

Java® Programming

24-Hour Trainer

Yakov Fain

EXHIBIT
1005

JAVA® PROGRAMMING 24-HOUR TRAINER

INTRODUCTION xxix

LESSON 1	Introducing Java	1
LESSON 2	Eclipse IDE	9
LESSON 3	Object-Oriented Programming	17
LESSON 4	Class Methods	31
LESSON 5	Back to Java Basics	41
LESSON 6	Packages, Interfaces, and Encapsulation	55
LESSON 7	Programming with Abstract Classes and Interfaces	65
LESSON 8	Introducing the Graphic User Interface	75
LESSON 9	Event Handling in UI	89
LESSON 10	Introduction to Java Applets	103
LESSON 11	Developing a Tic-Tac-Toe Applet	111
LESSON 12	Developing a Ping-Pong Game	123
LESSON 13	Error Handling	133
LESSON 14	Introduction to Collections	143
LESSON 15	Introduction to Generics	155
LESSON 16	Working with Streams	163
LESSON 17	Java Serialization	175
LESSON 18	Network Programming	185
LESSON 19	Processing E-Mails with Java	199
LESSON 20	Introduction to Multi-Threading	209
LESSON 21	Digging Deeper into Concurrent Execution	223
LESSON 22	Working with Databases Using JDBC	235
LESSON 23	Swing with JTable	251
LESSON 24	Annotations and Reflection	261

Continues

LESSON 25 Remote Method Invocation 273
LESSON 26 Java EE 6 Overview 283
LESSON 27 Programming with Servlets 293
LESSON 28 JavaServer Pages 309
LESSON 29 Developing Web Applications with JSF 323
LESSON 30 Introducing JMS and MOM 335
LESSON 31 Introducing JNDI 349
LESSON 32 Introduction to Enterprise JavaBeans 361
LESSON 33 Introduction to the Java Persistence API 375
LESSON 34 Working with RESTful Web Services 387
LESSON 35 Introduction to Spring MVC Framework 399
LESSON 36 Introduction to Hibernate Framework 411
LESSON 37 Bringing JavaFX to the Mix 421
LESSON 38 Java Technical Interviews 435
APPENDIX What's on the DVD? 447

INDEX 451

Java® Programming

24-HOUR TRAINER

Java® Programming

24-HOUR TRAINER

Yakov Fain



Wiley Publishing, Inc.

Java® Programming 24-Hour Trainer

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2011 by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-0-470-88964-0
ISBN: 978-1-118-05817-6 (ebk)
ISBN: 978-1-118-05818-3 (ebk)
ISBN: 978-1-118-05819-0 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

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

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

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

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

Library of Congress Control Number: 2010942181

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

In memory of my friend, Felix Rubinchik.

ABOUT THE AUTHOR



YAKOV FAIN works as a software architect for Farata Systems, a company that provides consulting services in the field of development of enterprise applications. He has authored several technical books and lots of articles on software development. Sun Microsystems has awarded Mr. Fain with the title of Java Champion, which has been given to only 150 people in the world. He leads the Princeton Java Users Group. Yakov's Twitter ID is @yfain.

ABOUT THE TECHNICAL EDITOR

ELLIOTTE RUSTY HAROLD is originally from New Orleans, to which he returns periodically in search of a decent bowl of gumbo. However, he currently resides in the Prospect Heights neighborhood of Brooklyn with his wife Beth, dog Thor, and cat Marjorie. He is the author of numerous books including *Refactoring HTML*, *Java Network Programming*, *Java I/O*, and *Processing XML with Java*. His open source projects include the XOM Library for processing XML with Java and the Amateur media player.

CREDITS

EXECUTIVE EDITOR
Robert Elliott

PROJECT EDITOR
Charlotte Kughen, The Wordsmithery LLC

TECHNICAL EDITOR
Elliotte Rusty Harold

PRODUCTION EDITOR
Kathleen Wisor

COPY EDITOR
Sadie Kleinman

EDITORIAL DIRECTOR
Robyn B. Siesky

EDITORIAL MANAGER
Mary Beth Wakefield

FREELANCER EDITORIAL MANAGER
Rosemarie Graham

MARKETING MANAGER
Ashley Zurcher

PRODUCTION MANAGER
Tim Tate

VICE PRESIDENT AND EXECUTIVE GROUP PUBLISHER
Richard Swadley

VICE PRESIDENT AND EXECUTIVE PUBLISHER
Barry Pruett

ASSOCIATE PUBLISHER
Jim Minatel

PROJECT COORDINATOR, COVER
Katie Crocker

COMPOSITOR
Jeff Lytle, Happenstance Type-O-Rama

PROOFREADERS
Nancy Carrasco
Jen Larsen, Word One New York

INDEXER
Robert Swanson

COVER DESIGNER
Michael E. Trent

COVER IMAGE
© FotografiaBasica / iStockPhoto.com

ACKNOWLEDGMENTS

FIRST OF ALL I want to thank my family for understanding that stealing time from family to write a computer book is OK.

I'd also like to thank Elliotte Rusty Harold from Google for his valuable comments during a technical edit of this book. Elliotte himself has authored more than 20 computer books (see `http://www.elharo.com`) and is respected in the Java community around the world. His comments on my lessons ranged from the philosophical to a simple "I don't agree with this solution." I have addressed all of them.

Three people volunteered to help me with the book: Nick Golubyev (Singapore), Dmitry Danileyko (Russia), and Viktor Gamov (USA). They wrote several code samples for the book, which really saved me time! Thank you, guys!

Big thanks to the Wiley editors for doing a great job of editing and for not cursing me for not meeting deadlines.

CONTENTS

INTRODUCTION *xxix*

LESSON 1: INTRODUCING JAVA 1

Why Learn Java? 1
Setting the Goals 2
The Life Cycle of a Java Program 2
JDK and JRE 3
Java SE and EE 3
Downloading and Installing JDK in Windows 3
Your First Java Program: Hello World 4
Compiling and Running Hello World 6
Try It 7
Lesson Requirements 7
Step-by-Step 7

LESSON 2: ECLIPSE IDE 9

Introducing Eclipse IDE 9
Downloading and Installing Eclipse 10
Creating Hello Project in Eclipse 11
Creating the HelloWorld Class in Eclipse 13
Java Packages 13
Completing Code Generation 14
Try It 15
Lesson Requirements 15
Step-by-Step 15

LESSON 3: OBJECT-ORIENTED PROGRAMMING 17

Classes and Objects 17
Variables, Constants, and Data Types 19
Declaring Variables 19
Constants 19
Primitive Data Types 19
Variable Scope 20
Wrappers, Autoboxing, and Unboxing 21

Program Comments **21**
First Useful Program 22
Conditional Statement if 25
switch Statement 26
Inheritance 26
Method Overriding 27
Try It **28**
Lesson Requirements 28
Hints 28
Step-by-Step 28

LESSON 4: CLASS METHODS **31**

Method Arguments **31**
Method Overloading **31**
Constructors **32**
The Keyword super **34**
The Keyword this **34**
Passing by Value or by Reference **36**
Variable Scopes **37**
The Keyword static **37**
Try It **38**
Lesson Requirements 38
Step-by-Step 39

LESSON 5: BACK TO JAVA BASICS **41**

Arrays **41**
Loops **42**
Debugging Java Programs **46**
More about if and switch Statements **48**
The Flavors of if Statements 49
The switch Statement 50
Command-Line Arguments **51**
Try It **52**
Lesson Requirements 52
Step-by-Step 52

LESSON 6: PACKAGES, INTERFACES, AND ENCAPSULATION **55**

Java Packages **55**
Encapsulation **56**
Access Levels 57

Page 18 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

The Keyword final **57**
final Variables 57
final Methods 58
final Classes 58
Interfaces **58**
Marker Interfaces 60
Casting **60**
Try It **62**
Lesson Requirements 62
Step-by-Step 62

LESSON 7: PROGRAMMING WITH ABSTRACT CLASSES AND INTERFACES **65**

Abstract Classes **65**
Assignment 65
Solution with an Abstract Class 66
Polymorphism **68**
Making the Interface Solution Polymorphic 69
Interfaces versus Abstract Classes **70**
Try It **71**
Lesson Requirements 71
Step-by-Step 71

LESSON 8: INTRODUCING THE GRAPHIC USER INTERFACE **75**

Swing Basics **75**
Layout Managers **77**
A Simple Calculator with FlowLayout 77
A Brief Introduction to Layout Managers 79
FlowLayout 79
GridLayout 79
BorderLayout 81
Combining Layout Managers 81
BoxLayout 84
GridBagLayout 84
CardLayout 86
Containers with Absolute Layout 86
More about Swing Widgets 86
Swing GUI Builders **87**
Try It **87**
Lesson Requirements 87
Step-by-Step 88

Page 19 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

LESSON 9: EVENT HANDLING IN UI 89

Introduction to Event Listeners 89
Teaching the Calculator to Calculate 90
Registering Components with ActionListener 91
Finding the Source of an Event 91
How to Pass Data between Objects 93
More Swing Listeners 96
How to Use Adapters 97
Inner Classes 98
Anonymous Inner Classes 99
Closures 100
Try It 100
Lesson Requirements 100
Step-by-Step 101

LESSON 10: INTRODUCTION TO JAVA APPLETS 103

An Unofficial History of Java Applets 103
Restrictions of Java Applets 104
Learning HTML on the Run 105
Writing Applets Using Swing 107
Try It 109
Lesson Requirements 109
Step-by-Step 109

LESSON 11: DEVELOPING A TIC-TAC-TOE APPLET 111

The Strategy 111
Coding Tic-Tac-Toe 112
Try It 121
Lesson Requirements 121
Step-by-Step 121

LESSON 12: DEVELOPING A PING-PONG GAME 123

The Strategy 123
The Code 124
The User Interface 124
The Constants 127
The Engine 128
Finishing the Ping-Pong Game 130

