UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

STINGRAY GROUP INC. AND STINGRAY MUSIC USA, INC.,
PETITIONER,

v.

EDWIN A. HERNANDEZ-MONDRAGON,
PATENT OWNER.

CASE: TO BE ASSIGNED
U.S. PATENT NO. 10,524,002

DECLARATION OF JUNE ANN MUNFORD

EXHIBIT
1008

1. My name is June Ann Munford. I am over the age of 18, have personal knowledge of the facts set forth herein, and am competent to testify to the same.

2. I earned a Master of Library and Information Science (MLIS) from the University of Wisconsin-Milwaukee in 2009. I have over ten years of experience in the library/information science field. Beginning in 2004, I have served in various positions in the public library sector including Assistant Librarian, Youth Services Librarian and Library Director. I have attached my Curriculum Vitae as Appendix A to this declaration.

3. During my career in the library profession, I have been responsible for materials acquisition for multiple libraries. In that position, I have cataloged, purchased and processed incoming library works. That includes purchasing materials directly from vendors, recording publishing data from the material in question, creating detailed material records for library catalogs and physically preparing that material for circulation. In addition to my experience in acquisitions, I was also responsible for analyzing large collections of library materials, tailoring library records for optimal catalog search performance and creating lending agreements between libraries during my time as a Library Director.

4. I am fully familiar with the catalog record creation process in the library sector. In preparing a material for public availability, a library catalog record describing that material would be created. These records are typically written in Machine Readable Catalog (herein referred to as "MARC") code and contain information such as a physical description of the material, metadata from the material's publisher, and date of library acquisition. In particular, the 008 field of the MARC record is reserved for denoting the date of creation of the library record itself. As this typically occurs during the process of preparing materials for public access, it is my experience that an item's MARC record indicates the date of an item's public availability.

5. Typically, in creating a MARC record, a librarian would gather various bits of metadata such as book title, publisher and subject headings among others and assign each value to a relevant numerical field. For example, a book's physical description is tracked in field 300 while title/attribution is tracked in field 245. The 008 field of the MARC record is reserved for denoting the creation of the library record itself. As this is the only date reflecting the inclusion of said materials within the library's collection, it is my experience that an item's 008 field accurately indicates the date of an item's public availability.

6. I have reviewed Exhibit 1005, Java Programming: 24-Hour Trainer by Yakov Fain ("Fain").

7. Attached hereto as Appendix B is a true and correct copy of the MARC record for Fain as held by the Penn State University Library. I secured this record myself from the library's public catalog. The MARC record contained within Appendix B accurately describes the title, author, publisher, and ISBN number of Fain.

8. Attached hereto as Appendix C is a true and correct copy of Fain as held by the Penn State University Library. This PDF was secured from the library's collection. In comparing Exhibit 1005 to Appendix C, it is my determination that Exhibit 1005 is a true and correct copy of Fain.

9. The 008 field of the MARC record in Appendix B indicates the date of record creation. The 008 field of Appendix B shows the following:

008 101202s2011 inua s 001 0 eng d

10. Accordingly, 008 field of the MARC record in Appendix B indicates the Penn State University Library first acquired Fain as of December 2, 2010. Considering this information, it is my determination that Fain was made available to the public shortly after its initial acquisition as of December 2, 2010.

11. 260 field of the MARC record in Appendix B identifies information relating to the publication, printing, distribution, issue, release, or production of Fain. This field identifies John Wiley as the publisher and a publication date of 2011. John Wiley is a well-known publisher and the 2011 publication date indicated in field 260 matches the 2011 copyright date indicated on the face of Ex. 1005.

12. Based on the information above, it is my conclusion that Fain was published shortly after December 2, 2010, and certainly no later than some time in 2011.

13. I have been retained on behalf of the Petitioner to provide assistance in the above-illustrated matter in establishing the authenticity and public availability of the documents discussed in this declaration. I am being compensated for my services in this matter at the rate of $200.00 per hour plus reasonable expenses. My statements are objective, and my compensation does not depend on the outcome of this matter.

14. I declare under penalty of perjury that the foregoing is true and correct. I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code.

Dated: 12/16/2024



June Ann Munford

Appendix A

June A. Munford
Curriculum Vitae

Education

University of Wisconsin-Milwaukee - MS, Library & Information Science, 2009
Milwaukee, WI

- Coursework included cataloging, metadata, data analysis, library systems, management strategies and collection development.
- Specialized in library advocacy, cataloging and public administration.

Grand Valley State University - BA, English Language & Literature, 2008
Allendale, MI

- Coursework included linguistics, documentation and literary analysis.
- Minor in political science with a focus in local-level economics and government.

Professional Experience

Researcher / Expert Witness, October 2017 – present
Freelance ● Pittsburgh, Pennsylvania & Grand Rapids, Michigan

- Material authentication and public accessibility determination. Declarations of authenticity and/or public accessibility provided upon research completion. Experienced with appeals and deposition process.

- Research provided on topics of public library operations, material publication history, digital database services and legacy web resources.

- Past clients include Alston & Bird, Arnold & Porter, Baker Botts, Fish & Richardson, Erise IP, Irell & Manella, O'Melveny & Myers, Perkins-Coie, Pillsbury Winthrop Shaw Pittman and Slayden Grubert Beard.

Library Director, February 2013 - March 2015
Dowagiac District Library ● Dowagiac, Michigan

- Executive administrator of the Dowagiac District Library. Located in

Southwest Michigan, this library has a service area of 13,000, an annual operating budget of over $400,000 and total assets of approximately $1,300,000.

● Developed careful budgeting guidelines to produce a 15% surplus during the 2013-2014 & 2014-2015 fiscal years while being audited.

● Using this budget surplus, oversaw significant library investments including the purchase of property for a future building site, demolition of existing buildings and building renovation projects on the current facility.

● Led the organization and digitization of the library's archival records.

● Served as the public representative for the library, developing business relationships with local school, museum and tribal government entities.

● Developed an objective-based analysis system for measuring library services - including a full collection analysis of the library's 50,000+ circulating items and their records.

November 2010 - January 2013
Librarian & Branch Manager, Anchorage Public Library ● Anchorage, Alaska

● Headed the 2013 Anchorage Reads community reading campaign including event planning, staging public performances and creating marketing materials for mass distribution.

● Co-led the social media department of the library's marketing team, drafting social media guidelines, creating original content and instituting long-term planning via content calendars.

● Developed business relationships with The Boys & Girls Club, Anchorage School District and the US Army to establish summer reading programs for children.

June 2004 - September 2005, September 2006 - October 2013
Library Assistant, Hart Area Public Library
Hart, MI

● Responsible for verifying imported MARC records and original MARC

cataloging for the local-level collection as well as the Michigan Electronic Library.

● Handled OCLC Worldcat interlibrary loan requests & fulfillment via ongoing communication with lending libraries.

Professional Involvement

Alaska Library Association - Anchorage Chapter
- Treasurer, 2012

Library Of Michigan
- Level VII Certification, 2008
- Level II Certification, 2013

Michigan Library Association Annual Conference 2014
- New Directors Conference Panel Member

Southwest Michigan Library Cooperative
- Represented the Dowagiac District Library, 2013-2015

Professional Development

Library Of Michigan Beginning Workshop, May 2008
Petoskey, MI
- Received training in cataloging, local history, collection management, children's literacy and reference service.

Public Library Association Intensive Library Management Training, October 2011
Nashville, TN
- Attended a five-day workshop focused on strategic planning, staff management, statistical analysis, collections and cataloging theory.

Alaska Library Association Annual Conference 2012 - Fairbanks, February 2012
Fairbanks, AK
- Attended seminars on EBSCO advanced search methods, budgeting, cataloging, database usage and marketing.

Depositions

2019 ● Fish & Richardson

Apple v. Qualcomm (IPR2018-001281, 39521-00421IP, IPR2018-01282 and 39521-00421IP2)

2019 ● Erise IP

Implicit, LLC v. Netscout Systems, Inc (Civil Action No. 2:18-cv-53-JRG)

2019 ● Perkins-Coie

Adobe Inc. v. RAH Color Technologies LLC (Cases IPR2019-00627, IPR2019-00628, IPR2019-00629 and IPR2019-00646)

2020 ● O'Melveny & Myers

Maxell, Ltd. v. Apple Inc. (Case No. 5:19-cv-00036-RWS)

2021 ● Pillsbury Winthrop Shaw Pittman LLP

Intel v. SRC (IPR2020-1449)

2022 ● Perkins-Coie

Realtek v. Future Link (IPR2021-01182)

2023 ● Fish & Richardson

Neuroderm Ltd. v. Abbvie, Inc (Case No. PGR2022-00040)

2023 ● Fish & Richardson

Nearmap US Inc. v. Pictometry International Corp. (IPR2022-00735)

2023 ● Fish & Richardson

Samsung Electronics v. MemoryWeb LLC (Case No. 39843-0136PS1)

2023 ● Pillsbury Winthrop Shaw Whitman LLP

Gravel Rating Systems v. Costco Wholesale Corp. (Civil Action No. 4:21-cv-149-ALM)

2024 ● Willkie-Farr

Netflix, Inc. v. VideoLabs. Inc. (IPR2023-00628)

Limited Case and Clientele History

Alston & Bird

- Ericsson

  v. Collision Communications (IPR2022-01233)

- Nokia

  v. Neptune Subsea, Xtera (Case No. 1:17-cv-01876)

- Universal Electronics Inc

  v. Roku Inc (IPR2022-00818)

Arnold & Porter

- Ivantis

  v. Glaukos (Case No. 8:18-cv-00620)

- Samsung

  v. Jawbone (Case No. 2:21-cv-00186)

Benesch Friedlander Coplan & Aronoff

- Voyis

  v. Cathx (Case No. 5:21-cv-00077-RWS)

Deschert LLP

- Smaxtec Animal Care

  v. ST Reproductive Technologies, LLC (IPR2024-00885)

Erise I.P.

