Author Proof Instructions

Please return answers to the questions on this form and your completed proof to the attention of Rachel George by the indicated due date.

<table>
<thead>
<tr>
<th>Course rev:</th>
<th>516/CN/A.2/611/A.1</th>
<th>Coordinator:</th>
<th>Rachel George</th>
<th>Editor:</th>
<th>Beverly Voigt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proof sent:</td>
<td>10/31/06</td>
<td>Proof due back to Pubs:</td>
<td>11/1/06</td>
<td>TE copy sent to:</td>
<td>Stephen Neal</td>
</tr>
<tr>
<td>Author name:</td>
<td>Valliappa Lakshmanan</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Enclosed:

☑ Course Notes ☑ IG ☑ Exams ☑ MA

You have the following options for correcting and returning the proof:

1. If you have made minor corrections that do not require repagination, please return only those corrections by e-mail to rachel_george@learningtree.com, by overnight delivery, by fax at 310-414-9498, or by giving the coordinator the corrections over the phone. Please use red or green ink on hardcopy pages.

2. If you insert, delete, or move any pages, please return the entire proof by overnight delivery (e.g., Federal Express)—do not use regular mail.

Please return the proof by

11/1/06

8:00 a.m., Pacific Time.

(continued on next page)
Please review and respond to the following questions and issues:

Production:

Hi, Lak,

The exam files you submitted showed no changes from the A.1 revision. I just wanted to double check that you had no changes for any of the exams. Thanks!

Rachel George
Publications Production Coordinator
rachel_george@learningtree.com
310-342-2041

Editorial:

Hi, Lak.

I proofed your 516 minor revision. Everything looked very clean! I have a few questions for you (below).

Please contact me if you’d like to discuss these issues. Otherwise, please send your responses to the production coordinator. Thanks!

Beverly Voigt
Managing Editor
beverly_voigt@learningtree.com
310-342-2228

<table>
<thead>
<tr>
<th>Page number</th>
<th>Question/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-13</td>
<td>First bullet, last dash: OK to change the comma between “init” and “copyconfig” from Courier New to Arial? It’s not part of the code, correct? In fact, can this just be changed to “the init and copyconfig targets”?</td>
</tr>
<tr>
<td>2-69</td>
<td>Second bullet, third dash: This refers to levels that are better named, not named levels that are better, correct? If so, I’d like to hyphenate “better named” to make it clear.</td>
</tr>
<tr>
<td>3-10</td>
<td>First bullet: Are “Promotion” and “Discount” two separate things (i.e., not “Promotion,Discount”)? If so, I’ll add a space after the comma and change the comma to Arial. Also, should it read “... may need to interface” rather than “... interfaces”?</td>
</tr>
</tbody>
</table>

(continued on next page)
<table>
<thead>
<tr>
<th>Page Numbers</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-78</td>
<td>Second bullet, last dash: I’m wondering if we can lowercase “Proxy” in “… within factories using a Proxy.” Throughout the course, it seems to be lowercase when it’s referred to generically (see slides 3-19, 6-6, 6-9, etc.) and uppercase when referring to the “Proxy object” or the “Proxy pattern.” Is it OK to make this distinction? If so, should we lowercase “Proxy” here?</td>
</tr>
<tr>
<td>5-12 &amp; 5-24</td>
<td>We replaced the original bitmappy graphic with a cleaner one. Is this new one OK?</td>
</tr>
</tbody>
</table>

That’s it! Thank you.

Please verify the following items before returning your proof:

- Entry of my original author input is correct and complete.
- All Pubs’ questions have been answered.
- Conferred with TE and included his/her comments.
- Exercises are complete and solutions are correct.
- Cross-references in course materials correspond accurately.
- IG comments correspond to correct page in Course Notes.
- All acronyms/abbreviations have been spelled out on first reference.
- Background Questionnaire is relevant and appropriate.
- Exams reflect current course revision and there is only one correct answer for each exam question.

Comments?

____________________________________

____________________________________

____________________________________

____________________________________

Due back to Pubs by 8:00 a.m., Pacific Time:
11/1/06

- [x] Course Notes  - [x] IG  - [x] Exams  - [x] MA
Attendee Background Questionnaire

Course location: ____________________________ Date: ____________________________

Welcome to Course 516. Please take a few moments to complete this short questionnaire/self-assessment and hand it to your instructor before the start of class or at the first break. The information you provide will assist the instructor in meeting your needs during the course.