Page 20 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

Try It **131**
Lesson Requirements 132
Hints 132
Step-by-Step 132

LESSON 13: ERROR HANDLING **133**

Stack Trace **133**
Java Exceptions **134**
Exception Hierarchy **135**
try/catch Blocks **136**
The throws Clause **137**
The finally Clause **137**
The throw Operator **138**
Creating Your Own Exceptions **139**
Try It **141**
Lesson Requirements 141
Step-by-Step 142

LESSON 14: INTRODUCTION TO COLLECTIONS **143**

Arrays Revisited **143**
Classes ArrayList and Vector **144**
Collection Interfaces from java.util **146**
Classes Hashtable and HashMap **147**
Class Properties 148
Enumeration and Iterator **149**
Class LinkedList **150**
Class BitSet **151**
Try It **152**
Lesson Requirements 152
Step-by-Step 153

LESSON 15: INTRODUCTION TO GENERICS **155**

Generics with Classes **155**
Defining Generics **157**
Wild Cards **158**
Bounded Wild Cards **159**
Generic Methods **160**
What to Read Next **161**
Try It **161**
Lesson Requirements 161
Step-by-Step 162

Page 21 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

LESSON 16: WORKING WITH STREAMS 163

Byte Streams 164
Buffered Streams 165
Character Streams 166
Data Streams 171
The Class File 171
Try It 172
Lesson Requirements 172
Step-by-Step 172

LESSON 17: JAVA SERIALIZATION 175

The Class ObjectOutputStream 176
The Class ObjectInputStream 178
The Interface Externalizable 179
Class Versioning 182
Serializing into Byte Arrays 182
Try It 183
Lesson Requirements 183
Step-by-Step 184

LESSON 18: NETWORK PROGRAMMING 185

Reading Data from the Internet 186
Connecting through HTTP Proxy Servers 189
How to Download Files from the Internet 189
The Stock Quote Program 191
Socket Programming 193
Why Use Sockets? 194
The Stock Quote Server with Sockets 194
Try It 197
Lesson Requirements 197
Hints 197
Step-by-Step 198

LESSON 19: PROCESSING E-MAILS WITH JAVA 199

Protocols and Servers 199
Creating Birthday Mailer 200
Required Supporting Classes 200
Writing the Mail Sender 201
How to Retrieve E-Mails 204

Page 22 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

Useful Open-Source Components 205
Try It 206
Lesson Requirements 206
Step-by-Step 206

LESSON 20: INTRODUCTION TO MULTI-THREADING 209

The Class Thread 210
The Interface Runnable 211
Sleeping Threads 212
How to Kill a Thread 214
Thread Priorities 216
Thread Synchronization and Race Conditions 216
Thread States 219
Wait and Notify 219
Try It 222
Lesson Requirements 222
Step-by-Step 222

LESSON 21: DIGGING DEEPER INTO CONCURRENT EXECUTION 223

Joining Threads 223
Goodies from java.util.concurrent 224
ReentrantLock versus Synchronized 225
Executor Framework 225
A Brief Introduction to Concurrent Collections 229
SwingWorker Thread 230
Try It 233
Lesson Requirements 233
Step-by-Step 233

LESSON 22: WORKING WITH DATABASES USING JDBC 235

JDBC Driver Types 236
Creating a Database with Derby 236
Sample JDBC Program 238
Processing Result Sets 241
The PreparedStatement Class 243
The CallableStatement Class 244
The ResultSetMetaData Class 244
Scrollable Result Sets and RowSet 246
Transactional Updates 247
Connection Pools and DataSources 248

Page 23 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

Try It **248**
Lesson Requirements 248
Hint 248
Step-by-Step 249

LESSON 23: SWING WITH JTABLE **251**

JTable and the MVC Paradigm **251**
The Model **252**
Mandatory Callbacks of Table Models 253
Optional Callbacks of Table Models 256
Introduction to Renderers **257**
Summary **259**
Try It **259**
Lesson Requirements 259
Hint 259
Step-by-Step 260

LESSON 24: ANNOTATIONS AND REFLECTION **261**

Java Annotations Basics **262**
@Override 263
@Deprecated 264
@Inherited 264
@Documented 264
Custom Annotations **264**
Reflection **267**
Run-Time Annotation Processing **270**
Summary **271**
Try It **272**
Lesson Requirements 272
Step-by-Step 272

LESSON 25: REMOTE METHOD INVOCATION **273**

Developing Applications with RMI **274**
Defining Remote Interfaces 274
Implementing Remote Interfaces 275
Registering Remote Objects 276
Writing RMI Clients 277
Security Considerations 278
Finding Remote Objects 279

Page 24 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

Try It **279**
Lesson Requirements 280
Hints 280
Step-by-Step 280

LESSON 26: JAVA EE 6 OVERVIEW **283**

The Big Picture **284**
JCP, JSR, and Other Acronyms 284
Tiers of Java EE Applications 285
Containers versus Application Servers 287
Profiles and Pruning **288**
Miscellaneous Concepts **288**
Try It **289**
Lesson Requirements 289
Hints 289
Step-by-Step 289

LESSON 27: PROGRAMMING WITH SERVLETS **293**

The Big Picture **293**
The Thin Client **294**
How to Write a Servlet **295**
How to Deploy a Servlet **296**
Installing the GlassFish Plug-In for Eclipse **297**
How to Create a Servlet with Eclipse **298**
Browser-Servlet Data Flow **301**
HTTP Get and Post Requests **301**
Session Tracking **302**
Cookies 302
URL Rewriting 303
Server-Side HttpSession 303
Filters 304
Event Listeners **305**
Asynchronous Servlets **306**
Try It **307**
Lesson Requirements 307
Hints 307
Step-by-Step 307

LESSON 28: JAVASERVER PAGES **309**

Embedding Java Code into HTML **310**
Implicit JSP Objects **311**

Page 25 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

Overview of the JSP Tags 312
Directives 312
Declarations 312
Expressions 313
Scriptlets 313
Comments 313
Standard Actions 313
Error Pages 314
JavaBeans 315
Using JavaBeans in JSP 316
How Long Does a Bean Live? 316
Loading JSP from Servlets 317
Tag Libraries 318
JSTL 320
Try It 320
Lesson Requirements 321
Step-by-Step 321

LESSON 29: DEVELOPING WEB APPLICATIONS WITH JSF 323

The Big Picture 324
Managed Beans 324
Creating a JSF Website 326
Templating 326
Page Navigation 331
Converters and Validators 333
Try It 334
Lesson Requirements 334
Hints 334
Step-by-Step 334

LESSON 30: INTRODUCING JMS AND MOM 335

Messaging Concepts and Terminology 335
Two Modes of Message Delivery 337
JMS API Overview 337
Types of Messages 338
How to Send a Message 338
How to Receive a Message 340
How to Publish a Message 343
How to Subscribe for a Topic 343
Message Selectors 344

Page 26 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

Administering Objects in Open MQ 345
Try It 346
Lesson Requirements 346
Hints 347
Step-by-Step 347

LESSON 31: INTRODUCING JNDI 349

Java Naming and Directory Interface 349
Administering JNDI Objects in GlassFish 350
Accessing the GlassFish Naming Service with JNDI 352
Injection of JNDI Resources 355
DataSource and JNDI 355
Lightweight Directory Access Protocol 357
Try It 358
Lesson Requirements 358
Step-by-Step 359

LESSON 32: INTRODUCTION TO ENTERPRISE JAVABEANS 361

Who Needs EJB Containers? 361
Types of EJBs 362
Stateless Session Beans 363
The Bean 363
The Client's View 363
The Server 366
Asynchronous Methods 367
Stateful Session Beans 368
Singleton Beans 369
Deploying EJB 370
Message-Driven Beans 371
Timer Service 372
Summary 372
Try It 372
Lesson Requirements 373
Hint 373
Step-by-Step 373

LESSON 33: INTRODUCTION TO THE JAVA PERSISTENCE API 375

The Big Picture 375
Mapping Objects to Database Tables 376
JPQL 378

Page 27 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

EntityManager 379
A Brief Introduction to the Criteria API 380
Bean Validation 382
Try It 382
Lesson Requirements 382
Step-by-Step 382

LESSON 34: WORKING WITH RESTFUL WEB SERVICES 387

The SOAP Web Services 387
The RESTful Web Services 388
The RESTful Stock Server 388
Try It 397
Lesson Requirements 397
Hints 397
Step-by-Step 397

LESSON 35: INTRODUCTION TO SPRING MVC FRAMEWORK 399

Bean Wiring 400
Overview of Spring MVC 400
Processing HTML with Spring 402
Understanding the Spring Web MVC Workflow 402
Processing HTML Forms 406
Try It 409
Lesson Requirements 409
Step-by-Step 409

LESSON 36: INTRODUCTION TO HIBERNATE FRAMEWORK 411

The Big Picture 411
Installing and Configuring Hibernate 412
Adding Hibernate Jars to an Eclipse Project 412
Testing the Database Connection 413
Configuring Hibernate with a DBMS 413
Retrieving Data with Hibernate 415
Try It 417
Lesson Requirements 417
Hints 417
Step-by-Step 418

Page 28 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

LESSON 37: BRINGING JAVAFX TO THE MIX 421

Consuming Stock Quotes with JavaFX 421
Code Walkthrough 422
Reading JavaFX Code 425
The HTML Client 429
Try It 431
Lesson Requirements 432
Hint 432
Step-by-Step 432

LESSON 38: JAVA TECHNICAL INTERVIEWS 435

Getting the Interview 435
Doing Well at the Interview 436
Considering the Offer 437
Interviewing Enterprise Developers 438
To Get or Not to Get Certified? 439
Technical Questions and Answers 439
Epilogue 446

APPENDIX: WHAT'S ON THE DVD? 447

Using the DVD on a PC 447
Using the DVD on a Mac 448
What's on the DVD? 448
Troubleshooting 448
Customer Care 449

INDEX *451*

Page 29 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

INTRODUCTION

IT DOESN'T TAKE a rocket scientist to notice the recent serious decline in the publishing of computer books, to say nothing of computer magazines. I enjoy reading books about software and visit the nearby Barnes and Noble at least twice a month. During the last three years the number of bookcases dedicated to computer books has shrunk substantially.