- Apple

  v. Ericsson Inc. (IPR2022-00715)

  v. Future Link Systems (IPRs 6317804, 6622108, 6807505, and 7917680)

  v. INVT (Case No. 20-1881)

  v. Navblazer LLC (IPR2020-01253)

  v. Qualcomm (IPR2018-001281, 39521-00421IP, IPR2018-01282, 39521-00421IP2)

  v. Quest Nettech Corp (Case No. 2:19-cv-00118-JRG)

  v. Telefonaktiebolaget LM Ericsson (IPR2022-00275)

  v. Theta IP, LLC (IPR2024-00818)

- Fanduel

  v. CGT (Case No. 19-1393)

- Garmin

  v. Phillips North America LLC (Case No. 2:19-cv-6301-AB-KS)

- Netscout
    - v. Longhorn HD LLC (Case No. 2:20-cv-00349)
    - v. Implicit, LLC (Civil Action No. 2:18-cv-53-JRG)

- Sony Interactive Entertainment LLC
    - v. Bot M8 LLC (IPR2020-01288)
    - v. Infernal Technology LLC (Case No. 2:19-CV-00248-JRG)

- Unified Patents
    - v. GE Video Compression (Civil Action No. 2:19-cv-248)

Fish & Richardson

- Apple
    - v. AliveCor (3:21-cv-03958)
    - v. LBS Innovations (Case No. 2:19-cv-00119-JRG-RSP)
    - v. Koss Corporation (IPR2021-00305)
    - v. Masimo (IPR 50095-0012IP1, 50095-0012IP2, 50095-0013IP1, 50095-0013IP2, 50095-0006IP1, 50095-0135IP1)
    - v. Neonode (Case No. 21-cv-08872-EMC)
    - v. Qualcomm (IPR2018-001281, 39521-00421IP, IPR2018-01282, 39521-00421IP2)

- Dell
    - v. Neo Wireless (IPR2022-00616)

- Dish Network
    - v. Entropic Communications, LLC (Case No. 2:2023-CV-01043)

v. Realtime Adaptive Streaming (Case No. 1:17-CV-02097-RBJ)

v. TQ Delta LLC (Case No. 18-1798)

- Evapco Dry Cooling

  v. SPG Dry Cooling (IPR2021-00688)

- Genetec

  v. Sensormatic Electronics (Case No. 1:20-CV-00760)

- Huawei

  v. Bell Northern Research LLC (IPR2019-01174)

- Kianxis

  v. Blue Yonder (Case No. 3:20-cv-03636)

- LG Electronics

  v. Bell Northern Research LLC (Case No. 3:18-cv-2864-CAB-BLM)

- Metaswitch

  v. Sonus Networks (IPR2018-01719)

- Microsoft

  v. Throughputer Inc (IPR2022-00757)

- Mom Enterprises

  v. Ddrops Company (Case No. 1:22-cv-00332)

- MLC Intellectual Property

v. MicronTech (Case No. 3:14-cv-03657-SI)

- Nearmap Inc
  - v. EagleView Technologies (IPR2022-01009)

- Neuroderm Ltd.
  - v. Abbvie, Inc (Case No. PGR2022-00040)

- Realtek Semiconductor
  - v. Future Link (IPR2021-01182)

- Quectel
  - v. Koninklijke Philips (Case No. 1:20-cv-01710)

- Samsung
  - v. Aire Technology (IPR2022-00877)
  - v. Bell Northern Research (Case No. 2:19-cv-00286-JRG)
  - v. Communication Technologies Inc (IPR2022-01221)
  - v. Jawbone Innovations (IPR2022-00865)
  - v. MemoryWeb LLC (IPR2022-00885)
  - v. Dodots Licensing Solutions (IPR2023-00701)
  - v. Telefonaktiebolaget LM Ericsson (IPR2021-00615)

- Texas Instruments
  - v. Vantage Micro LLC (IPR2020-01261)

- Xilinx
  - v. Sentient Sensors LCC (Case No. 1:22-cv-00173)

Groombridge Wu

- Nearmap

  v. Eagle View Technologies Inc. and Pictometry International Corp. (Case No. 2:21-cv-00283-TS-DAO)

Irell & Manella

- Curium

O'Melveny & Myers

- Apple

  v. Maxell (Case No. 5:19-cv-00036-RWS)

- Disney Media & Entertainment Distribution LLC

  v. Digital Media Technology Holdings, LLC (Case No. 2:22-cv-01642)

- Micron Technology Inc.

  v. Besang Inc (IPR2023-00900)

- Samsung

  v. Daedalus Prime LLC (1335 Investigation)

Perkins-Coie

- Amazon

  v. Nokia (IPR2024-00847 and IPR2024-00848)

- Heru Industries
  - v. The UAB Research Foundation (IPR2022-01148)
- Intel Corporation
  - v. BeSang Inc. (IPR2023-00991)
  - v. BitMICRO (Case No. 5:23-cv-00625)
  - v. Telefonaktiebolaget L M Ericsson (IPR2024-00610)
- Realtek Semiconductor
  - v. Future Link (IPR2021-01182)
- Twitter Inc
  - v. VOIP-Pal.com (Case No. 3:20-cv-02397-JD)
- TCL Industries
  - v. Koninklijke Philips NV (IPR2021-00495, IPR2021-00496 and IPR2021-00497)

Pillsbury Winthrop Shaw Pittman

- Intel
  - v. FG SRC LLC (Case No. 6:20-cv-00315)
- Gravel Rating Systems
  - v. Costco (Case No. 4:21-cv-149)
  - v. Lowe's Home Centers (Case No. 4:21-cv-150)
  - v. T-Mobile USA (Case No. 4:21-cv-152)
  - v. Kohl's Inc. (Case No. 4:21-cv-258)
  - v. Under Armor (Case No. 4:21-cv-356)

Quinn Emanuel

- ServiceNow, Inc.

  v. InQuisient, Inc. (Case No. 22-900-CJB)

- Wolfspeed

  v. The Trustees of Purdue University (Case No. 1:2021-cv-00840)

Sheppard, Mullin, Richter & Hampton LLP

- Cadence Design Systems Inc.

  v. Semiconductor Design Technologies (Case No. 3:23-cv-01001)

Stern, Kessler, Goldstein & Fox

- Preliminary Research

Willkie, Farr & Gallagher LLP

- Neurent

  v. The Foundry, LLC (IPR2024-00669)

- Netflix, Inc.

  v. VideoLabs, Inc. (IPR2023-00628)

Appendix B

MARC View

LEADER 02329nam a22003617a 4500			
001 19489713			
003 SIRSI			
005 20170114130243.0			
006 m d			
007 cr n			
008 101202s2011 inua s 001 0 eng d			
010			a| 2010942181
016	7		a| 015674011 2| Uk
020			a| 0470889640
020			a| 9780470889640
040			a| YDXCP c| YDXCP d| BWX d| GZD d| BTCTA d| NHA d| UKMGB d| BDX d| UPM d| DLC d| WaSeSS
042			a| lccopycat
050	0	0	a| QA76.73.J38 b| F345 2011
082	0	0	a| 005.2/762 2| 23
100	1		a| Fain, Yakov.
245	1	0	a| Java programming 24-hour trainer h| [electronic resource] / c| Yakov Fain.
246	3	0	a| Java programming twenty four hour trainer
260			a| Hoboken, IN : b| John Wiley, c| 2011.
300			a| xxxiv, 469 p. : b| ill. ; c| 24 cm. + e| 1 DVD-ROM (4 3/4 in.).
500			a| Includes index.
505	0		a| Note continued: Lesson Requirements -- Step-by-Step -- Lesson 36 Introduction To Hibernate Framework -- The Big Picture -- Installing and Configuring Hibernate -- Adding Hibernate Jars to an Eclipse Project -- Testing the Database Connection -- Configuring Hibernate with a DBMS -- Retrieving Data with Hibernate -- Try It -- Lesson Requirements -- Hints -- Step-by-Step -- Lesson 37 Bringing Javafx To The Mix -- Consuming Stock Quotes with JavaFX -- Code Walkthrough -- Reading JavaFX Code -- The HTML Client -- Try It -- Lesson Requirements -- Hint -- Step-by-Step -- Lesson 38 Java Technical Interviews -- Getting the Interview -- Doing Well at the Interview -- Considering the Offer -- Interviewing Enterprise Developers -- To Get or Not to Get Certified? -- Technical Questions and Answers -- Epilogue -- APPENDIX: WHAT'S ON THE DVD? -- Using the DVD on a PC -- Using the DVD on a MAC -- What's on the DVD? -- Troubleshooting -- Customer Care.
506			a| License restrictions may limit access.
520			a| Covers the concepts, terms, and programming practices of Java to create Web applications.
650		0	a| Java (Computer program language)
650		0	a| Computer software.
655		0	a| Electronic books.
856	4	0	u| http://SK8ES4MC2L.search.serialssolutions.com/?sid=sersol&SS_jc=TC0000538186&title=Java%20Programming%2024-Hour%20Trainer
949			a| Electronic resource w| ASIS c| 1 i| 19489713-1001 l| ONLINE m| ONLINE r| Y s| Y t| ONLINE u| 1/31/2017 o| .STAFF. 360 MARC

ASK A LIBRARIAN

PennState

Copyright @2019 The Pennsylvania State University. All rights reserved. Except where otherwise noted, this work is subject to a Creative Commons Attribution 4.0 license. Details and exceptions.

Legal Statements | PSU Hotlines

PENN STATE UNIVERSITY LIBRARIES

Libraries Home (814) 865-6368
Libraries Intranet (Staff Only)
Accessibility Help
Website Feedback
Policies and Guidelines
Staff Directory

CONNECT WITH PENN STATE UNIVERSITY LIBRARIES

Facebook
Twitter
Instagram



Java programming 24-hour trainer [electronic resource] / Yakov Fain

Author	Fain, Yakov
Additional Titles	Java programming twenty four hour trainer
Published	Hoboken, IN : John Wiley, 2011.
Physical Description	xxxiv, 469 p. : ill. ; 24 cm. + 1 DVD-ROM (4 3/4 in.).

ASK A LIBRARIAN

Access Online

serialssolutions.com

Search inside at Google Books

Restrictions on Access

License restrictions may limit access.

Contents

Note continued: Lesson Requirements -- Step-by-Step -- Lesson 36 Introduction To Hibernate Framework -- The Big Picture -- Installing and Configuring Hibernate -- Adding Hibernate Jars to an Eclipse Project -- Testing the Database Connection -- Configuring Hibernate with a DBMS -- Retrieving Data with Hibernate -- Try It -- Lesson Requirements -- Hints -- Step-by-Step -- Lesson 37 Bringing Javafx To The Mix -- Consuming Stock Quotes with JavaFX -- Code Walkthrough -- Reading JavaFX Code -- The HTML Client -- Try It -- Lesson Requirements -- Hint -- Step-by-Step -- Lesson 38 Java Technical Interviews -- Getting the Interview -- Doing Well at the Interview -- Considering the Offer -- Interviewing Enterprise Developers -- To Get or Not to Get Certified? -- Technical Questions and Answers -- Epilogue -- APPENDIX: WHAT'S ON THE DVD? -- Using the DVD on a PC -- Using the DVD on a MAC -- What's on the DVD? -- Troubleshooting -- Customer Care.

Summary

Covers the concepts, terms, and programming practices of Java to create Web applications.

Subject(s)

Java (Computer program language)
Computer software

Genre(s)

Electronic books

ISBN

0470889640
9780470889640

Note

Includes index.

View MARC record | catkey: 19489713

PennState

Copyright @2019 The Pennsylvania State University. All rights reserved. Except where otherwise noted, this work is subject to a Creative Commons Attribution 4.0 license. Details and exceptions.