Name: __________________________________ Company/org.: ____________________________

Job title: ____________________________ Job function: ____________________________

1. What is your primary job function? (Please check one.)
   - Application programmer
   - Database programmer
   - Web developer
   - Business analyst
   - Project leader or manager
   - Other: ____________________

2. What is your level of knowledge/experience and interest in the following areas?

   Knowledge/experience | Interest
   ----------------------|-------
   Low                  | Medium | High |
   Low                  | Medium | High |
   Low                  | Medium | High |
   Low                  | Medium | High |
   Low                  | Medium | High |
   Low                  | Medium | High |

   Multithreaded programming
   Performance tuning
   Java security APIs
   Java 5 new language features
   Object-oriented principles
   GOF patterns
   Servlet/JSP programming
   Swing programming
   Ant
   Unit testing

3. What Java projects are you currently working on? __________________________________________

4. What is your main objective in attending this course? ______________________________________

5. Which Learning Tree courses have you taken? ____________________________________________

Please hand the completed form to your instructor, either before the start of class or during the first break.
Instructor Guide for Course 516

Best Practices in Java Programming: Hands-On

by Valliappa Lakshmanan

Technical Editor:
Stephen Neal

This document corresponds to the A.2 revision of the course notes
Course History

Your feedback on this course is important!

Please forward your comments and suggestions to the Product Development team via the Product Development Web site at http://pdev.learningtree.com.

516/CN/A.2/611/A.1  November 2006
Made part of the JUnit exercise a bonus to make it easier to finish Chapter 2 on the first day even for slow classes. Improved wording on a few slides, and an explicit example of enum. Otherwise no major changes.

516/CN/A.1/608/A.1b  August 2006
This is the first public revision of the course.

Slides that covered more than about 2/3 were separated into two slides. Added intro sections on Eclipse and Tomcat. Completely rewrote the second day’s material to essentially remove XDoclet, JavaBeans, etc. XDoclet will be removed with the next brochure change. JavaBeans may or may not go. The intent of these sections was changed to reflect a “framework” focus from that of “reducing code duplication.”

516/CN/A.1b/607/—  July 2006
This is the beta revision of the course.
Notes to Instructor

The course is divided into four approximately equal parts:
1. Ant, unit testing, and logging
2. Design patterns and other ways of improving code quality
3. Performance tuning and threading
4. Miscellaneous topics: proxies, scripting, JMX, security, secure coding

There is a lot of material and you will almost definitely not have to chew up time to keep the class from finishing early. However, do not be misled by the slide count. Most of the slides have only the top of the slide with material, to help in those cases when you just can not get the screen high enough. Because of this, a good rule of thumb in this course is two minutes per slide (not three).

On the first day, the fun thing is unit testing. Blaze through the Ant section—they will get how useful Ant is by the end of the course because everything will be launched from Ant. Don’t oversell Log4J—the party line is that they can use any logging API that will let them turn on/off logging on a per-class basis. The second day is the best practices core of the course. Get them excited about good design and proper encapsulation. The key point of the third day is that performance tuning is dangerous, and that the stuff we’re talking about is only for 2 percent of their code—not for everything. There is no thread weaving the final day of topics together, but they are all quite interesting, so few students will complain.

Most importantly, give students time to finish the exercises. For slow classes, run through the exercise manual and answer all the questions so that they only need to do the code. Use the “hints” page for the debrief of all the exercises. It shows what the code used to be and what they changed it to.

A few pointers to have a successful teach with this course:

(1) Be a bit observant with the JUnit exercise. It involves them using BigDecimal. The experienced developers fly through the exercise; the beginners may need help. I went around saying loud enough for every one to hear: “Create a BigDecimal object and call its substract method.”

(2) Next, do debrief all exercises. I answered all the questions in the ex manual and then showed them the webpage that contains the “Exercise hints.” This has the old code in red and the new code in blue, along with context. So, it saves you from hunting around in Eclipse for where the code is. At that time, make sure to mention (I did this four or five times during the course) that the TODOs are the “key steps to implement the Factory pattern” (or whatever it is they are doing).

(3) Don't get bogged down. The performance chapter should not take a whole day :)

516-IG-3
Course 516

Best Practices in Java Programming: Hands-On
Acknowledgments

The author would like to acknowledge the following for their contributions to this course:

- Ian Darwin
- Donna Lahey
- Steve Lockwood
- Megan Marler
- Chris Mawata
- Karen Najjoum
- Steve Neal
- David Vickars
- A., S.M, and S.S.
Introduction and Overview

Notes:

<ipf>B,1: Set the audience’s expectation for a great course!</ipf>

• The introduction provided in Chapter 0 sets the stage for the entire course!
  - The audience forms its opinion of you and the course in the first few minutes of the presentation
  - Be positive, concise, and enthusiastic—set the audience’s expectation for a great course!
  - We spend a lot of time preparing the other chapters of the course
    -- Because of its importance, Chapter 0 is worth some rehearsal to make it smooth
Course Objectives