Today newly baked programmers prefer Google to books. They don't realize that Google is OK when you know what you are looking for, while books can give you a new perspective on how other people develop software.

During a recent visit to a major bookstore I noticed a series of books on digital photography. I'm one of the legions of people who are thinking of replacing a point-and-shoot camera with a digital SLR such as a Nikon D90. There were about 10 different thin books on that shelf, and I picked the one on the D90. It was about 65 pages long. I quickly skimmed the pages, learning that compared to a pocket camera one of the main advantages of D-SLR is the ability it gives you to change lenses. On top of that, these cameras use a small mirror that enables you to see exactly what you're shooting, plus you can take more shots per second. Not much information for a $25 book.

The secret was that the book came with a DVD, which contained 80 minutes of instructional videos! The printed part of the book and the DVD complemented each other. The wannabe photographers don't want to read books.

The new generation of computer programmers doesn't want to read either. They want to watch videos. They can be YouTube videos on how to do something, or screencasts. The material has to be prepared for easy consumption.

But because not everything in the world of software can be videotaped, future computer books will still include several hundred pages — some code samples and short instructions will go there.

Remember those 1,500-page Bibles on software? It seems that their days are over. These days a 500-page computer book can be considered thick. Five years from now a 200-page software book will become the standard. But books will come with DVDs on which the authors walk you through all the examples mentioned on paper.

I like this new 24-Hour Trainer series from Wiley Publishing. This is not to say that you can learn the software covered in these books within 24 hours. It's about having a trainer that's with you 24 hours a day. Each book in this series is a set of short chapters containing a minimum of theory but accompanied by an instructional DVD with screencasts recorded by the author.

WHO THIS BOOK IS FOR

This book is for anyone who wants to learn how to program with the Java language. No previous programming experience is expected.

This tutorial can be used by Java developers looking for simple working examples that use certain features of the language.

Accomplished Java developers can also use this book as a refresher while preparing for a technical job interview.

This tutorial can be used by university students who are interested in learning from a practitioner who has spent 20-plus years developing enterprise software for a living.

College professors should appreciate the fact that each lesson ends with a "Try It" section — a prepared home assignment for each lesson.

WHAT THIS BOOK COVERS

To be called a Java developer, a person has to know not only the core syntax of this programming language, but also the set of server-side technologies called Java EE (Enterprise Edition). This book covers both. At the time of this writing, the latest version of core Java is 6, and the latest release of Java EE is also 6.

Java is a general-purpose language — you can program applets running under control of a Web browser, stand-alone applications that run independently on the user's computer, and applications that connect to remote servers. Java is used for writing applications running in mobile devices, cars, or in household appliances.

The first 25 lessons of this book cover a program that can run on users' computers. The next 12 are about Java EE technologies used for Java programs that run on servers. The final lesson is dedicated to the process of getting prepared for Java technical job interviews.

This book doesn't cover Java programming for mobile devices.

HOW THIS BOOK IS STRUCTURED

This book is a tutorial. Each lesson walks you through how to use certain elements and techniques of the Java language or gives an introduction to the server-side Java EE technologies. The "Try It" sections serve as continuations of materials explained in the lessons. The screencasts included on the DVD usually illustrate how the author compiles and runs code samples included in the lessons. You can choose to read the lesson and then try to run the example on your own, or you can read the lesson, watch the video, and then try to run the examples. After this part is done, start working on the assignments from the "Try It" section.

This book contains 38 lessons, which are not formally broken into sections. The first 25 lessons cover Java SE topics. Lessons 26 through 34 cover Java EE topics. Lessons 35 and 36 introduce you to popular third-party frameworks Spring and Hibernate. Lesson 37 demonstrates an application that utilizes several Java EE technologies. And, finally, Lesson 38 helps you to get prepared for the technical interview when you decide to look for a Java-related job.

Page 32 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

The lessons are short and to the point. The goal is to explain the material quickly so you can start applying it hands-on as soon as possible. Some readers may feel that more explanation of certain subjects are required, and they are encouraged to do some extra research; there are lots and lots of online materials available on any Java-related subject. But the coverage of the material given in this book definitely helps in understanding what to focus on and what to look for.

WHAT YOU NEED TO USE THIS BOOK

To do the examples in this book, you do not need to purchase any software—freely available software is used here. Installing Java Development Kit and Eclipse IDE is explained in the first two lessons, and this is all you need for Lessons 1 through 25. In Lesson 26, you install Java Application Server GlassFish, which is used for explanation of the materials in Lessons 26 through 34. If you decide to learn third-party frameworks Spring and Hibernate, you need to download and configure their software libraries as explained in Lessons 34 and 35. Whenever you need to download certain software, detailed instructions are given in the book and/or in the screencasts included on the DVD.

From the hardware perspective, you can use either PC running Windows or one of the Apple computers running Mac OS. Linux fans are also able to run all book samples. You should have at least 1GB of RAM on your computer to run all code examples from this book. But adding more memory can make your Java compiler and Eclipse IDE work faster.

HOW TO READ THIS BOOK

This book is a tutorial, and I assume in each lesson that the reader is already familiar with the materials from those that came before it. If you are new to Java, I highly recommend that you read this book sequentially.

Each lesson except the last one has a corresponding video screencast on the accompanying DVD, in which I either work on the assignment from the "Try It" section of the lesson or simply install and configure some software. Ideally, you should try to do all the assignments from the "Try It" sections on your own, and use the videos only if you get stuck or don't understand the instructions. But if you prefer to learn by following the instructor, just watch the video first and then try to repeat the same actions on your own. Whatever works is fine.

Java is a multi-platform language and programs written for Microsoft Windows, say, should work the same way in Mac OS or on Linux computers. I'm using a Mac, but also have special software that enables me to run Microsoft Windows. In this book I use the open-source Eclipse Integrated Development Environment, which exists on all major platforms and looks pretty much the same on each. So regardless of your preferred operating system, you'll be able to run all the code samples from this book.

Page 33 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

CONVENTIONS

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.



Notes, tips, hints, tricks, and asides to the current discussion are offset and placed in italics like this.



References like this one point you to the DVD or `www.wrox.com` *to watch the instructional video that accompanies a given lesson.*

As for styles in the text:

- We *highlight* new terms and important words when we introduce them.
- We show file names, URLs, and code within the text like so: `persistence.properties`.
- We present code like the following:

```
We use a monofont type with no highlighting for most code examples.
```

SOURCE CODE

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. All of the source code used in this book is available for download at `www.wrox.com`. You will find the code snippets from the source code are accompanied by a download icon and note indicating the name of the program, so you know it's available for download and can easily locate it in the download file. Once at the site, simply locate the book's title (either by using the Search box or by using one of the title lists) and click the Download Code link on the book's detail page to obtain all the source code for the book.



Listings include the filename in the title. If it is just a code snippet, you'll find the filename in a code note such as this:

Code Filename

Page 34 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon



Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 978-0-470-88964-0.

After you download the code, just decompress it with your favorite compression tool. Alternately, you can go to the main Wrox code download page at `www.wrox.com/dynamic/books/download.aspx` to see the code available for this book and all other Wrox books.

The author will also maintain up-to-date code samples in the code repository hosted by Google at the following address: `http://code.google.com/p/practicaljava/`.

This website also contains a sample online store application contributed by Nick Golubyev. Even though this application was not used in the book, it serves as a good example of an end-to-end application developed with Java EE 6 technologies. You can find the application by clicking the Source tab in the folder `java-ee`. You can find instructions on building the online store under the tab Wiki.

ERRATA

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, like a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata you may save another reader hours of frustration, and at the same time you will be helping us provide even higher quality information.

To find the errata page for this book, go to `www.wrox.com` and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page you can view all errata that has been submitted for this book and posted by Wrox editors. A complete book list including links to each book's errata is also available at `wrox.com/misc-pages/booklist.shtml`.

If you don't spot "your" error on the Book Errata page, go to `wrox.com/contact/techsupport.shtml` and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

P2P.WROX.COM

For author and peer discussion, join the P2P forums at `p2p.wrox.com`. The forums are a Web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

Page 35 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

At p2p.wrox.com you will find a number of different forums that will help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to p2p.wrox.com and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join as well as any optional information you wish to provide and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.



You can read messages in the forums without joining P2P but in order to post your own messages, you must join.

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the Web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

Page 36 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

1

Introducing Java

Today Java is one of the most popular programming languages for everything from programming games to creating mission-critical applications such as those for trading on Wall Street or controlling Mars rovers. In this lesson you are introduced to some of the very basic Java terms, and you will download and install the Java Development Kit (JDK) and compile your first program.

WHY LEARN JAVA?

The Java programming language was originally created in 1995 by James Gosling from Sun Microsystems (currently a subsidiary of Oracle Corporation). The goal was to provide a simpler and platform-independent alternative to C++. You'll see what *platform independence* means a little later, in the section "The Life Cycle of a Java Program." For now, let's look at some of the reasons why Java can be your language of choice.

Java is a general-purpose programming language that's used in all industries for almost any type of application. If you master it, your chances of getting employed as a software developer will be higher than if you specialize in some domain-specific programming languages.

There are around six million professional Java developers in the world and the majority of them are ready to share their knowledge by posting blogs and articles or simply answering technical questions online. If you get stuck solving some problem in Java, the chances are very high that you'll find the solution on the Internet.

Since the pool of Java developers is huge, project managers of large and small corporations like to use Java for the development of new projects — if you decide to leave the project for whatever reason, it's not difficult to find another Java programmer to replace you. At this point you may ask, "Does that also mean that my Java skills will be easily replaceable?" To improve your value and employability you need to *master* not only the syntax of the language, but also the right set of Java-related technologies that are in demand (you'll learn them in this book).

Not only is Java open-source, but there are millions of open-source projects being developed in Java. Joining one of these projects is the best way to get familiar with the process of project development and secure your very first job without having any prior real-world experience as a programmer.

The Java language is *object-oriented* (OO), which allows you to easily relate program constructs to objects from the real world (more on this in Lessons 3–7).