Legal Statements | PSU Hotlines

PENN STATE UNIVERSITY LIBRARIES

Libraries Home (814) 865-6368
Libraries Intranet (Staff Only)
Accessibility Help
Website Feedback
Policies and Guidelines
Staff Directory

CONNECT WITH PENN STATE UNIVERSITY LIBRARIES

Facebook
Twitter
Instagram

Appendix C

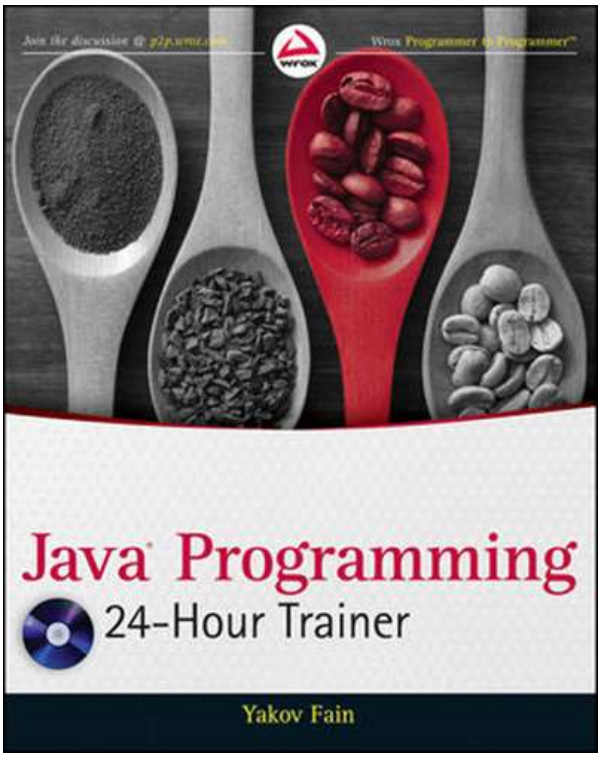
O'REILLY

BOOK

Java® Programming 24-Hour Trainer

★★★★☆ 18 reviews

By Yakov Fain



Continue

TIME TO COMPLETE:
11h 46m

LEVEL:
Intermediate to advanced

SKILLS:
Java

PUBLISHED BY:
Wrox

PUBLICATION DATE:
February 2011

PRINT LENGTH:
504 pages

+ Add to playlist

A unique book-and-video package of Java® Programming 24-Hour Trainer, presented by Java guru Yakov Fain

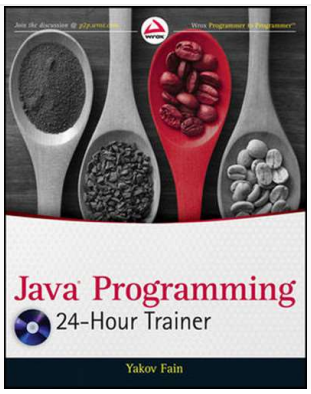
As one of the most popular software languages for building Web applications, Java is often the first programming language developers learn. The latest version includes numerous updates that both novice and experienced developers need to know. With this invaluable book-and-video package, Java authority Yakov Fain fully covers Java's new features as well as its language extensions, classes and class methods, and the Swing Application Framework. For each lesson that he discusses in the book, there is an accompanying instructional video to reinforce your learning experience.

- Features a unique book-and-video combination that starts from scratch and introduces newcomers to Java
- Integrates instructional videos with each lesson in the book, teaching by doing!
- Teaches skills businesses demand when hiring new Java developers
- Each lesson includes hands-on programming exercises to enrich your learning experience
- Shares easy-to-follow lessons, reinforced by step-by-step instructions, screen shots, and supplemental exercises so that you have a variety of learning styles from which to choose

Author is a Java authority and a "Java Champion," a title bestowed on an exclusive group of Java community leaders.

Note: CD-ROM/video and other supplementary materials are not included as part of eBook file.

You might also like

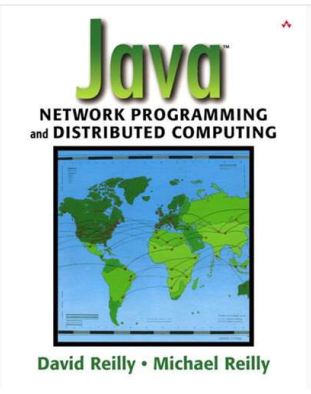


VIDEO

Java® Programming 24-Hour Trainer

With Yakov Fain

Welcome to the video portion of Java® Programmin... 24-Hour Trainer A unique book-and-video package of Java® Programming

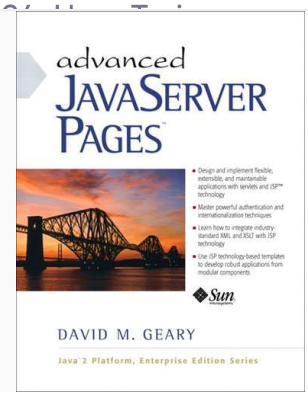


BOOK

Java™ Network Programming and... Distributed Computing

By David Reilly and Michael Reilly

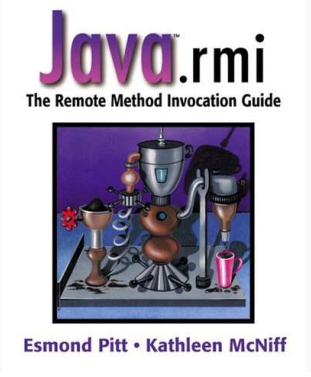
is an accessible in-troduction to the changing face of... networking theory, Java™ technology, and the fundamental elements of the Java



BOOK

Advanced JavaServer Pages™

By David M. Geary



BOOK

java.rmi: The Remote Method Invocation... Guide

By Esmond Pitt and Kathleen McNiff

Design and implement flexible, extensible, and maintainable applications

"This book collects an astonishing range of Java RMI materia... suitable for any Java programmer regard...

About the Publisher

Wrox Press, established in 1992 to publish books for computer programmers, is driven by the Programmer to Programmer philosophy. Wrox books are written by programmers for programmers, and the Wrox brand means authoritative solutions to real-world programming problems. Wrox's unique author-editorial process delivers the best and most useful information you need in the timeliest manner.

Resources

More about Wrox
Purchase book
Audio-Video
Examples

techniques that David Geary presents in Advanced JavaServer Pages. Geary tackles JavaServer Pages technology with the same exhaustive, authoritative approach that made his Graphic Java books into worldwide best sellers. He presents in-depth coverage of advanced topics including: Implementing custom JSP technology tags Designing scalable and extensible Web applications Using authentication to build secure JSP technology-based applications Internationalizing JSP technology-based applications for global environments Advanced JavaServer Pages features detailed chapters on internationalization, authentication, JSP technology templates, design, and XML. It concludes with a comprehensive case study that ties together key topics developed in the book and

from which to choose Author is a Java authority and a "Java Champion," a title bestowed on an exclusive group of Java community leaders. Note:CD-ROM/video and other supplementary materials are not included as part of eBook file.

much-just under 300 pages. An effective Distributed Computing Primer (UDP) and TCP. The book includes discussions of multithreaded applications, network protocols, and Java Implementations" The definitive guide to remote method invocation (RMI). This book provides in-depth coverage of RMI, including how to apply RMI to get two computers to communicate with each other. Learn how to write network-aware applications in Java™ Networking and Programming and Distributed Java and

demonstrates how to integrate advanced techniques based on JSP technology. This book is a must-have resource for every developer of Java technology creating server-side applications with JSP technology and servlets.

Object-oriented programming, RMI to create applications in networked environments using interfaces, remote objects, the RMI registry, servers, security, and mobile code applying advanced concepts e.g. activation, socket factories, Internet firewalls, distributed garbage collection, and design patterns working with alternatives or extensions to the standard RMI implementation e.g., JNDI naming services and CORBA IIOP. Features: Complete RMI Alamanac lift-out Extensive code samples and exercises

0201700433B07022001

Object-oriented programming, RMI to create applications in networked environments using interfaces, remote objects, the RMI registry, servers, security, and mobile code applying advanced concepts e.g. activation, socket factories, Internet firewalls, distributed garbage collection, and design patterns working with alternatives or extensions to the standard RMI implementation e.g., JNDI naming services and CORBA IIOP. Features: Complete RMI Alamanac lift-out Extensive code samples and exercises

0201700433B07022001

Companion Website, http://www.oreilly.com/pub/a/javaserver/index.html, offers downloadable source code, a list of FAQs about Java networking and RMI, and links to related resources.

0201710374B03012002

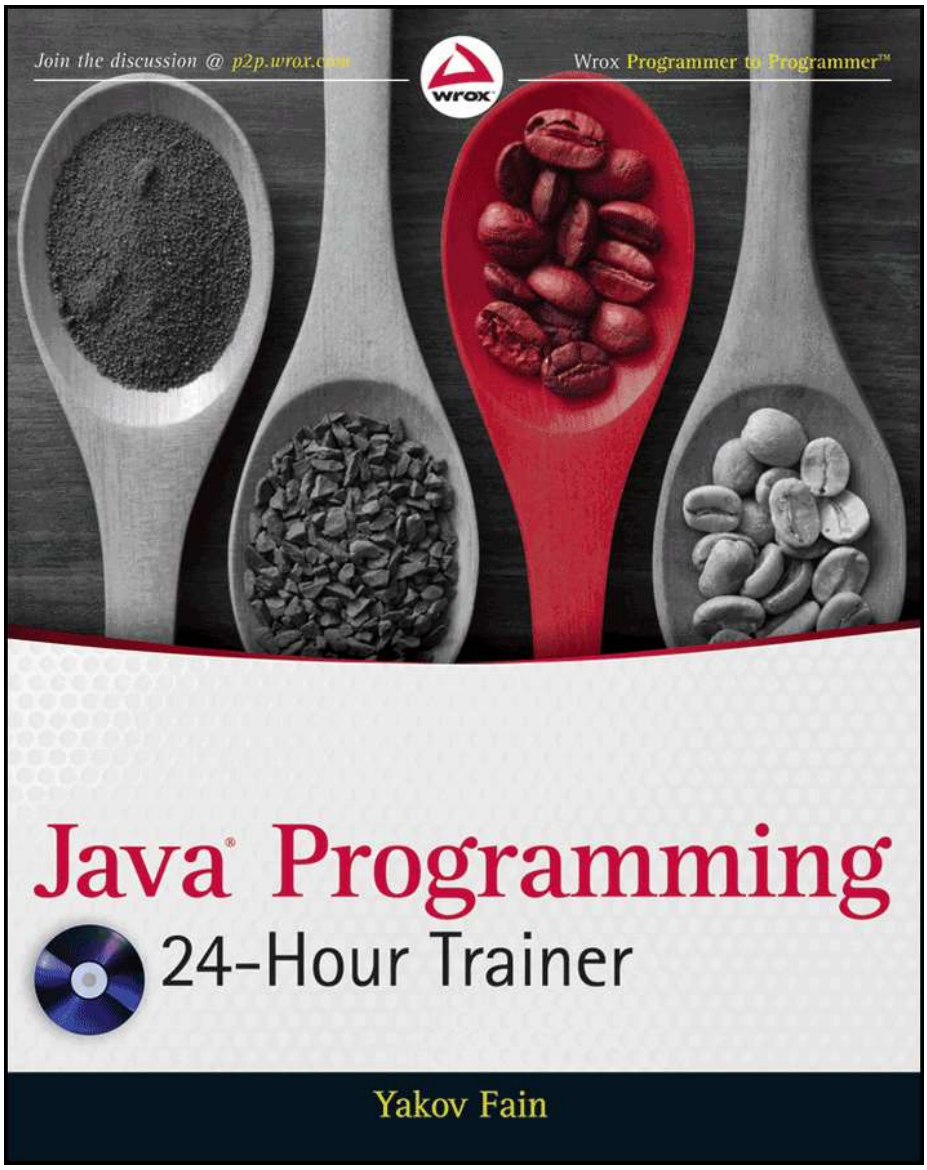

Join the discussion @ p2p.wrox.com
Wrox Programmer to Programmer™
wrox
Java® Programming
24-Hour Trainer
Yakov Fain

Table of Contents

Cover

Contents

Chapter 1: Introducing Java

Why Learn Java?