In this course, you will learn how to

- Apply Java best practices to increase productivity and build fast, secure, and reliable applications
- Optimize the compilation, deployment, and testing of software applications
- Solve architectural problems with proven design patterns and advanced language features
- Maximize software performance
- Improve the reliability of threaded applications
- Customize application behavior with scripting
- Secure sensitive data and authenticate users with JAAS

Notes:

• Briefly explain the main objectives of the course, focusing on the benefits to the students (approximately 5 minutes)
  - Look at it from the student’s point of view
    -- “Radio WIIFM”: What’s In It For Me?
    -- Provides motivation for students to be interested in course topics
  - Set realistic expectations

Expectations are VERY important. I cannot stress enough that on this slide you must do at least one thing: Tell everyone you just gave them a brief overview of what we will cover in the course and ask them to indicate what their expectations are (are they in line with what we said we will cover?). Much better to find out today, not on Friday when the course is over

Expound the benefits of the topic’s we’ll be covering: E.g. Threading allows applications to support many more users…
Course Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction and Overview</td>
<td></td>
</tr>
<tr>
<td>Chapter 1</td>
<td>Effective Programming in Java</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>Optimizing Software Development</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>Improving Code Quality</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>Tuning for Performance</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>Taking Full Advantage of Threads</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>Extending Application Functionality</td>
</tr>
<tr>
<td>Chapter 7</td>
<td>Implementing Security Best Practices</td>
</tr>
<tr>
<td>Chapter 8</td>
<td>Course Summary</td>
</tr>
<tr>
<td></td>
<td>Course Evaluation</td>
</tr>
<tr>
<td>Appendix A</td>
<td>Self-Study: Servlets</td>
</tr>
<tr>
<td>Appendix B</td>
<td>Self-Study: Leveraging the Operating System With NIO and JNI</td>
</tr>
<tr>
<td>Appendix C</td>
<td>Self-Study: Java Cryptography APIs and Tools</td>
</tr>
</tbody>
</table>

Notes:

<ipf>R,3: Provide rough time schedule (chapters per day)</ipf>

- **Provide rough time schedule (chapters per day) (1 minute)**
  - Don’t re-cover course objectives here, just give students some idea of time allocation
  - Still allows you some flexibility to adjust timeline
Course Materials

- **Course Notes**
  - Copies of all slides and supplemental presentation material
- **Exercise Manual**
- **Supplementary Course Materials at**
  [www.learningtree.com/bonus](http://www.learningtree.com/bonus)

Notes:

<ipf>L.4: Customize notes by underlining, highlighting, adding addtl. notes</ipf>

- **Tell students they can maximize the effectiveness of the Course Notes by underlining, highlighting, and adding their own notes (1 minute)**
  - Reinforces the learning experience, lets them focus on what’s important to them
  - Adding their own notes to course notebook makes it a more valuable reference when they return to work
Expected Background

- **Real-world experience in Java is expected**
  - Familiarity with the core Java API
  - Understanding of object-oriented principles
  - Experience designing, defining, and extending Java classes
  - Experience implementing Java interfaces
  - To the level of Learning Tree International Course 471, Hands-On Java Programming

- **Experience with XML is helpful**
  - Several tools use XML for their configuration

- **Experience with Java Web applications is helpful**
  - Web applications are mentioned in several examples
  - Appendix A contains some introductory material

Notes:

<i>Expected Background</i>
Our Classroom Environment

• In our public courses, we use the following setup:
  — Windows XP Professional, per team of students
  — Java 5
  — Apache Tomcat
  — Apache Ant
  — Apache log4j
  — Jakarta JMeter
  — Java Interactive Profiler (JIP)
  — BeanShell
  — Eclipse
  — MySQL
  — TextPad
  — Cygwin (C compiler, UNIX shell)
Hands-On Exercises

- **Exercises appear throughout the course**
  - Involve the same case study (and code base) throughout
  - Familiarize yourself with the objectives of the case study in the first exercise

- **Designed to reinforce your learning**
  - In each exercise, you will write code relevant to the learning objective only
  - Questions in Exercise Manual meant to help you understand “why”
  - Work on exercises in teams
  - Discuss solutions with your partner, then try them

- **Many exercises have “if you have more time” sections**

- **Solutions are provided for all exercises**

- **All exercises, solutions, and course examples are available to take home**
  - You can also take home any work you perform during the course

---

Notes:

<ipf>R,7: Hands-On Exercises</ipf>

THIS IS A DIFFERENT SLIDE from other Learning Tree courses.

Tell them that exercises are very important. Also point out that they will not write complete code, Only the part of the code relevant to the learning objective – set the expectation right here

Why are exercises important?