There are plenty of technical features that make Java the right choice for many projects, and you'll have a chance to see this for yourself while reading this book, watching the screencasts from the accompanying DVD, and deploying all code samples from the book on your computer.

SETTING THE GOALS

The goal of this rather slim tutorial is to give you just enough information about most of the Java language elements, techniques, and technologies that are currently being used in the real world. The first half of the book is more about the Java language itself, while the second is about server-side Java technologies, and this is where Java shines in the enterprise world.

The brevity of some of the lessons may make you wonder if it's even possible to explain a subject in just 10 pages while there are whole books devoted for the same topic. My approach is to cover just enough for you to understand the concept, important terms, and best practices. Prerecorded screencasts on the DVD will help you to repeat the techniques explained in the lesson on your own.

There are plenty of additional materials online that will help you to study any specific topic more deeply. But you'll get a working and practical knowledge about Java just by using the materials included with this book.

The goal of this book is not just to get you familiar with the syntax of the Java language, but to give you practical Java skills that will allow you to develop business applications either on your own or by working as a team member in a larger-scale project.

THE LIFE CYCLE OF A JAVA PROGRAM

There are different types of programming languages. In some of them you write the text of the program (aka the source code) and can execute this program right away. These are interpreted languages (e.g., JavaScript).

But Java requires the source code of your program to be compiled first. It gets converted to either machine-specific code or a bytecode that is understood by some run-time engine or a virtual machine.

Not only will the program be checked for syntax errors by a Java compiler, but some other libraries of Java code can be added (*linked*) to your program after the compilation is complete (deployment stage).

In this lesson you write a very basic Java program that will output the words "Hello World" on your computer's screen.

Technically you can write the source code of your Java program in any plain text editor that you prefer (Notepad, TextEdit, vi, etc.), but to compile your program you'll need additional tools and code libraries that are included in the Java Development Kit (JDK).

Page 38 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

JDK AND JRE

If you are planning to use a specific computer to develop Java programs, you'll need to download and install JDK. If you are planning to use this computer only to run Java programs that were compiled somewhere else, you just need the Java Runtime Environment (JRE).

If you have JDK installed on your machine, it includes JRE.

Java's platform independence comes from the fact that your Java program doesn't know under which operating system (OS) or on which hardware it's being executed. It operates inside the preinstalled JRE that is pretty much the same on every platform.

Since you'll be learning how to develop Java programs, download JDK from the following website: `http://www.oracle.com/technetwork/java/javase/downloads/jdk6-jsp-136632.html`.

JAVA SE AND EE

But before rushing to the downloading process, you need to get familiar with two more terms: Java SE (Standard Edition) and Java EE (Enterprise Edition). The latter includes the server-side tools and libraries that you'll get familiar with starting in Lesson 26.

For now, just download the latest version of the JDK SE Development Kit. (The letter U followed by a number represents the update number of Java 6.)

Select the platform (Windows, Linux, Solaris) on which you are planning to develop Java programs and continue the download process.



JDK for Mac OS X is preinstalled on Apple's computers and no additional installing is required. The current version of JDK, which comes with computers running Snow Leopard, is JDK 6. It's located in directory /Library/Java. In this book we'll be using Java for the Windows OS, but Java is cross-platform and the book examples will work under MAC OS and Linux as well.

DOWNLOADING AND INSTALLING JDK IN WINDOWS

After selecting the Windows platform and clicking the Download button you'll see a Login for Download screen, which is optional: You can skip this step.

Click Save File on the pop-up screen, as shown in Figure 1-1 (the file name depends on the version of JDK).

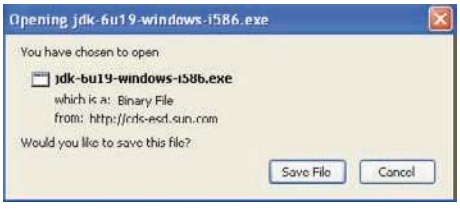


FIGURE 1-1

After the file is saved, start this executable and the installation wizard will lead you through the process. Read and accept the license agreement and click the button Next. Note the item at the bottom of the left box in Figure 1-2 — it is an open-source database management system (DBMS) called Java DB. You'll need JavaDB for Lesson 22.

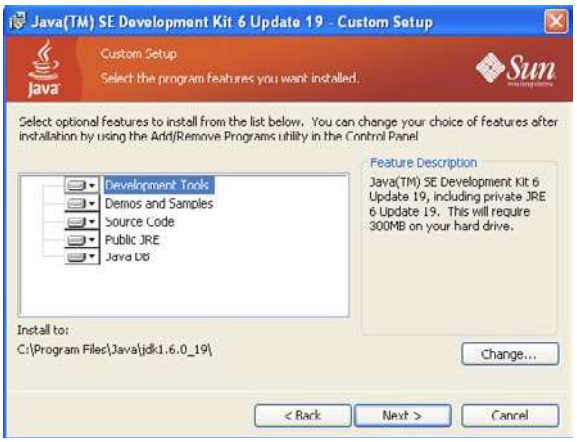


FIGURE 1-2

A couple of minutes into the installation process you'll see a pop-up window asking you where to install JRE. I'll assume that you've accepted the default directory (`c:\Program Files\Java\jre6`), but you can select a different one. Shortly, you should see a message telling you that installation was successful. Click Finish.

At the end of the installation process you'll see a website suggesting you register the install. This step is optional.

If you have previous versions of JDK installed on your computer, each of them will be sitting in its own directory, e.g., `c:\Program Files\Java\jdk1.6.0_019`. But for a sanity check, I always open a command line window and enter `java -version` at the command prompt. Figure 1-3 shows the confirmation that I really have JRE 1.6.0_19 (Java 6 is also referred to as Java 1.6).

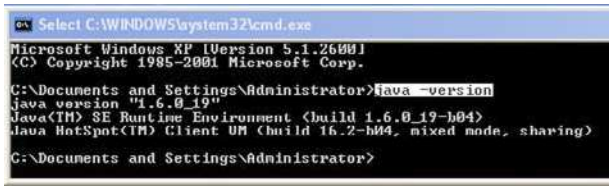


FIGURE 1-3

Congratulations! Your JDK and JRE are installed.

YOUR FIRST JAVA PROGRAM: HELLO WORLD

Historically, the first program you write while learning a new programming language is the program Hello World. If you visit the website `www.helloworldexample.net/` you'll see how to write this program in many different languages, including Java.

To start writing a Java program you could use any plain text editor — I'll use Notepad. The file that contains Java code must be saved in a file with its name ending in `.java`.

Enter the following code in the text editor.



LISTING 1-1: HelloWorld.java

```
public class  HelloWorld {

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

Create a directory, `c:\PracticalJava\Lesson1`, and save the program you just created in the file `HelloWorld.java` (if you use Notepad, select All Files in the Save as Type drop-down to avoid auto-attachment of the `.txt` suffix).

Keep in mind that Java is a case-sensitive language, which means that if you named the program HelloWorld with a capital *H* and a capital *W*, you should not try to start the program helloworld. Your first dozen syntax errors will be caused by improper capitalization.

What follows is a really short explanation of some of the terms and language elements used in the HelloWorld program. You'll get more comfortable with them after mastering the first several lessons in this book.

Our first program contains a *class*, `HelloWorld`. Give the Java class and its file the same name. (There could be exceptions to this rule, but not in this simple program.) While writing Java programs you create *classes*, which often represent objects from real life. You'll learn more about classes in Lesson 3.

The class `HelloWorld` contains a *method*, `main()`. Methods in Java classes represent functions (actions) that a class could perform. A Java class may have several methods, but if one of them is called `main()` and is declared (if it has a *method signature*), as in our class, this makes this Java class executable. If a class doesn't have a method `main()`, it can be used from other classes, but you can't run it as a program.

The method `main()` calls the method `println()` to display the text "Hello World" on the screen. Here is the method signature (similar to a title) of the method `main()`:

```
public static void main(String[] args)
```

This method signature includes the access level (`public`), instructions on usage (`static`), return value type (`void`), name of the method (`main`), and argument list (`String[] args`).

- The keyword `public` means that the method `main()` can be accessed by any other Java class.
- The keyword `static` means that you don't have to create an instance of this class to use this method.
- The keyword `void` means that the method `main()` doesn't return any value to the calling program.
- The keyword `String[] args` tells us that this method will receive an array of characters as the argument (you can pass external data to this method from a command line).

The `main()` method is the starting point of your program. You can write a program in Java SE that consists of more than one class, but at least one of them has the method `main()`, otherwise the program won't start. A Java class can have more than one method. For example, the class `Employee` can have the methods `updateAddress()`, `raiseSalary()`, `changeName()`, etc.

The *body* of the method `main()` contains the following line:

```
System.out.println("Hello World");
```

The preceding `println()` method is used to print data on the system console (command window). Java's method names are always followed by parentheses.

`System` here represents another Java class.

The dot notation `System.out` means that the variable out is defined inside the class `System`.

`out.println()` tells you that there is an object represented by a variable called `out` and it has a method called `println()`.

You will be using this dot notation to refer to class methods or variables.

Compiling and Running Hello World

Now you need to compile this program. The `javac` compiler is a part of JDK, so open a command window on your PC, change the current directory to `c:\PracticalJava\Lesson1` and try to compile the program:

```
cd PracticalJava\Lesson1
 javac HelloWorld.java
```

Oops. I got an error telling me that `javac` is not recognized as an internal or external command, operable program, or batch file. The OS doesn't know where `javac` is located. You need to add the directory in which the `javac` compiler resides to the system variable `PATH`. In Windows OS, go to Control Panel, click the System icon, select the tab Advanced, and click Environment Variables. Edit the system variable `PATH` so it starts with the following: `C:\Program Files\Java\jdk1.6.0_19\bin;` By the time you'll be reading this book the Java version number may be different so check the name of the directory before modifying the system variable PATH.

Restart your command window and now the OS should be able to find `javac`. Get into the directory where `HelloWorld.java` is located and repeat the compilation command.

You won't see any confirmation of a successful compilation; just type `dir` to confirm that a new file named `HelloWorld.class` has been created. This proves that your program has been successfully compiled.

If the program has syntax errors, the compiler prints error messages. In this case fix the errors, and recompile the program again. You may need to do it more than once until the file `HelloWorld.class` is successfully created.

Now run the program by typing the command