Setting the Goals

The Life Cycle of a Java Program

JDK and JRE

Java SE and EE

Downloading and Installing JDK in Windows

Your First Java Program: Hello World

Try It

Chapter 2: Eclipse IDE

Introducing Eclipse IDE

Downloading and Installing Eclipse

Creating Hello Project in Eclipse

Creating the HelloWorld Class in Eclipse

Java Packages

Completing Code Generation

Try It

Chapter 3: Object-Oriented Programming

Classes and Objects

Variables, Constants, and Data Types

Variable Scope

Wrappers, Autoboxing, and Unboxing

Program Comments

Try It

Chapter 4: Class Methods

Method Arguments

Method Overloading

Constructors

The Keyword super

The Keyword this

Passing by Value or by Reference

Variable Scopes

The Keyword static

Try It

Chapter 5: Back to Java Basics

Arrays

Loops

Debugging Java Programs

More about if and switch Statements

Command-Line Arguments

Try It

Chapter 6: Packages, Interfaces, and Encapsulation

Java Packages

Encapsulation

The Keyword final

Interfaces

Casting

Try It

Chapter 7: Programming with Abstract Classes and Interfaces

Abstract Classes

Polymorphism

Interfaces versus Abstract Classes

Try It

Chapter 8: Introducing the Graphic User Interface

Swing Basics

Layout Managers

Swing GUI Builders

Try It

Chapter 9: Event Handling in UI

Introduction to Event Listeners

Teaching the Calculator to Calculate

More Swing Listeners

How to Use Adapters

Inner Classes

Try It

Chapter 10: Introduction to Java Applets

An Unofficial History of Java Applets

Restrictions of Java Applets

Learning HTML on the Run

Writing Applets Using Swing

Try It

Chapter 11: Developing a Tic-Tac-Toe Applet

The Strategy

Coding Tic-Tac-Toe

Try It

Chapter 12: Developing a Ping-Pong Game

The Strategy

The Code

Finishing the Ping-Pong Game

Try It

Chapter 13: Error Handling

Stack Trace

Java Exceptions

Exception Hierarchy

try/catch Blocks

The throws Clause

The finally Clause

The throw Operator

Creating Your Own Exceptions

Try It

Chapter 14: Introduction to Collections

Arrays Revisited

Classes ArrayList and Vector

Collection Interfaces from java.util

Classes Hashtable and HashMap

Enumeration and Iterator

Class LinkedList

Class BitSet

Try It

Chapter 15: Introduction to Generics

Generics with Classes

Defining Generics

Wild Cards

Bounded Wild Cards

Generic Methods

What to Read Next

Try It

Chapter 16: Working with Streams

Byte Streams

Buffered Streams

Character Streams

Data Streams

The Class File

Try It

Chapter 17: Java Serialization

The Class ObjectOutputStream

The Class ObjectInputStream

The Interface Externalizable

Class Versioning

Serializing into Byte Arrays

Try It

Chapter 18: Network Programming

Reading Data from the Internet

Connecting through HTTP Proxy Servers

How to Download Files from the Internet

The Stock Quote Program

Socket Programming

Try It

Chapter 19: Processing E-Mails with Java

Protocols and Servers

Creating Birthday Mailer

How to Retrieve E-Mails

Useful Open-Source Components

Try It

Chapter 20: Introduction to Multi-Threading

The Class Thread

The Interface Runnable

Sleeping Threads

How to Kill a Thread

Thread Priorities

Thread Synchronization and Race Conditions

Thread States

Wait and Notify

Try It

Chapter 21: Digging Deeper into Concurrent Execution

Joining Threads

Goodies from java.util.concurrent

SwingWorker Thread

Try It

Chapter 22: Working with Databases Using JDBC

JDBC Driver Types

Creating a Database with Derby

Sample JDBC Program

Processing Result Sets

The PreparedStatement Class

The CallableStatement Class

The ResultSetMetaData Class

Scrollable Result Sets and RowSet

Transactional Updates

Connection Pools and DataSources

Try It

Chapter 23: Swing with JTable

JTable and the MVC Paradigm

The Model

Introduction to Renderers

Summary

Try It

Chapter 24: Annotations and Reflection

Java Annotations Basics

Custom Annotations

Reflection

Run-Time Annotation Processing

Summary

Try It

Chapter 25: Remote Method Invocation

Developing Applications with RMI

Try It

Chapter 26: Java EE 6 Overview

The Big Picture

Profiles and Pruning

Miscellaneous Concepts

Try It

Chapter 27: Programming with Servlets

The Big Picture

The Thin Client

How to Write a Servlet

How to Deploy a Servlet

Installing the GlassFish Plug-In for Eclipse

How to Create a Servlet with Eclipse

Browser-Servlet Data Flow

HTTP Get and Post Requests

Session Tracking

Event Listeners

Asynchronous Servlets

Try It

Chapter 28: JavaServer Pages

Embedding Java Code into HTML

Implicit JSP Objects

Overview of the JSP Tags

Error Pages

JavaBeans

Loading JSP from Servlets

Tag Libraries

JSTL

Try It

Chapter 29: Developing Web Applications with JSF

The Big Picture

Managed Beans

Creating a JSF Website

Converters and Validators

Try It

Chapter 30: Introducing JMS and MOM

Messaging Concepts and Terminology

Two Modes of Message Delivery

JMS API Overview

Types of Messages

How to Send a Message

How to Receive a Message

How to Publish a Message

How to Subscribe for a Topic

Message Selectors

Administering Objects in Open MQ

Try It

Chapter 31: Introducing JNDI

Java Naming and Directory Interface

Administering JNDI Objects in GlassFish

Accessing the GlassFish Naming Service with JNDI

Injection of JNDI Resources

DataSource and JNDI

Lightweight Directory Access Protocol

Try It

Chapter 32: Introduction to Enterprise JavaBeans

Who Needs EJB Containers?

Types of EJBs

Stateless Session Beans

Stateful Session Beans

Singleton Beans

Deploying EJB

Message-Driven Beans

Timer Service

Summary

Try It

Chapter 33: Introduction to the Java Persistence API

The Big Picture

Mapping Objects to Database Tables

JPQL

EntityManager

A Brief Introduction to the Criteria API

Bean Validation

Try It

Chapter 34: Working with RESTful Web Services

The SOAP Web Services

The RESTful Web Services

The RESTful Stock Server

Try It

Chapter 35: Introduction to Spring MVC Framework

Bean Wiring

Overview of Spring MVC

Processing HTML with Spring

Try It

Chapter 36: Introduction to Hibernate Framework

The Big Picture

Installing and Configuring Hibernate

Retrieving Data with Hibernate

Try It

Chapter 37: Bringing JavaFX to the Mix

Consuming Stock Quotes with JavaFX

Code Walkthrough

Try It

Chapter 38: Java Technical Interviews

Getting the Interview

Doing Well at the Interview

Considering the Offer

Interviewing Enterprise Developers

To Get or Not to Get Certified?

Technical Questions and Answers

Epilogue

Appendix: What's on the DVD?

Using the DVD on a PC

Using the DVD on a Mac

What's on the DVD?

Troubleshooting

Customer Care

Introduction

Who This Book Is For

What This Book Covers

How This Book Is Structured

What You Need to Use This Book

How to Read This Book

Conventions

Source Code

Errata

p2p.wrox.com

End-User License Agreement

Download CD/DVD Content

Chapter 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).

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.



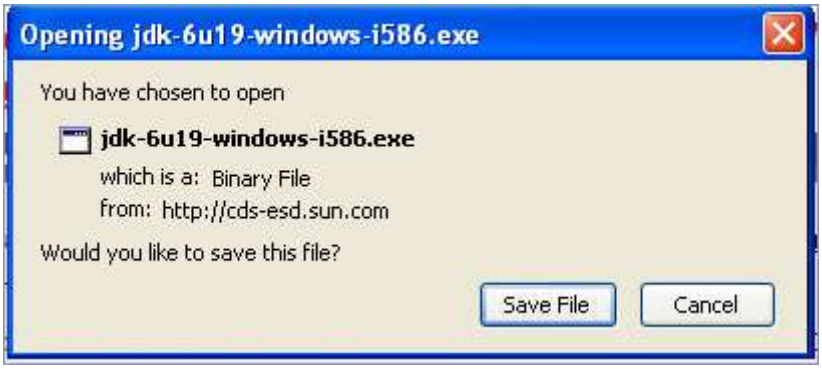
JDK for Mac OS X is preinstalled on Apple's computers and no additional installing is required. The current version of JDK, which comes with computers running Snow Leopard, is JDK 6. It's located in directory /Library/Java. In this book we'll be using Java for the Windows OS, but Java is cross-platform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

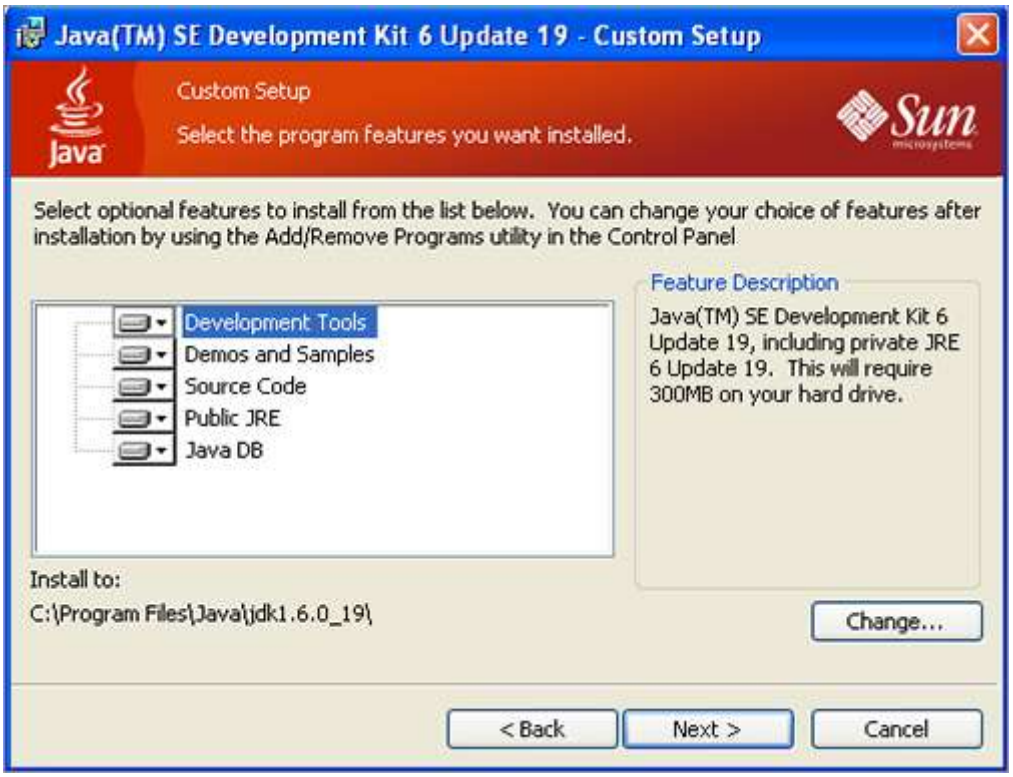
Figure 1-1



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

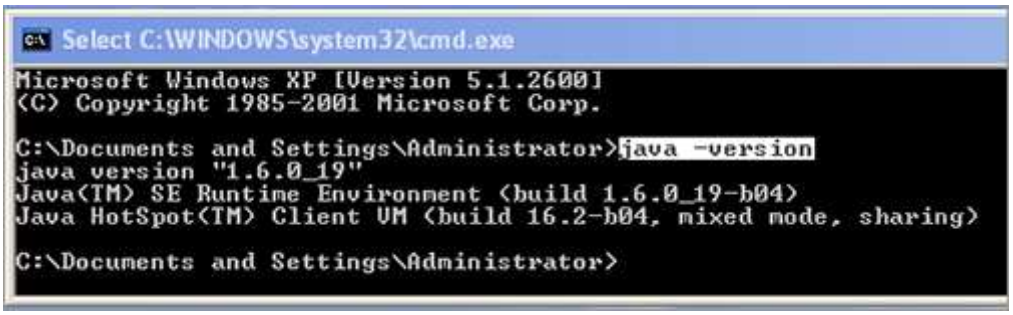


A couple of minutes into the installation process you'll see a pop-up window asking you where to install JRE. I'll assume that you've accepted the default directory (`c:\Program Files\Java\jre6`), but you can select a different one. Shortly, you should see a message telling you that installation was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

Available for download on Wrox.com

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.

