.



informit.com/onlinebooks



informIT

www.informit.com

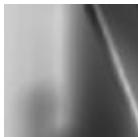
YOUR GUIDE TO IT REFERENCE

Licensed by



Articles

Keep your edge with thousands of free articles, in-depth features, interviews, and IT reference recommendations – all written by experts you know and trust.



Online Books

Answers in an instant from **InformIT Online Book's** 600+ fully searchable on line books. For a limited time, you can get your first 14 days **free**.



Catalog

Review online sample chapters, author biographies and customer rankings and choose exactly the right book from a selection of over 5,000 titles.