```
java HelloWorld
```

Note that this time we used the term `java`, which starts the Java Virtual Machine (JVM). The words "Hello World" are displayed in the command window. In Figure 1-4 is a screenshot that shows the compilation command, the content of the folder after the compilation, and the output of the HelloWorld program.

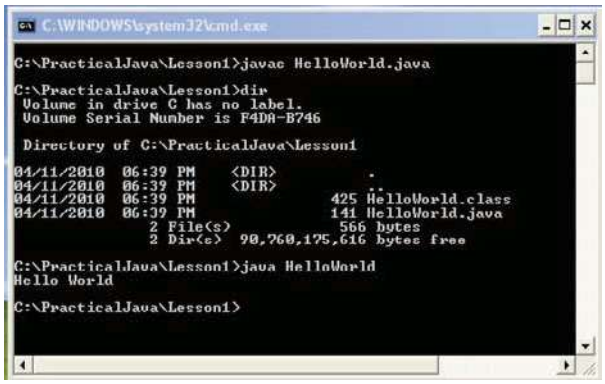


FIGURE 1-4

TRY IT

In this lesson your goal is to write your first Java program that outputs the words "Hello World."

Lesson Requirements

For this lesson download and install the current version of JDK from `http://java.sun.com/javase/downloads/index.jsp`. Set the value of the system `PATH` variable to include the `bin` directory, where your Java compiler (`javac`) and other executables were installed.

Step-by-Step

1. Open a plain text editor of your choice and enter the text of the Hello World program.
2. Save the program in the file `HelloWorld.java`.
3. Compile the program in the command window using the command `javac HelloWorld.java`.
4. Run the program by using the command `java HelloWorld`.



Please select Lesson 1 on the DVD with the print book, or watch online at `www.wrox.com/go/fainjava` *to view the video that accompanies this lesson.*

2

Eclipse IDE

While your first Java program was written in a plain text editor and compiled from a command window, this is not a productive way of developing software. Professional programmers use one of the Integrated Development Environments (IDEs), which include an editor, a compiler, type-ahead help, a debugger, and a lot more (you'll get familiar with these features later in this lesson). There several popular Java IDEs, such as Eclipse, NetBeans, IntelliJ IDEA, and RAD. Some are free, and some are not.

Eclipse is by far the most widely used IDE, and I'll be using it for compiling and running most of the examples in this book. But switching from one IDE to another is a pretty simple process, and if you see that in some areas one IDE makes you more productive than the other, just use the best one for the job. For example, NetBeans IDE can offer great help if you'll be developing the server-side Java EE applications explained later in the book (starting in Lesson 26).

INTRODUCING ECLIPSE IDE

Eclipse IDE is an open-source product that was originally created with a substantial code donation by IBM to the Java community, and from that moment Eclipse was a 100% community-driven product. It started as an IDE for developing Java programs, but today it's a development platform used for building thousands of tools and plug-ins. Some people are using its Rich Client Platform (RCP) API to develop user interfaces (UIs) for applications. Other Java languages are also supported by Eclipse. Visit the downloads page to see a just a portion of the Eclipse-based products available: `www.eclipse.org/downloads`.

Besides being an IDE, Eclipse supports plug-in development, and each developer can add only those plug-ins that he or she is interested in. For example, there is a plug-in to display UML diagrams, another offers a reporting system, and there are plug-ins for developing applications in C, Adobe Flex, and others.

DOWNLOADING AND INSTALLING ECLIPSE

At work, I use Eclipse IDE for Java EE Developers. Each version of Eclipse IDE has a name. At the time of this writing, the current version is called Helios. In Windows OS, my first download window looks as shown in Figure 2-1 (note `jee` in the file name).

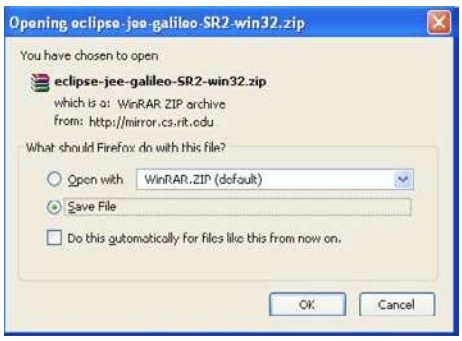


FIGURE 2-1

The installation of Eclipse IDE comes down to a simple unzipping of the downloaded file into a disk drive of your choice. You'll find the file `eclipse.exe` in the Eclipse folder — just run this program. You'll immediately see a pop-up window asking you to select a *workspace*, which is a directory on your hard disk where one or more of your *projects* is going to be stored.

If you want to create an application in Eclipse, you start with creating a project. In the real world, the source code of a decent-size application can consist of several Eclipse projects.

For this book I selected the following workspace directory: `c:\PracticalJava\workspace`.

Eclipse Java EE IDE starts with a Welcome panel — just close it by clicking the little x on the Welcome tab. Figure 2-2 is a snapshot of the freshly installed IDE with an empty workspace.

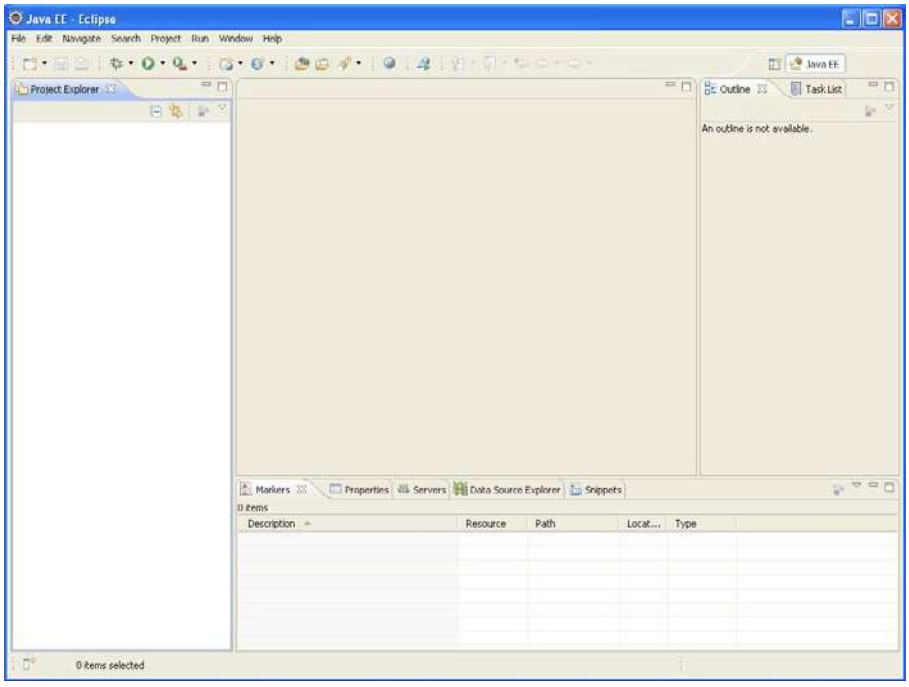


FIGURE 2-2

To be precise, you are looking at Java EE *perspective* (note the Java EE tab at the top), which is a collection of *views*. On the left you see a Project Explorer view. The area in the middle is reserved for the code editor view — you start using it as soon as you create your first Java class. The Outline view is on the right — you'll see the names of your classes, methods, and variables (see Lesson 3) there when they are available.

There are many other views that you can open and close by yourself by selecting Window ➪ Show View. These include Console, Search, Servers and others.

Since you are just starting to learn the language, there is no need to work in the Java EE perspective — you can get by in the Java perspective. Click the little icon with the plus sign by the Java EE tab and select "Java perspective." You'll see a slightly different set of views with the Package Explorer and Hierarchy views on the left, Task List on the right, and Problems, Javadoc and Declaration tabs at the bottom, as shown in Figure 2-3.

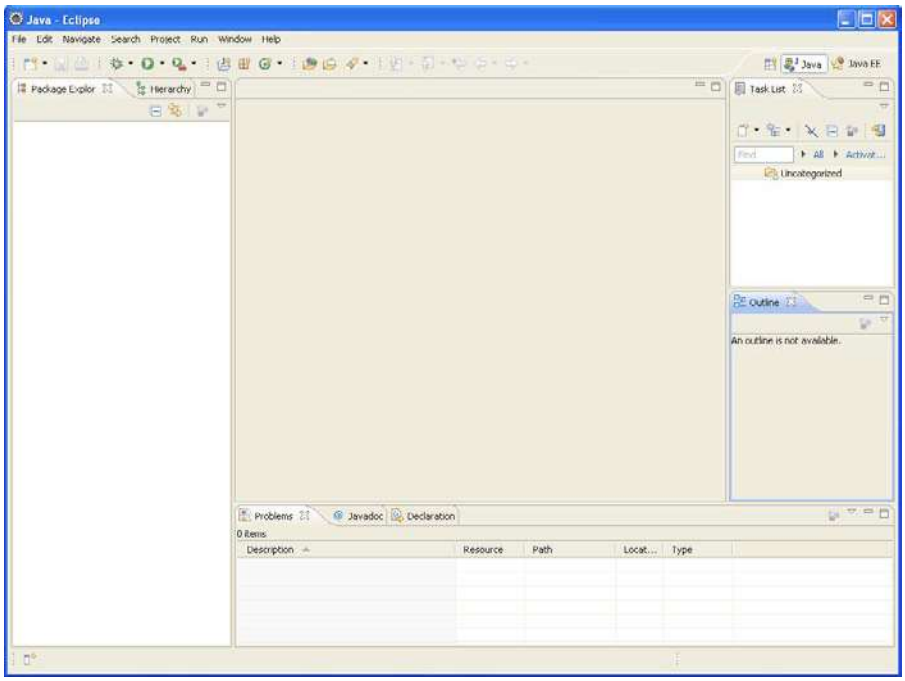


FIGURE 2-3

CREATING HELLO PROJECT IN ECLIPSE

In Lesson 1 you simply created the class `HelloWorld`, but in Eclipse you have to create the project first. Select File ➪ New ➪ Java Project and enter `Hello` as the name of the project in the pop-up window, as shown in Figure 2-4.

You can select the version of the JRE you want to work with. In Lesson 1 I've installed the JDK and JRE of version 1.6.0_19, but you may have more than one version of JRE, and Eclipse can compile the code that will run in another version of JRE. This is a pretty nice feature of Eclipse IDE — it doesn't come with its own JRE, but it allows you to pick the version that fits your needs. In some cases you have to compile the code with older versions of JRE if you know that this program will have to run in the older JVMs.

After you click Next, the next window will ask you to specify the folder where compiled Java classes of the Hello project should be created. By default, Eclipse creates a Hello folder for this project with a `bin` sub-folder for compiled classes and an `src` sub-folder for the source code. In Lesson 1 both `HelloWorld.java` and `HelloWorld.class` were sitting in the same folder, which is a bad practice.

Don't change the name of the output directory — just click Finish on that window. In Figure 2-5 you see a new project, Hello, in the Package Explorer view of Eclipse.

This project has an empty folder, `src` — you can save the source code of `HelloWorld.java` there when ready. The JRE folder contains all required libraries supporting the JVM where HelloWorld will run.

These library files have `.jar` in their names. Java SDK comes with a jar utility that allows you to create a file archive that contains one or more compiled classes. While the JRE folder contains classes created by developers of the JRE itself, most real-world applications consist of groups of files (packages) located in one or more jars.

It doesn't make much sense to put the only `HelloWorld` class inside the jar, but as your sample applications grow, you'll see how to group and compress files in jars.

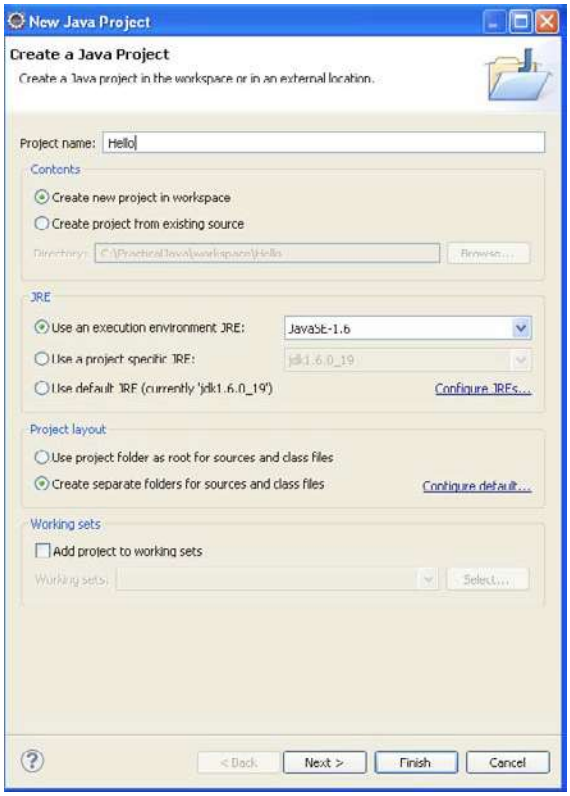


FIGURE 2-4

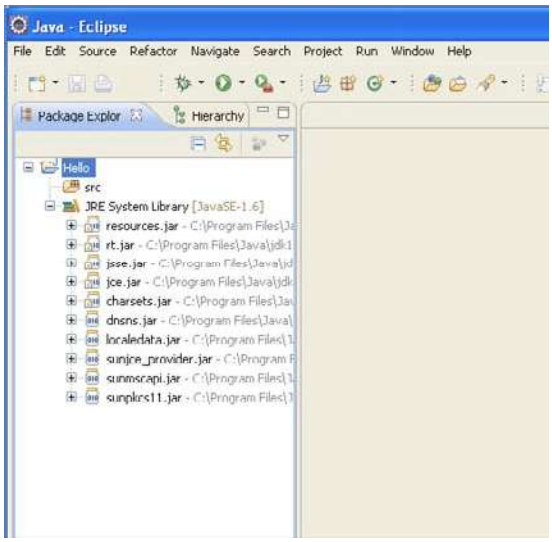


FIGURE 2-5

CREATING THE HELLOWORLD CLASS IN ECLIPSE

Our Hello project will contain one Java class: `HelloWorld` from Lesson 1. Select File ➪ New ➪ Class and enter `HelloWorld` in the Name field in the pop-up window shown in Figure 2-6.

Then enter `com.practicaljava.lesson2` in the Package field. This is a new addition that was not present in the previous version of `HelloWorld`.



FIGURE 2-6

JAVA PACKAGES

Packages in Java are used to better organize multi-file projects and for data protection. It's not unusual for a project to have several hundreds of Java classes. Keeping them all in one directory is never a good idea. Hence the files will be located in various directories and sub-directories.

What are the naming conventions for packages? Java developers use so-called *reverse-domain name* conventions. Let's say you work on a project called Sales for a company called Acme, which has an Internet site at `acme.com`. Then every package name will start with the reverse URL of the company, followed by the project name: `com.acme.sales`.

All Java classes that belong to this package would be stored in the following directory structure: `com/acme/sales`.

If some of the Java classes are specific to domestic sales, while others are used in international sales, you will create two more sub-directories: `com/acme/sales/domestic` and `com/acme/sales/international`.

While directory names are separated by a forward slash or backslash, the corresponding Java packages are separated with periods. Java has a special keyword package, and its declaration has to be the first line of the class (program comments don't count). For example:

```
package com.acme.sales.domestic;
```

Let's assume that you work for a company called Practical Java on project Lesson2; the name of the package will be `com.practicaljava.lesson2`, which is exactly what I've entered in the Package field shown in Figure 2-6.

Besides being used for better organization of Java classes, packages help in controlling data access. You learn about this feature in the section "Access Levels" in Lesson 6.

COMPLETING CODE GENERATION

You may have noticed that I also checked off the box asking Eclipse to generate the main method for me.

Click Finish, and in a couple of seconds Eclipse will generate for you the initial code for the class `HelloWorld`, as shown in Figure 2-7.

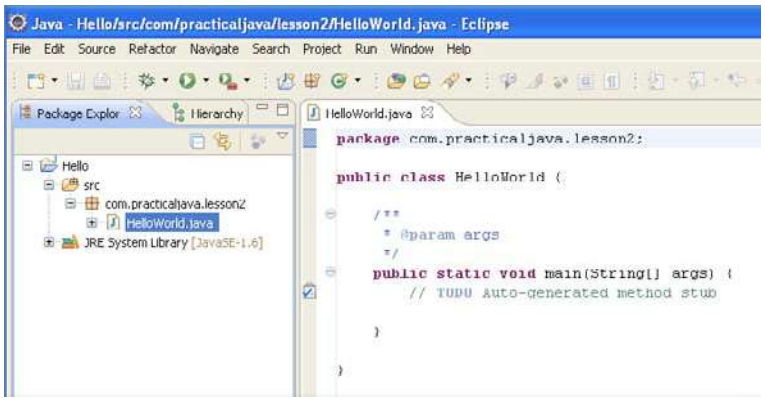


FIGURE 2-7

The generated code is shown in Eclipse's editor view. It starts with the `package` statement, and the class declaration with the method name goes next.

Type in the line `System.out.println("Hello World");` under the `TODO` comment (comments are explained in the next section), and save the code by pressing the little diskette image on the toolbar or using the Ctrl+S key combination.

As you type in the code, Eclipse displays context-sensitive help suggesting a selection of possible values, minimizing guesswork and typing errors. I made a snapshot, shown in Figure 2-8, right after entering `System.o`:



FIGURE 2-8

By default, saving the code results in an invocation of the compiler. If you didn't make any syntax errors, Eclipse will create `HelloWorld.class` in the `bin` directory of the Hello project. In case of compilation errors, Eclipse puts a little red round bullet in front of problematic lines.

Now you can run the program by pressing the round green button on the toolbar. The output of the program will be shown in the Console view panel in the lower part of Eclipse IDE, as in Figure 2-9.

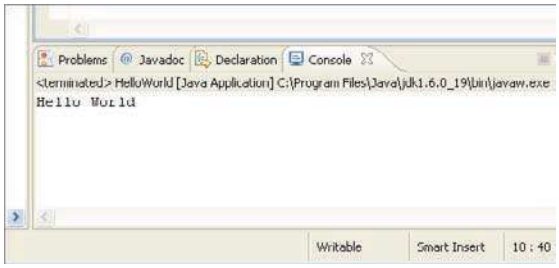


FIGURE 2-9

The format of this book doesn't have space for more detailed coverage of all the features of Eclipse. The good news is that there are plenty of tutorials describing every feature of Eclipse at `www.eclipse.org/documentation`.

TRY IT

In this lesson your goal is to write, compile, and run HelloWorld in Eclipse IDE.

Lesson Requirements

For this lesson download and install the current version of Eclipse IDE for Java EE Developers from `www.eclipse.org/downloads`.

Step-by-Step

1. Create the Hello project in Eclipse IDE.
2. Create a new Java class, `HelloWorld`, with the method `main()`, as described earlier.
3. Compile the program by clicking Save.
4. Run the program by clicking the green button in the Eclipse toolbar.



Please select Lesson 2 on the DVD with the print book, or watch online at `www.wrox.com/go/fainjava` *to view the video that accompanies this lesson.*

3

Object-Oriented Programming

Starting from this lesson you'll be studying various elements of the Java language with brief descriptions to get you started programming in the shortest possible time. But you are certainly encouraged to refer to the extensive Java documentation that's available online at `http://java.sun.com/javase/6/docs/`.

CLASSES AND OBJECTS

Java is an object-oriented language, which means that it has constructs to represent objects from the real world. Each Java program has at least one class that knows how to do certain things or how to represent some type of object. For example, the simplest class, `HelloWorld`, knows how to greet the world.

Classes in Java may have methods (aka functions) and fields (aka attributes or properties).

Let's create and discuss a class named `Car`. This class will have *methods*, describing what this type of vehicle can do, such as start the engine, shut it down, accelerate, brake, lock the doors, and so on.

This class will also have some *fields*: body color, number of doors, sticker price, and so on.



LISTING 3-1: Class Car