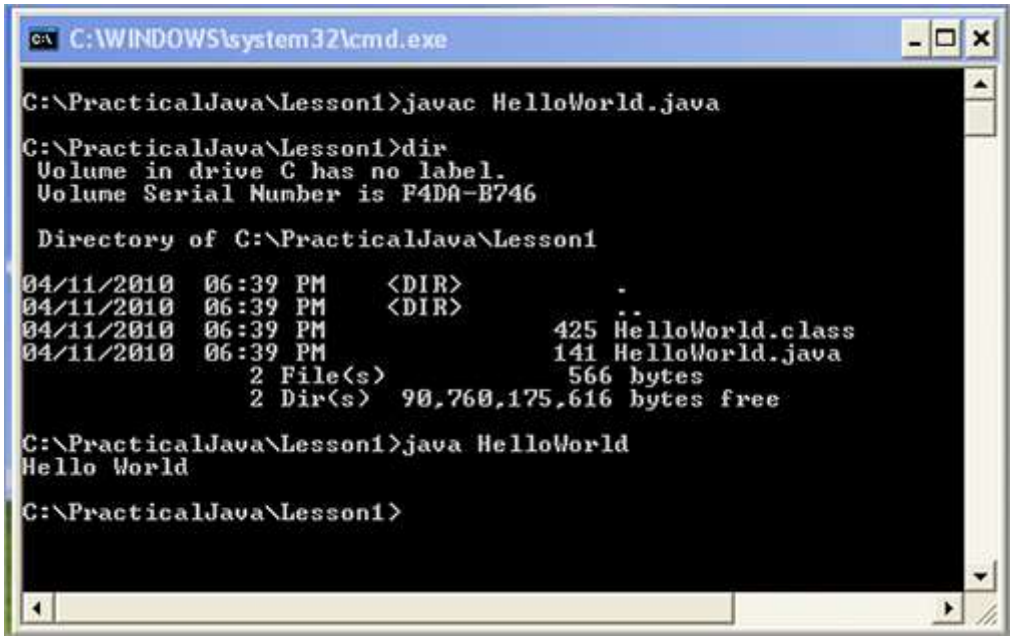
If the program has syntax errors, the compiler prints error messages. In this case fix the errors, and recompile the program again. You may need to do it more than once until the file `HelloWorld.class` is successfully created.

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.

Chapter 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

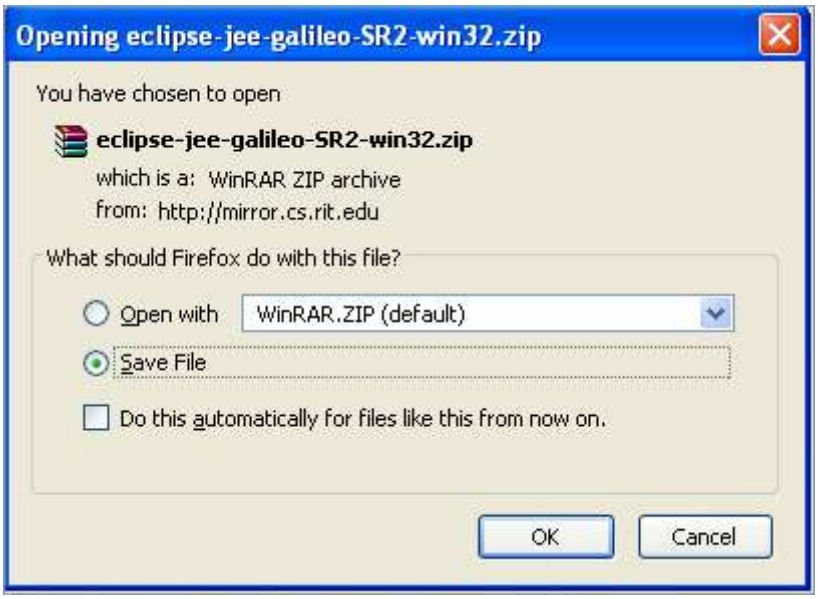
Eclipse IDE is an open-source product that was originally created with a substantial code donation by IBM to the Java community, and from that moment Eclipse was a 100% community-driven product. It started as an IDE for developing Java programs, but today it's a development platform used for building thousands of tools and plug-ins. Some people are using its Rich Client Platform (RCP) API to develop user inter-

faces (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

Figure 2-1



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).

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.

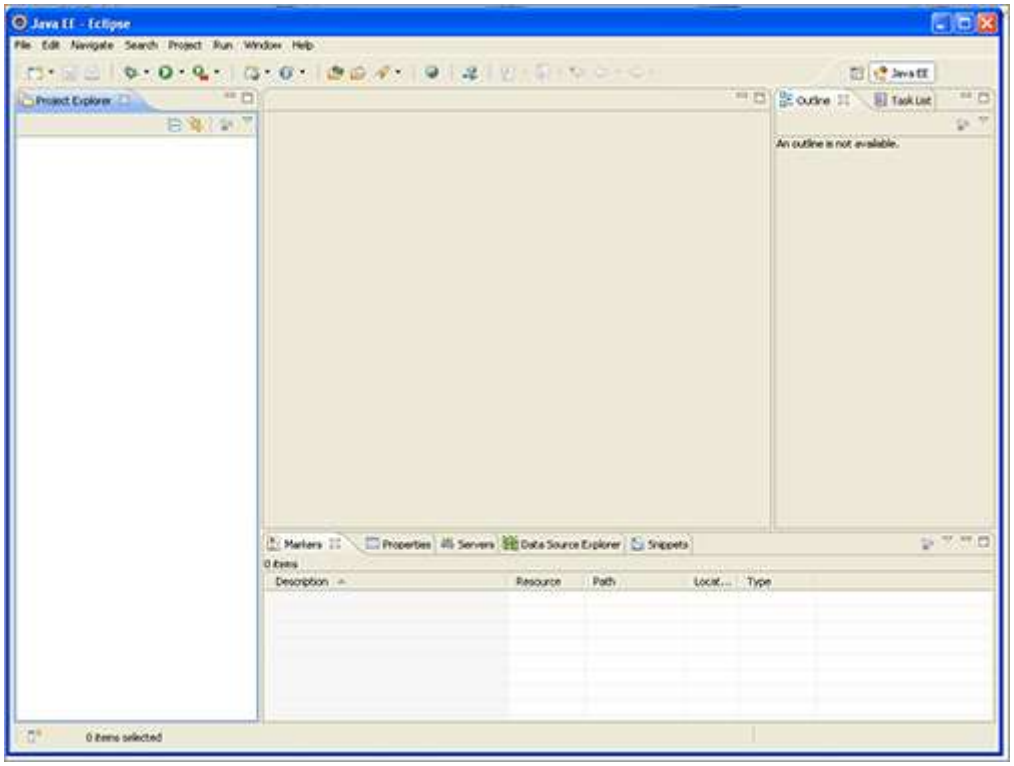
If you want to create an application in Eclipse, you start with creating a project. In the real world, the source code of a decent-size application can consist of several Eclipse projects.

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



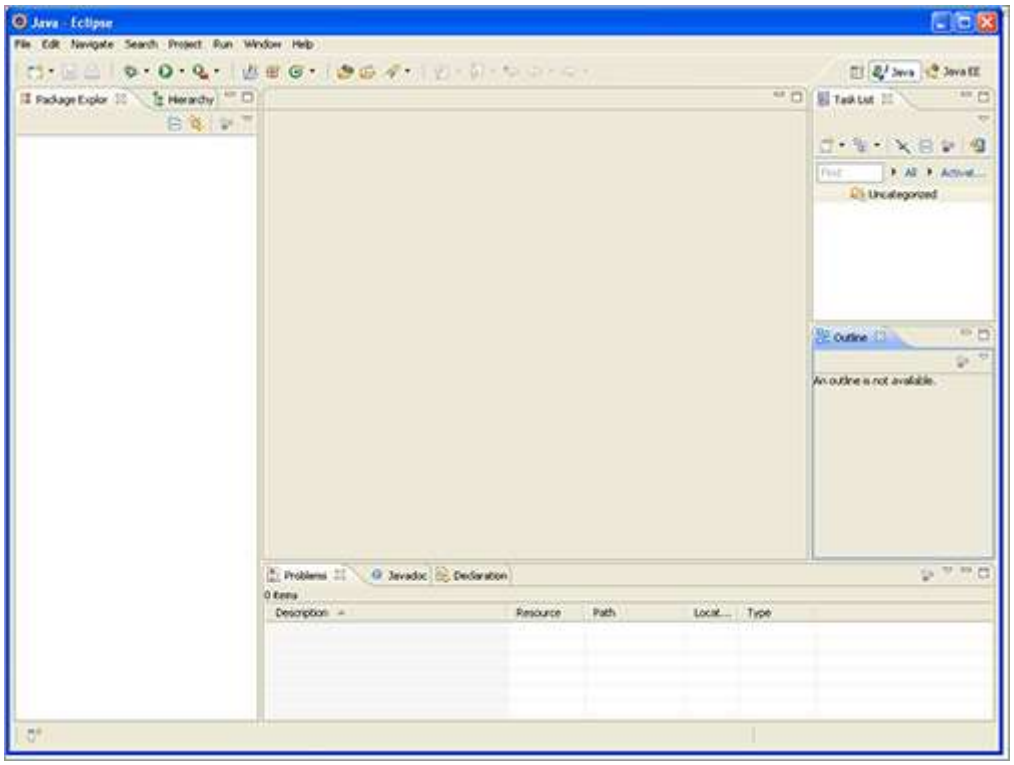
To be precise, you are looking at Java EE **perspective** (note the Java EE tab at the top), which is a collection of **views**. On the left you see a Project Explorer view. The area in the middle is reserved for the code editor view — you start using it as soon as you create your first Java

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**.

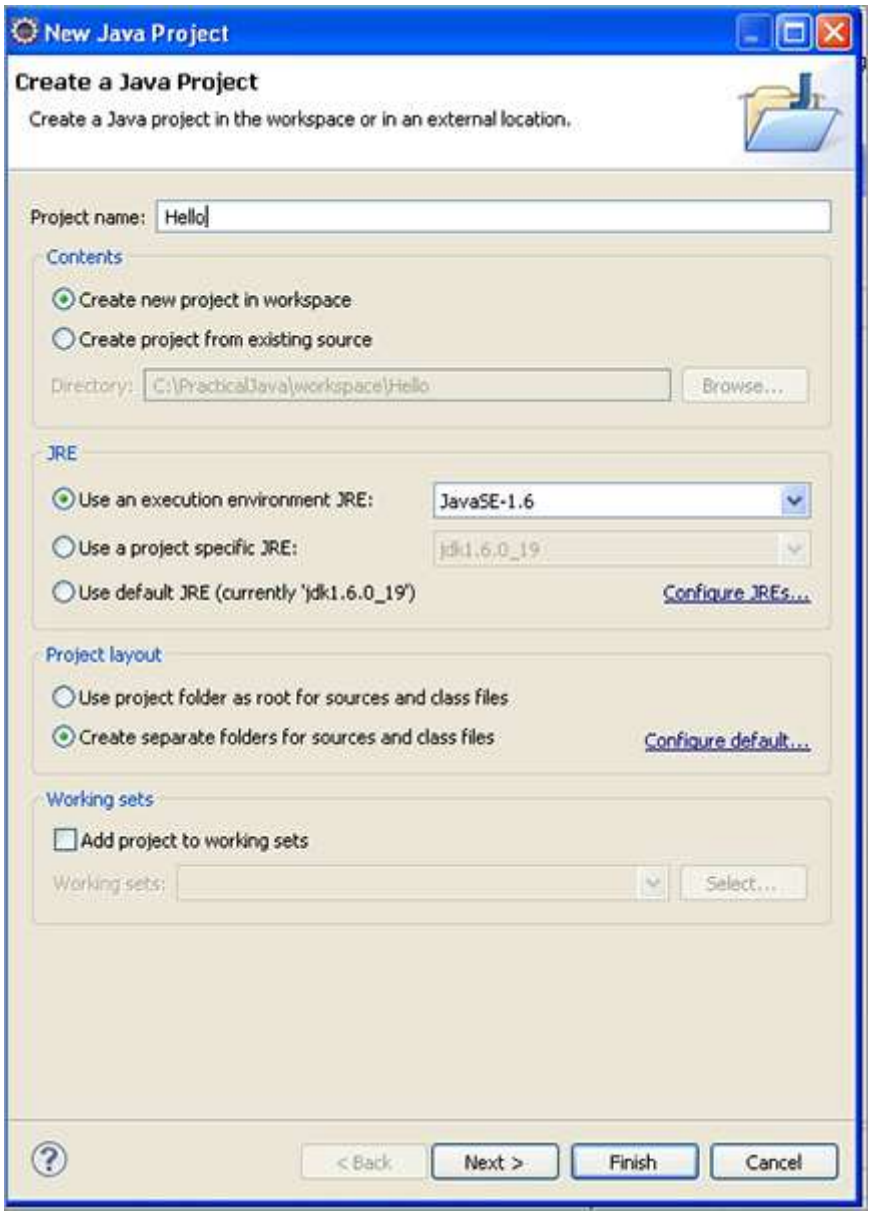
Figure 2-3



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**.

Figure 2-4

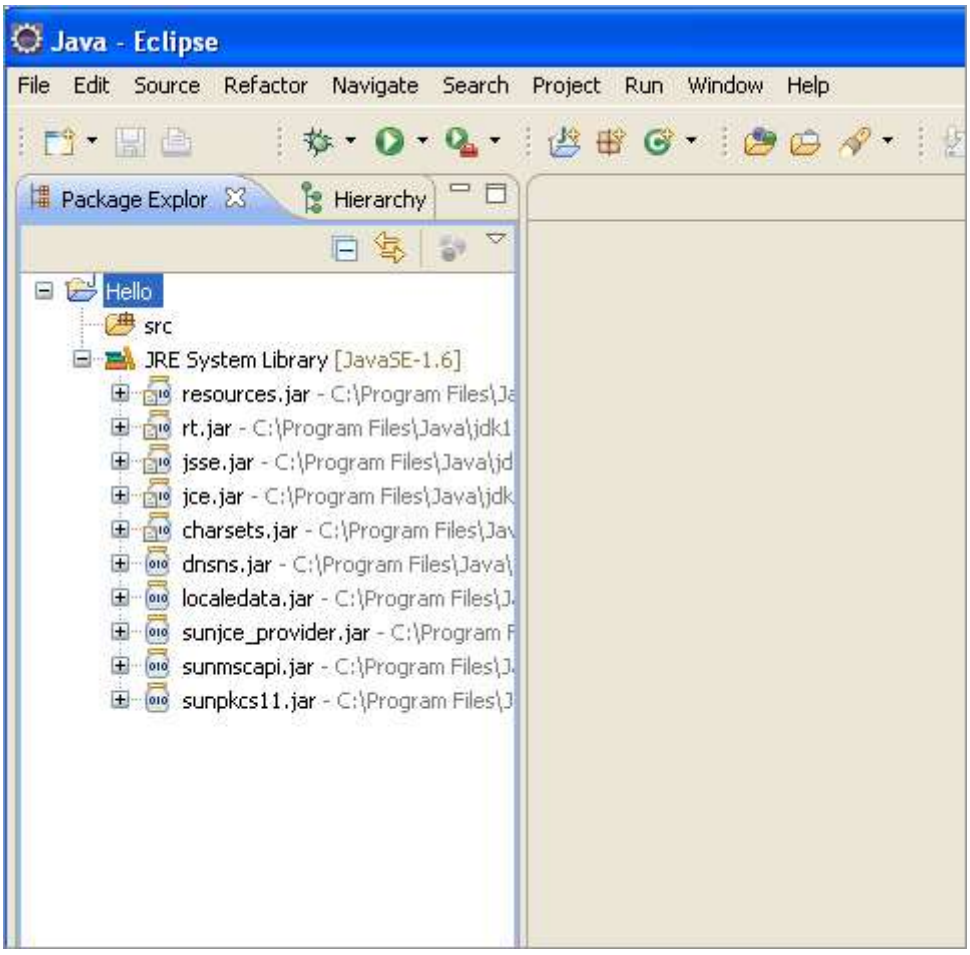


You can select the version of the JRE you want to work with. In Lesson 1 I've installed the JDK and JRE of version 1.6.0_19, but you may have more than one version of JRE, and Eclipse can compile the code that will run in another version of JRE. This is a pretty nice feature of Eclipse IDE — it doesn't 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.

Figure 2-5



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.

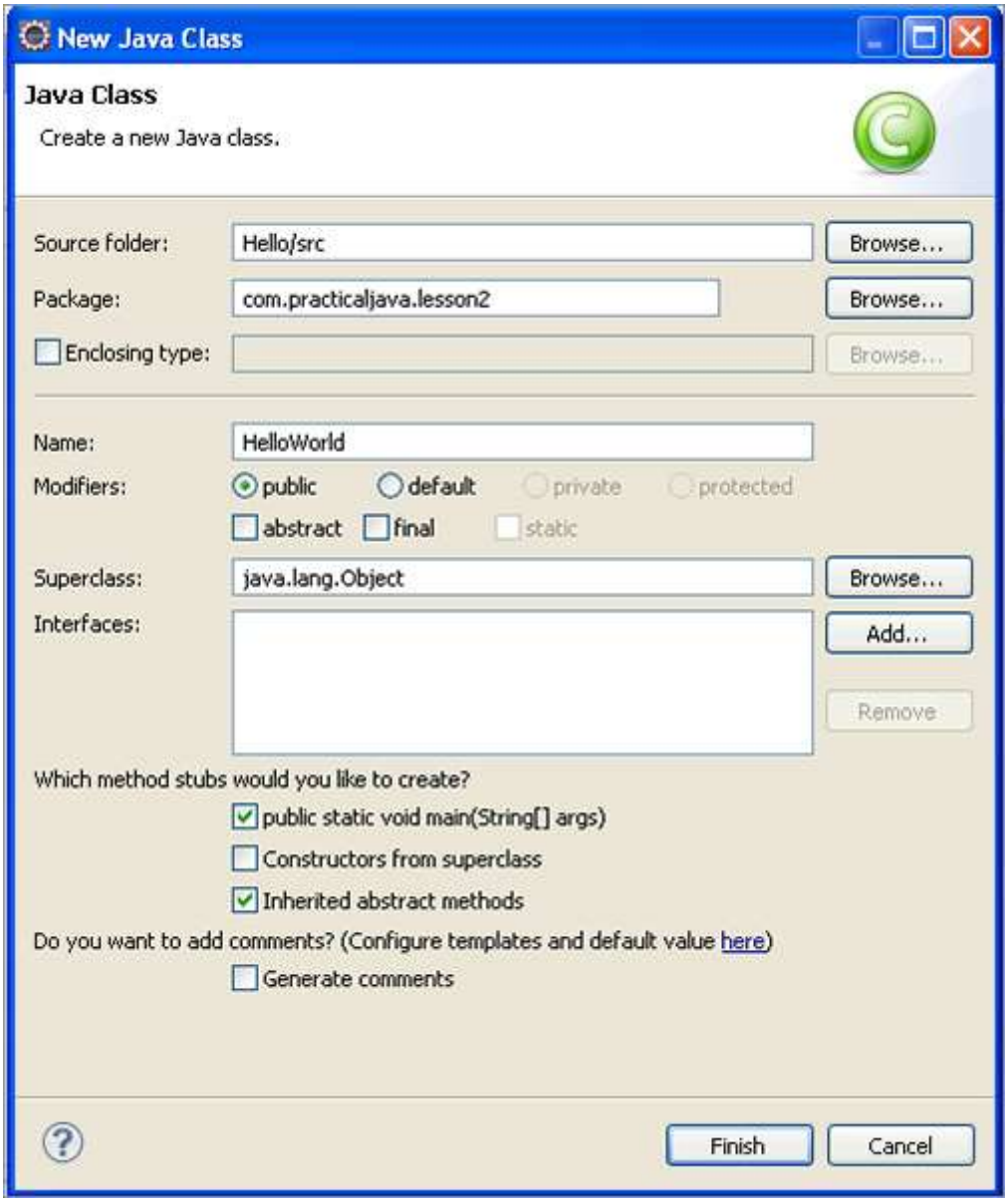
This project has an empty folder, `src` — you can save the source code of `HelloWorld.java` there when ready. The JRE folder contains all required libraries supporting the JVM where HelloWorld will run.

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.

Creating the HelloWorld Class in Eclipse

Figure 2-6



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`.

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**.