```
class Car{
    String color;
    int numberOfDoors;

    void startEngine {
       // Some code goes here
    }
    void stopEngine {
         int tempCounter=0;
```

continues

LISTING 3-1 *(continued)*

```
        // Some code goes here
    }
}
```

Lines that start with double slashes are comments, which will be explained later in this lesson.

`Car` represents common features for many different cars: All cars have such properties as color and number of doors, and all of them perform similar actions. You can be more specific and create another Java class called `Car`. It's still a car, but with some properties specific to the model James Bond. You can say that the class `JamesBondCar` is a *subclass* of `Car`, or, using Java syntax, `JamesBondCar extends Car`.



LISTING 3-2: Class JamesBondCar

```
class JamesBondCar extends Car{
    int currentSubmergeDepth;
    boolean isGunOnBoard=true;
    final String MANUFACTURER;

    void submerge {
        currentSubmergeDepth = 50;
       // Some code goes here
    }
    void surface {
      // Some code goes here
    }
}
```

But even after defining all the properties and methods for the class `JamesBondCar` you can't drive it, even on the computer screen. A Java class is like a blueprint in construction or engineering, and until you build real objects based on this blueprint you can't use them.

Creating objects, aka *instances*, based on classes is the equivalent of building real cars based on blueprints. To create an instance of a class means to create the object in the computer's memory based on the class definition.

To instantiate a class (to put one more car on the road), you'll declare a *variable* of this class's type, and use the `new` operator for each new instance of the car:

```
JamesBondCar car1 = new JamesBondCar();
JamesBondCar car2 = new JamesBondCar();
```

Now the variables `car1` and `car2` can be used to refer to the first and second instance of the `JamesBondCar` respectively. To be precise, declaring the variables pointing at the instances is needed if you are planning to refer to these instances in the program. The statement `new JamesBondCar()` creates the instance, too. You can create many cars based on the same specification. Even though they all represent the same class, they may have different values in their properties — some of them are red, some of them have two doors while others have four, etc.

VARIABLES, CONSTANTS, AND DATA TYPES

Some values representing an object can change over time (variables) and some remain the same (constants). This section will shed more light on the use of both types.

Declaring Variables

Java is a statically typed language: The program *variables* must be declared (named and typed) first, and then you can assign them values either at the time of declaration or later on in one of the class methods. For example, the variable `isGunOnBoard` has been initialized during its declaration in Listing 3-2, and `currentSubmergeDepth` got its value in the method `submerge()`.

The class `Car` from Listing 3-1 defines a *variable* color of type `String`, which is used to represent text values.

Constants

To store values that never change, you need to declare a *constant* — just add the keyword `final` to the declaration line, as in Listing 3-2:

```
final String MANUFACTURER;
```

Java developers usually render the names of constants in upper case. The value of a constant can be assigned only once, and since you are creating an instance of a specific car, its manufacturer is known and can't change during the life span of this object:

```
 MANUFACTURER = "J.B. Limited";
```

Even though you don't have to initialize final variables during their declaration, it is a good practice to follow. The preceding example should be written as

```
final String MANUFACTURER="J.B. Limited";
```

Primitive Data Types

There are eight primitive data types in Java: Four are for integer values, two are for values with a decimal point, one is for storing single characters, and one is for Boolean data that allows only either `true` or `false` as a value. Following are some examples of variable declarations and initializations:

```
int chairs = 12;
char grade = 'A';
boolean cancelJob = false;
double nationalIncome = 23863494965745.78;
float hourlyRate = 12.50f;   // add an f at the end of
                             //float literals
long totalCars  = 46372836483921;  // add an l at the end
                                   // of long literals
```

The following table summarizes some characteristics of the Java data types.

PRIMITIVE TYPE	SIZE	MIN VALUE	MAX VALUE	WRAPPER CLASS
byte	8 bits	-128	127	Byte
short	16 bits	-32,768	32,767	Short
int	32 bits	-2,147,483,648	2,147,483,647	Integer
long	64 bits	-9,223,372,036,854,775,808	9,223,372,036,854,775,807	Long
float	32 bits	Single precision floating point; see Java language specification at http://bit.ly/9nlwjh	Single precision floating point; see Java language specification at http://bit.ly/9nlwjh	Float
double	64 bits	Double precision floating point; see Java language specification at http://bit.ly/9nlwjh	Double precision floating point; see Java language specification at http://bit.ly/9nlwjh	Double
char	16 bits	Unicode 0	Unicode 2 in a power of 16 value	Character
boolean	-	false (not a min)	true (not a max)	Boolean

Have you noticed that the `char` data type uses two bytes of memory to store the data? This allows you to store character sets that have a lot more symbols than traditional alphabets because a single byte can only represent 256 characters, while two bytes can represent 65,536 characters.

Following are some examples of variable declarations and initializations:

```
int chairs = 12;
boolean cancelJob = false;
double nationalIncome = 23863494965745.78;
float hourlyRate = 12.50f;
long totalCars  = 4637283648392L;
```

The last two literals in the preceding list end with the letters `f` and `L` to indicate that you want to store these data as `float` and `long` data types correspondingly. But most likely, the `double` datatype should fit all your needs in non-integer calculations.

VARIABLE SCOPE

If you declare a variable inside any method, the variable has a local scope (for example, `adjustedTax` in Listing 3-5 is local). This means that it's only visible for the code within this method (`calcTax`). When the method completes its execution, all local variables are automatically removed from memory by Java's *garbage collector*. Actually the scope is tighter than that.

It's accessible within the method only after the variables are declared, and only within the block in which it is declared. For instance, a variable declared inside a `for` loop will not be excessive outside the `for` loop even inside the same method.

If a variable has to be accessible from more than one class method, declare it on a class level. Listing 3-5 shows the class Tax, where grossIncome, dependents, and state are *class or member variables*. These variables are "alive" while the `Tax` object exists in memory. They can be shared and reused by all methods within the class and they can even be visible from external classes.



If a variable is declared with a static qualifier (see Lesson 4), it's shared by all instances of the class.

WRAPPERS, AUTOBOXING, AND UNBOXING

All primitive data types have corresponding *wrapper classes* that contain useful methods dealing with respective data types. The wrapper classes serve two purposes:

1. They contain a number of useful functions for manipulation with their primitive counterparts. For example, the class Integer offers such useful methods as conversion of a String into an int, or turning an int into a float, and more. The `Integer` class also allows you to set the minimum and maximum values for the number in question.
2. Some Java collections can't store primitives (such as `ArrayList`) and so primitives have to be wrapped into objects. For example,

```
ArrayList myLotteryNumbers = new ArrayList();
myLotteryNumbers.add(new Integer(6));
myLotteryNumbers.add(new Integer(15));
```

You don't need to explicitly create a new instance for every primitive as in the preceding code snippet. You can simply write `myLotteryNumbers.add(6);` and the primitive value `6` will automatically be wrapped into an instance of the `Integer` class. This feature is called *autoboxing*.

On the same note, the next line is also valid:

```
int luckyNumber= myLotteryNumber.get(23);
```

Even though `get(23)` will return the value of the 24th element (the numbering in Java collections starts with zero) as an `Integer` object, that object will automatically be converted into a primitive. This is called *unboxing*.

PROGRAM COMMENTS

While writing code in Java, you should add comments, the text that explains what the program does. Programs are being read a lot more often than they are being written. At some point, other software developers will read and try to understand your code. Be nice and make their job easier. A typical software developer doesn't like writing comments (regardless of what programming language he or she uses).

I can suggest to you a simple technique: Write comments first and only then the code. By the time your program is written it's already commented. You can write comments pretty much everywhere — before or inside the class, or inside the methods.

In Java you can use three types of comments:

- Block comments that contain more than one line of text located between the symbols `/*` and `*/`. For example:

```
/* This method will calculate the cost of shipping, handling,
     and all applicable taxes
*/
```

  The compiler ignores the text in comments and you can write whatever you want.

- If you want to write a short comment that fits on a single line, just start this line with two forward slashes (`//`). For example:

```
// This method will calculate the cost of shipping
```

- Some comments start with `/**` and end with `*/`. These are used by a special utility, javadoc, that can automatically extract the text from these comments and create program documentation. To get a feeling for what javadoc can generate, look at the documented API for Java itself `http://java.sun.com/javase/6/docs/api/overview-summary.html`.

If you'd like to learn how to use the javadoc utility, refer to its program documentation at `http://java.sun.com/javase/6/docs/technotes/guides/javadoc/index.html`.

First Useful Program

It's time to write a program that does something more useful than print "Hello World." This program will emulate the calculation of state tax. The goal is to show you how Java classes communicate, how methods are called, and how variables can be used.

First you need to decide what Java class(es) you need to write for the task at hand. Then think about the properties (class variables) and methods (functions) these classes should have.

Declaring a Tax Class with Properties

Since you are planning to calculate tax, it doesn't take a rocket scientist to figure out that you need to define a class called `Tax`. Start with the class name and curly braces — this is the simplest class you can create:

```
class Tax{
}
```

What data does this class need to perform tax calculations? You'll definitely need to know the gross income of a person for the tax year. This is a good candidate for a property of this class. Properties in Java are represented by *variables*. Pick one of the numeric data types. Gross income is not always

Page 58 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

an integer number, so let's use the `double` data type, as it's a number with a decimal point. You could use `float` instead, but you expect to process some very large incomes:

```
class Tax{
      double grossIncome;
}
```

You also need to know what state the person lives in — taxation rules vary by state. These are some of the abbreviations for the states in the USA: NY, NJ, CT. Use the data type String for storing text data:

```
class Tax{
      double grossIncome;
      String state;
 }
```

Add one more property for dependents of the taxable person. `Integer` will work just fine here — a person can't have two and a half dependents:

```
class Tax{
      double grossIncome;
      String state;
      int  dependents;
 }
```

Adding a Method to the Tax Class

Variables store data, and methods perform actions. It's time for actions. The first method, `calcTax()`, will calculate the state tax based on the values of gross income, number of dependents, and state:



LISTING 3-3: Class Tax

```
class Tax{
          double grossIncome;
          String state;
          int dependents;

          public double calcTax() {

             return 234.55;
          }
}
```

The `calcTax()` method signature tells the following:

- Any external class can access this method (`public`).
- This method will return a value of type `double`.
- The name of the method is `calcTax`.

The empty parentheses after the method name mean that it does not have any arguments or, in other words, it does not need any values from outside `Tax` to perform calculations. As a matter of fact, this version of `calcTax()` doesn't use any data for calculation — it just returns a hard-coded tax value of $234.55.

How do you decide if a method should return a value? If your method performs some calculations and has to give a value back to a calling program, it has to return a value. If a method directly modifies the class variables or simply outputs data somewhere (monitor, disk, server) it may not need to return any values. You still need to declare a "no return" in a method signature by using a special keyword, `void`:

```
public void PrintAnnualTaxReturn() {

    //Code goes here
}
```

With the Java `return` statement a method can return data contained in a variable to a calling program, for example:

```
return calculatedTax;
```

Keep in mind that if you declare a return type in the method signature but forget to include the `return` statement in the body of the method, the Java compiler will give you an error.

Declaring Another Class: TestTax

`Tax` will know how to calculate tax, but in a real-world application you'll have many classes that represent the various workflows of this process. For example, you may need to create a class called `Customer`. Depending on the type of employment or income, accountants use many different forms to file taxes, and each form can be represented by a separate class: `Form1040`, `Form1099`, and so on.

Each of these classes represents some entity, but none of them is an executable program — none of them will have the method `main()`. You need to create one more class to start the application and instantiate other classes as needed. I'll call this class `TestTax`. The class `TestTax` should be able to perform the following actions:

- Create an instance of the class `Tax`.
- Assign customer's data (gross income, state, dependents) to the class variables of the class `Tax`.
- Call the method `calcTax()`.
- Print the result on the screen.

The class `TestTax` will be stored in a separate file named `TestTax.java`.



LISTING 3-4: Class TestTax

```
class TestTax{
     public static void main(String[] args){
            Tax   t = new Tax(); // creating an instance
```

```
            t.grossIncome= 50000;  // assigning the values
            t.dependents= 2;
            t.state= "NJ";

            double yourTax = t.calcTax(); //calculating tax

           // Printing the result
           System.out.println("Your tax is " + yourTax);
    }
}
```

In the preceding code we've declared a variable, t, of type `Tax`.

The method `main()` is an entry point to the tax-calculation program. This method creates an instance of the class Tax and the variable t points to a place in your computer's memory where the `Tax` object was created. From now on, if you want to refer to this object use the variable t.

The following three lines assign values to the properties of the object `Tax`:

```
t.grossIncome= 50000;  // assigning the values
t.dependents= 2;
t.state= "NJ";
```

After that you can calculate tax by calling the method `calcTax()`, and the result returned by this method will be assigned to the variable `yourTax`. The method `calcTax()` still returns the hard-coded value, but you'll fix this in the "Try It" section of this lesson. The last line just displays the result on the system console.

At this point we already have two classes communicating with each other (`TestTax` and `Tax`).

Conditional Statement if

We make decisions all the time: "If she says this I'll answer with that, otherwise I'll do something else." Java has an `if` statement that determines whether some condition is true or false. Based on the answer to this question the execution of your program will be routed.

If the condition expression returns `true`, the code between the first curly braces will be executed; otherwise the code after the `else` statement will take place. For example:

```
if (totalOrderPrice > 100){
        System.out.println("You'll get a 20% discount");
}
else{
        System.out.println("Order books for more than a" +
                           " $100 to get a 20% discount");
}
```

Because this code sample has only one statement to execute in the `if` and `else` clauses, using curly braces is not a must, but they make the code more readable and prevent you from introducing hard-to-find bugs if, later on, you need to add more code in an `if` statement.

Page 61 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

switch Statement

The `switch` statement is an alternative to `if`. The case label in the switch condition (`taxCode`) is evaluated and the program goes to one of the following `case` clauses:

```
int taxCode=someObject.getTaxCode(grossIncome);
switch (taxCode){
   case 0:
     System.out.println("Tax Exempt");
     break;
   case 1:
     System.out.println("Low Tax Bracket");
     break;
   case 2:
     System.out.println("High Tax Bracket");
     break;
   default:
     System.out.println("Wrong Tax Bracket");

}
```

The preceding code will call only one `println()` method. Do not forget to put the `break` at the end of each `case` statement so the program will jump out of the `switch` statement after processing a case; otherwise the code will "fall-through" and print more than one line even though a `taxCode` can have only one value.

Inheritance

In object-oriented languages the term *inheritance* means an ability to define a new class based on an existing one (not from scratch).

Imagine that the class `Tax` calculates tax properly in all states except New Jersey, which has introduced new educational tax deductions. If you have a kid in college, this makes you eligible for an additional $500 deduction from your taxes. In this case you have to either change the method `calcTax()` in the class `Tax` to introduce a special case for New Jersey, or create another class based on `Tax` and add this new functionality there.

Every person inherits some features from his or her parents. A similar mechanism exists in Java. The special keyword `extends` is used to indicate that one class has been inherited from another:

```
class NJTax extends Tax{
}
```

The class `NJTax` will have all the features the class `Tax` has, plus you can add new properties and methods to it. In such a setup, the class `Tax` is called a *superclass*, and `NJTax` is called a *subclass*. You can also use the terms *ancestor* and *descendent*, respectively. This new class will have access to all variables and methods of its superclass, unless those have a private or package access level, which will be discussed in Lesson 5.

Let's extend the behavior of the class `Tax` in `NJTax`. The latter will have a method called `adjustForStudents()`:



LISTING 3-5: Class NJTax

```
class NJTax extends Tax{

    double adjustForStudents (double stateTax){
       double adjustedTax = stateTax - 500;
       return adjustedTax;
    }
}
```

To use this new method, the `TestTax` class should instantiate `NJTax` rather than `Tax` as it did in Listing 3-4:

```
NJTax  t= new NJTax();
```

Now you can call methods defined in the class `Tax` as well as those from `NJTax` using the reference variable `t`, for example:

```
NJTax  t= new NJTax();
double yourTax = t.calcTax();
double totalTax = t. adjustForStudents (yourTax);
```

I've added a new functionality to the tax-calculation program without changing the code of the class `Tax`. The preceding code fragment also shows how you can pass a result of processing from one method to another. The value of the variable `yourTax` was calculated by `calcTax()` and is passed to the method `adjustForStudents()` as an *argument*.

Method Overriding

Yet another important term in object-oriented programming is *method overriding*. Imagine `Tax` with 20 methods. Most of them work fine for all states, but there is one method that is not valid for New Jersey. Instead of modifying this method in the superclass, you could create another method in the subclass *with the same name and argument list*. If a subclass has that method with the same name and argument list, it will override (suppress) the corresponding method of its ancestor.

Method overriding comes in handy in the following situations:

- The source code of the superclass is not available, but you still need to change its functionality.
- The original version of the method is still valid in some cases and you want to keep it intact.
- You use method overriding to enable polymorphism, which will be explained in Lesson 7.

You'll have a chance to try method overriding in the "Try It" section. In Lesson 4 you learn about method overloading, which is a completely different animal.

TRY IT

In this section you create in Eclipse the tax-calculation application described in this lesson, and then modify it to replace the hard-coded value returned by the method `calcTax()` with some calculations. After this is done, you'll subclass the class Tax and override `calcTax()`.

Lesson Requirements

For this lesson you must have Eclipse IDE installed.



You can download the code and resources for this Try It from the book's web page at `www.wrox.com`. *You can find them in the Lesson3 folder in the download.*

Hints

This lesson has only brief introductions to basic Java language constructs. The online Java tutorial may be handy while completing this and future assignments. It's available at `http://download.oracle.com/javase/tutorial/java/index.html`.

Step-by-Step

1. In Eclipse, create a new project named Lesson 3.
2. Create a new `Tax` class (File ➪ New ➪ Class). Enter the code shown in Listing 3-3.
3. Create another class, `TestCalc`, and input the code from Listing 3-4.
4. Save both classes and run `TestCalc` (right-click and select Run As ➪ Java Application). The console view should display "Your tax is 234.55," as shown in Figure 3-1.

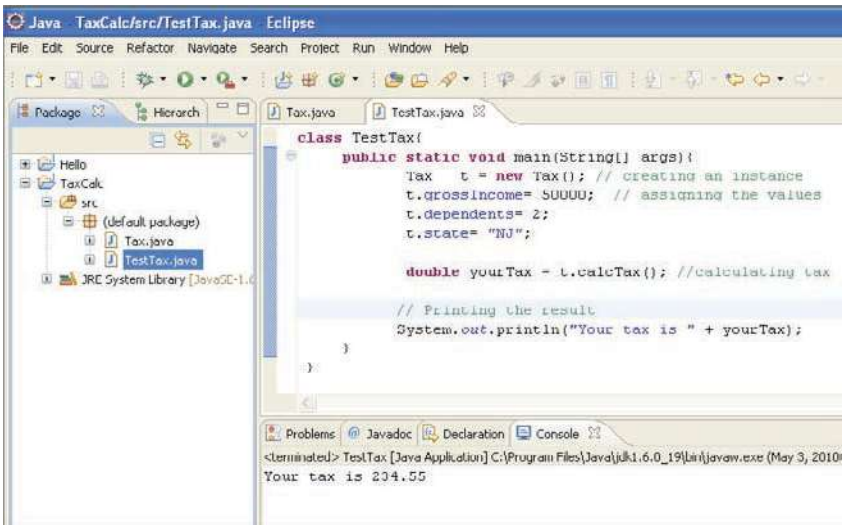


FIGURE 3-1