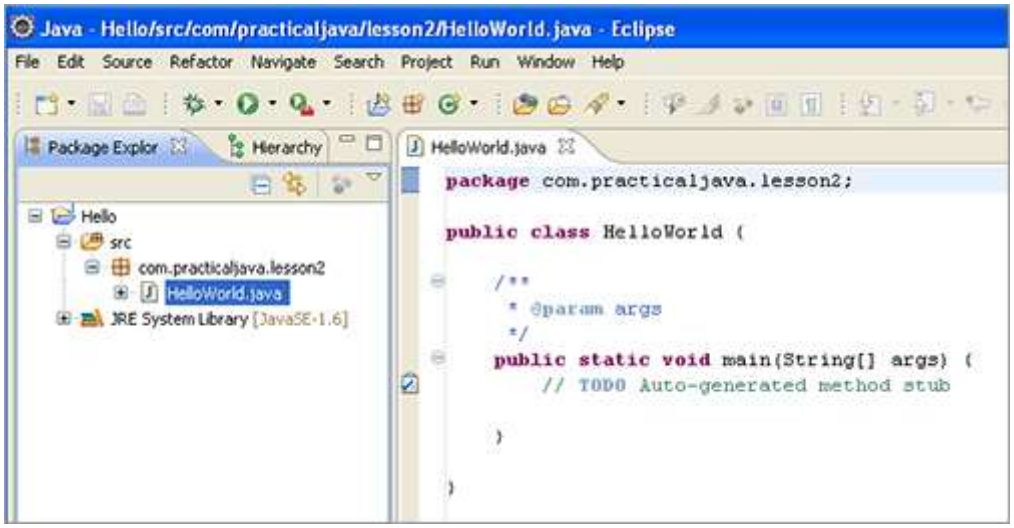
Besides being used for better organization of Java classes, packages help in controlling data access. You learn about this feature in the section "Access Levels" in Lesson 6.

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

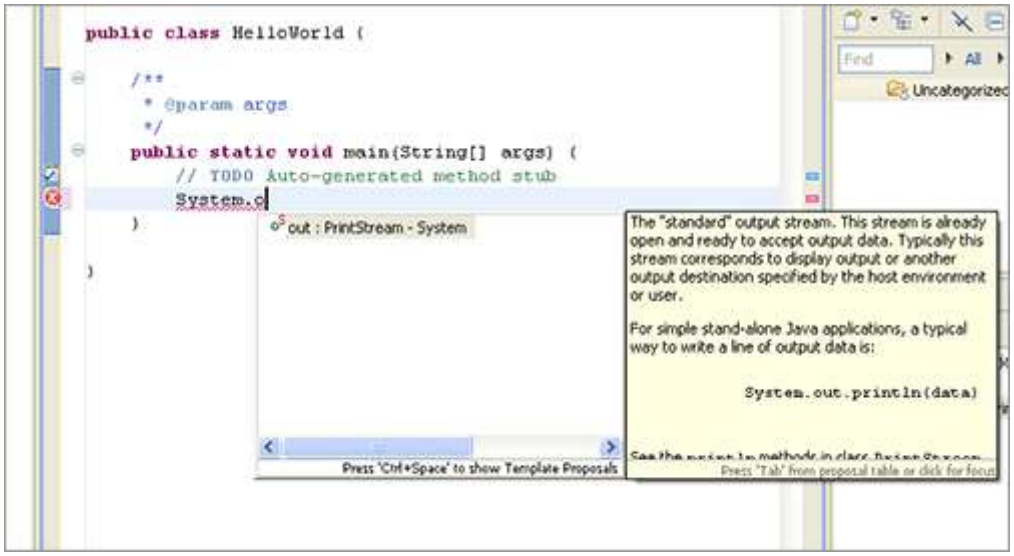


The generated code is shown in Eclipse's editor view. It starts with the `package` statement, and the class declaration with the method name goes next.

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`:

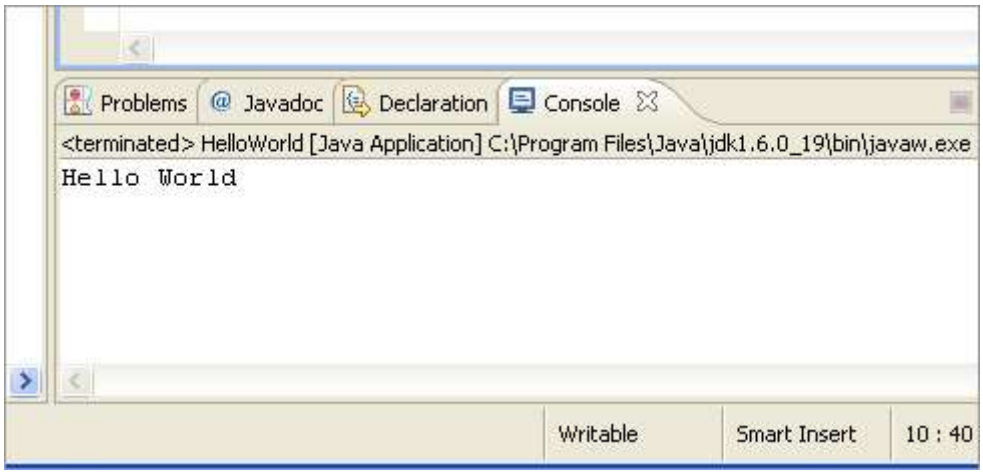
Figure 2-8



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.

Chapter 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;
      // 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  = 4637283648392l;  // 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 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.

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

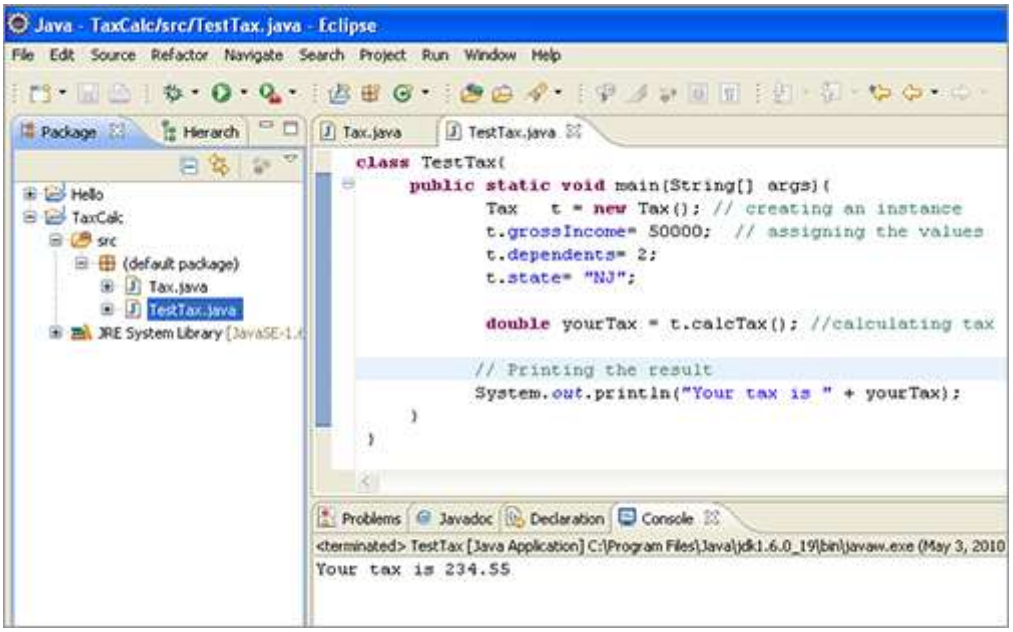
1. In Eclipse, create a new project named Lesson 3.

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



5. Replace the return of a hard-coded value with some tax calculations. Let's say that if the gross income was less than $30,000 you will deduct 5 percent for state tax. If it's greater than $30,000 you will deduct 6 percent. Modify the code of `calcTax` as follows. Run the program several times, modifying the values of the class variables of the class `Tax`. Make sure that the tax value on the console is properly calculated.

```
public double calcTax() {
      double  stateTax=0;
      if (grossIncome < 30000) {
        stateTax=grossIncome*0.05;
      }
      else{
        stateTax= grossIncome*0.06;
      }
        return stateTax;
}
```

6. Create the `NJTax` class shown in **Listing 3-5**.

7. Change the functionality of `calcTax()`by overriding it in `NJTax`. The new version of `calcTax()`should lower the tax by $500 before returning the value.

8. Modify the code of the `TestTax` class to instantiate `NJTax` instead of `Tax`. Observe that the $500 deduction is properly calculated.

To get the sample database files you can download Lesson 3 from the book's website at **www.wrox.com**.



Please select Lesson 3 on the DVD with the print book, or watch online at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.

Chapter 4

Class Methods

Methods contain code for actions or functions a class can perform. In this lesson you'll learn how to declare and call them.

Method Arguments

Each class method performs some functionality, such as calculating tax, placing an order, or starting the car. If a method requires some external data to perform its function, such data can be provided in the form of arguments or parameters, as in the function `adjustForStudents()` shown in Listing 3-5, which has one argument: `stateTax`.

In the method signature you need to declare the data type and the name of each argument, for example:

```
int calcLoanPayment(int amount, int numberOfMonths, String state){
   // Your code goes here
}
```

When you call a function, Java run time tries to find the function that has been declared with the specified signature. For example, if you try to call the preceding function like this:

```
calcLoanPayment(20000, 60);
```

Java compiler will give you an error complaining that no `calcLoanPayment()` function has been found that expects just two arguments.

Method Overloading

If you want to allow the calling of a function with different numbers of arguments, you need to create multiple versions of this function. For example, you can create a function that will use the state of New York to spare developers from providing the state as an argument. If most of the loan calculation is done for New Yorkers, such a function may be a good idea.

```
int calcLoanPayment(int amount, int numberOfMonths){
   // Your code for New York loan calculation goes here
}
```

Method overloading means having a class with more than one method having the same name but different argument lists. A method can be overloaded not only in the same class but in a descendant too. For example, the class `LoanCalulator` can have the function `calcLoanPayment()` defined with three arguments, while its descendant `MyLoanCalculator` may have a two-argument version of `calcLoanPayment()`.

Why overload methods in your classes? To provide users of these classes with a more flexible application program interface (API). In Lesson 1 you use the JDK function `println()` declared in the class `PrintStream` (see **Figure 4-1** or its JavaDoc description at **http://java.sun.com/javase/6/docs/api/java/io/PrintStream.html**). The function

`println()` has been overloaded there to give Java developers the freedom to call "the same" method with different types of arguments. In reality they are calling different methods with the same name.

Figure 4-1

void	**println()** Terminates the current line by writing the line separator string.
void	**println**(boolean x) Prints a boolean and then terminate the line.
void	**println**(char x) Prints a character and then terminate the line.
void	**println**(char[] x) Prints an array of characters and then terminate the line.
void	**println**(double x) Prints a double and then terminate the line.
void	**println**(float x) Prints a float and then terminate the line.
void	**println**(int x) Prints an integer and then terminate the line.
void	**println**(long x) Prints a long and then terminate the line.
void	**println**(Object x) Prints an Object and then terminate the line.
void	**println**(String x) Prints a String and then terminate the line.

Constructors

When a program creates an instance of a class, Java invokes the class's **constructor** — a special method that is called only once when the instance is being built with the operator `new`:

```
Tax t = new Tax();
```

Parentheses in the preceding code snippet mean that this code calls a no-argument constructor on the class `Tax`. Even if you didn't declare a constructor on a class, Java will create a no-argument constructor for you.

Constructors have the following characteristics:

- They are called when the class is being instantiated.
- They must have the same name as the class they're in.
- They can't return a value and you don't specify the keyword `void` as a return type.

Typically constructors are used to assign initial values to class variables, and in the next code snippet a three-argument constructor is defined:



Listing 4-1: Class Tax with constructor

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

    // Constructor
    Tax (double gi, String st, int depen){
       grossIncome = gi;  // class variable initialization
       state = st;
       dependents=depen;
    }
}
```

Creating an instance of this class can look like this:

```
Tax t = new Tax(65000,"NJ",3);
```

Note the difference in the initialization of the class variables: Here you pass the values during the class instantiation via the constructor's arguments, while in Listing 3-4 it took four lines of code to create an instance of `Tax` and then initialize the variables. If a constructor with arguments has been defined in a class, you can no longer use a default no-argument constructor — you have to explicitly write one if needed.

The code snippet above declares the variable `t` that points at an instance of the object `Tax` in memory. To refer to any specific field or call a method on this instance, you'll need to use so-called dot-notation — the name of the reference variable followed by a dot and the name of the field or a method. For example,

```
Public static void main(){
...
Tax t = new Tax(65000,"NJ",3);
t.dependents = 4;  // changing the number of dependents from 3 to 4
...
```

The Keyword super

If a method is overridden in a subclass, there are two versions of the method with the same signature. If you just call a method by name, e.g. `calcTax()` in the class `NJTax` from Lesson 3, the JVM will call the overridden version of the method. Once in a while you may need to call the ancestor's version of a method. The keyword `super` enables you to explicitly call the method or a constructor from the ancestor's class, for example `super.calcTax()`.

If one class is inherited from another, each of them may have its own constructor explicitly defined. As with any other class method, a constructor of a subclass can override the constructor of a superclass. But sometimes you may need to add into the subclass's construction some

functionality that has to be called after the ancestor's constructor code. In this case just add the explicit call to the constructor of a superclass, as shown in **Listing 4-2**. Invocation of the constructor of the superclass must be the first line in the constructor.



Listing 4-2: Calling the constructor of the ancestor

```
class SmallerTax extends Tax{
    // Constructor
    SmallerTax (double gi, String st, int depen){
       super(gi,st,depen);
       System.out.println("Applying special tax rates for my friends.");
    }
}
```

The Keyword this

The keyword `this` is useful when you need to refer to the instance of the class from its method. Review the code of the constructor from **Listing 4-1**. The names of the constructor's arguments were different from the names of the class variables. But the code in **Listing 4-3** shows how you can use the same variable names, both in the arguments and in the class variables. The keyword `this` helps to resolve name conflicts. To instruct JVM to use the instance variable `grossIncome`, use the following syntax:

```
this.grossIncome = 50000;
```

If there were only one `grossIncome` variable in the class `Tax`, you could simply omit the `this` prefix. But in **Listing 4-3** the absence of the `this` keyword would lead to ambiguity and the instance variable would never be initialized.



Listing 4-3: Resolving name conflicts with the this keyword

```
class Tax {
    double grossIncome; // class member variables
    String state;
    int dependents;

    // Constructor
    Tax (double grossIncome, String state, int dependents){
       this.grossIncome = grossIncome;  // instance variable initialization
       this.state = state;
       this.dependents=dependents;
    }
}
```

Consider a class called `SomeOtherClass` with a method defined as `verifyTax( Tax t)`. As you can see, it expects an instance of `Tax` as an argument. **Listing 4-4** shows how you can call it from the class `Tax` using the keyword `this` to pass a reference to the current instance of the class `Tax`.



Listing 4-4: Calling an overloaded constructor with the keyword this

```
class  Tax {
     void verifyTax(){
```

```
        SomeOtherClass s = new SomeOtherClass();
        s.verifyTax(this);
     }
}
```

Here's another use case — a class has several overloaded constructors with different numbers of arguments. You can use the `this()` notation to call a specific version of the constructor. In **Listing 4-5** the second constructor invokes the first one.



Listing 4-5: Calling an overloaded constructor with the keyword this

```
class Tax {
    double grossIncome; // class member variables
    String state;
    int dependents;

    // First Constructor
    Tax (double grossIncome, String state, int dependents){
       this.grossIncome = grossIncome;  // instance variable initialization
       this.state = state;
       this.dependents=dependents;
    }
   // Second Constructor
    Tax (double grossIncome, int dependents){
       this(grossIncome, "NY", dependents);     }

}
```

Passing by Value or by Reference

Calling a method with arguments enables you to pass some required data to the method. The question is how JVM passes these values to the

method. Does it create a copy of a variable in a calling program and give it to the method?

The primitive values are passed by value (meaning that an extra copy will be created in memory for each variable). If you create an instance of `Tax` as in **Listing 4-6**, there will be two copies of `grossIncome` and two copies of the variable `dependents` — one in `TestTax` and the other one in `Tax`.



Listing 4-6: The TestTax class

```
class TestTax{
     public static void main(String[] args){
            double grossIncome; // local variables
            String state;
            int dependents;

            grossIncome= 50000;
            dependents= 2;
            state= "NJ";

            Tax   t = new Tax(grossIncome, state, dependents);

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

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

In the preceding example, if you'll be changing the value of `grossIncome` or `dependents` in the constructor of the class `Tax`, it won't affect the values in the corresponding variables of the class `TestTax` because there will be two copies of these primitives.

Now consider another example. I'll declare another variable of type `Tax` and will assign the value of `t` to it:

```
Tax   t2 = t;
```

The variable `t` is pointing to an instance of the object `Tax` in memory. In other words, the variable `t` holds the reference (the address in memory) to an object. The code line above will not create another copy of the `Tax` object in memory, but will copy its address to the variable `t2`. Now we still have a single instance of the `Tax` object, but now two reference variables — `t` and `t2` — are pointing at it. Until both of these variables will go out of scope (explained in the next section), the object `Tax` will not be removed from memory.

The process of removal of unused objects from memory is called garbage collection (GC). JVM runs GC automatically.

The code in **Listing 4-4** passes the argument's value differently. It seems that it passes the entire instance of the object `Tax` to the method `verifyTax()`. In reality, though, another copy of just the reference variable pointing at the `Tax` instance will be created in `SomeOtherClass`, but the instance of the class `Tax` will remain the same.

This means that if the code in `SomeOtherClass` will be changing some properties of the `Tax` instance, the changes will be applied to the only copy of the `Tax` instance and will be visible from both `Tax` and `SomeOtherClass`.

Variable Scopes

Variable scope defines how long the variable will live and remain usable. If you declared the variable inside a method, it's a local variable that will go out of scope (become unavailable) as soon as the method

finishes its execution. For example, variables `t`, `grossIncome`, `dependents`, and `state` from **Listing 4-6** are local.

If variables have been declared outside the method (such as `grossIncome`, `dependents`, and `state` in **Listing 4-1**) they are **class variables** and can be used by any method of this class. On the other hand, the variables `grossIncome`, `dependents`, and `state` in **Listing 4-1** are also **instance variables** and store instance-specific data.

You can create more than one instance of the class `Tax`, and each instance can have different values in its instance variables. For example, the following lines of code will create two instances of `Tax`, with different values for `grossIncome`, `dependents`, and `state` (these instance variables are initialized in the constructor):

```
Tax   t1 = new Tax(50000, "NY", 3 );
Tax   t2 = new Tax(65000, "TX", 4 );
```

The Keyword static

Java has a special keyword, `static`, that indicates that the class variable will be shared by all instances of the same class. If the class `Tax` had the declaration `static double grossIncome;` then this variable's value would be shared by all instances of the class `Tax`, which doesn't make sense. Besides, after the creation of two instances (`t1` and `t2`), as in the preceding code, the first value of the variable (`50000`) would be overwritten with the second one (`65000`).

But if you introduce in `Tax` a class variable to count the number of its instances (think the number of customers whose taxes have been calculated), such a variable has to be declared as `static`, so its only version can be incremented by each instance on creation, as in **Listing 4-7**.



Listing 4-7: The Tax class with the keyword static

```
class Tax {
    double grossIncome; // class member variables
    String state;
    int dependents;
    static int customerCounter;

    // Constructor
    Tax (double gi, String st, int depen){
       grossIncome = gi;  // member variable initialization
       state = st;
       dependents=depen;
       customerCounter++;   // increment the counter by one
    }
}
```

You can also declare methods with the `static` qualifier. Such methods can be called without the need to instantiate the class first. This is usually done for utility methods that don't use any instance variables, but rather get input via the argument and return the result.

The following function converts Fahrenheit to Celsius and returns the result:

```
class WeatherReport{
      static double convertToCelsius(double far){
           return ((far - 32) * 5 / 9);
    }
}
```

You can call this function from another class without the need to instantiate `WeatherReport` first:

```
double centigrees=WeatherReport.convertToCelsius(98.7);
```

Try It

In this section you create yet another version of the `Tax` class with a three-argument constructor and add a utility function to convert the tax value from dollars to euros, assuming the dollar-to-euro conversion rate is 1.25.

Lesson Requirements

For this lesson you should have Eclipse IDE installed. If you prefer to use a plain text editor to code your Java programs, that's fine too.



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 Lesson4 folder in the download.

Step-by-Step

1. In Eclipse IDE, create a new project named Lesson 4.

2. Create a new class called `Tax` (File ⇒New ⇒Class). Enter the code shown in **Listing 4-7**.

3. Add the following statement as the last line of the constructor of the class `Tax`:

```
System.out.println("Preparing the tax data for customer #" + customerCounter);
```

4. Add the method `calcTax()` to the class `Tax` and calculate tax by multiplying the gross income by 0.33 and deducting the number of dependents multiplied by one hundred:

```
return (grossIncome*0.33 – dependents*100);
```

5. Add the `static` function to `Tax` to convert the calculated tax to euros, applying the currency-conversion rate of 1.25. Print the calculated tax in euros using the function `System.out.println()`.

6. Create a `TestTax` class and input the code from **Listing 4-6**. Add to this class yet another instance of the class `Tax`:

```
Tax   t2 = new Tax(65000, "TX", 4 );
```

Calculate the tax using the second instance of the class `Tax`:

```
double hisTax = t2.calcTax();
```

7. Call the method twice to convert the currency, passing the calculated tax from `t` and `t2` as an argument.

8. Run the class `TestCalc` (right-click and select Run As⇒ Java Application). The Console view should display the two “Preparing the tax…” messages followed by the two messages with the calculated tax in euros.
To get the sample database files you can download Lesson 4 from the book’s website at **www.wrox.com**.



Please select Lesson 4 on the DVD with the print book, or watch online at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.

Chapter 5

Back to Java Basics

This tutorial didn't start with detailed coverage of basic constructs of the Java language such as the syntax of `if` statements, loops, and the like. You started learning Java programming with getting used to object-oriented terms and constructs of the language. This lesson is a grab bag of basic language elements, terms, and data structures.

Arrays

An **array** is a data storage that's used to store multiple values of the same type. Let's say your program has to store names of 20 different girls, such as "Masha", "Matilda", "Rosa" and so on. Instead of declaring 20 different `String` variables, you can declare one `String` array with the capacity to store 20 elements:

```
String []  friends = new String [20];  // Declare and instantiate array
friends[0] = "Masha";                  //Initialize the first element
friends[1] = "Matilda";                //Initialize the second element
friends[2] = "Rosa";

// Keep initializing the elements of the array here

friends[19] = "Natasha";                //Initialize the last element
```