Approximate level of retention for an average adult for different learning methods:

- Presentation only = 30%
- Presentation and taking notes = 50%
- Applying techniques (exercises) = 70%
- Discussion (exercises with a partner) = 70% +
Java Best Practices

- Proven best practices and industry-standard techniques are mentioned throughout the course
  - Special icons will be used to easily identify these items
    - Indicates a recommended practice
    - Indicates something to avoid
- Not all best practices will be applicable to all situations
  - A catalog of ideas to draw from
  - Some best practices may need to be balanced against real-world concerns

Notes:

Make it clear that not every topic we cover will be applicable to their situation at present. Maybe in a couple of years … maybe never!
Asking Questions Is a Best Practice

Feel free to ask questions at any time
— Questions often indicate interest in the topic
— Questions may indicate material is unclear
— Instructor can elaborate, or suggest discussion offline

Do not feel free to debug exercises during lecture time
— Can be extremely distracting to others in the classroom

Notes:
<ipf>R,9: Asking Questions Is a Best Practice</ipf>
Chapter 1

Effective Programming in Java

Notes:

<ipf>B,1: Effective Programming in Java</ipf>
Chapter Objectives

In this chapter, we will

• Learn key characteristics of high-quality software
• Discuss code-level best practices
• Examine the class project

Notes:

<ipf>L,2: Chapter Objectives</ipf>

Both high-level and low-level. Tell them that we’ll concentrate on high-level best practices in the course.
Effective Programming in Java

Key Characteristics of High-Quality Software

Eclipse and Tomcat

Hands-On Exercise 1.1

Notes:

<ipf>R,3: Effective Programming in Java</ipf>
High-Quality Software

- High-quality software is software that works
  - Satisfies user requirements
  - Has needed functionality
  - Is stable
  - Performant

Notes:

<ipf>L.4: High-Quality Software</ipf>

I like to tell the story of Friendster and MySpace here. Friendster was the first to do social networking, but their website was complex (would allow only friends to view each others’ profiles) and took over 40s to load. MySpace was a copycat, with a faster website (any one can look at anything) that hit upon the idea of organizing around music bands. Unfortunately, because Friendster was already so slow, their architecture did not permit them to quickly add band-functionality to their website. Yahoo wants to buy MySpace for $800m while Friendster can not find a buyer at $12m.
Key Characteristics of High-Quality Software

- High-quality software meets user requirements in several key respects
- **Maintainability and testability (Chapter 2)**
  - Speed of development must keep up with business requirements
  - The addition of new features should not lead to earlier features breaking
- **Extensibility (chapters 3, 6)**
  - Software should have the features that users want
  - Ability for users and programmers to add behavior easily
- **Performance (Chapter 4)**
  - Be responsive to user interaction
  - Work within hardware constraints
- **Scalability (Chapter 5)**
  - Support a greater number of concurrent users
- **Security (Chapter 7)**
  - Safeguard data and resource access

**Notes:**

<ipf>L,5: Key Characteristics of High-quality Software</ipf>

Works with previous slide

This is the story-line of the course, and explains the choice of topics in this course.

Draw on the linkage between software that works and maintainability, testability and extensibility.

Spend time here. I tell the story (previous slide) to emphasize extensibility, performance, maintainability, scalability, simplicity, etc. Tie it to the course. In the beta, this slide was a key takeaway at the end of Day 1.
Coding Conventions

- Following a good coding convention improves maintainability

```java
/**
 * Analyze and forecast the quality of a promotion if run at a retailer.
 * @param retailerId unique id of retailer
 * @param promotion promotion to analyze
 * @return quality of the given promotion at the given retailer
 */
public AnalysisResult analyze(int retailerId, Promotion promotion) {
    // initialize analyzer for promotion
    PromotionAnalyzer analyzer = new PromotionAnalyzer(promotion);
    // load all retailer's purchases
    PurchaseDAO dao = new PurchaseDAO();
    List<Purchase> purchases = dao.getPurchasesForRetailer(retailerId);
    // perform the analysis
    for (Purchase purchase : purchases) {
        analyzer.updateAnalysis(purchase);
    }
    return analyzer.getQuality();
}
```

Notes:

<ipf>L,6: Coding Conventions</ipf>
Coding Best Practices

Code should be readable and easy to follow
— Indent code properly and consistently
— Use descriptive names
— Keep methods to no longer than a page
— Wrap lines before 80 characters
— Comment major steps
— Comment non-trivial methods
— Declare local variables close to their first use, not at start of method

Head off potential maintenance problems:
— Use compound statements for if, else, for, while, etc.
— Explicitly mark empty blocks

• Sun’s coding conventions generally accepted industry-wide
  — Integrated development environments have these built in
  — Be consistent within development team

Notes:

R,7: Coding Best Practices

We’ll talk about performance tuning, but tuning is only for very targeted situations. Most code should be simple even if the simplicity will make it less efficient.

Declaring local variables close to first use is also more efficient.

Compound statement: Otherwise, maintainer may mistakenly add new line and think it’s part of if block.
Javadoc

- Document all classes
- Document all public methods
  - Except possibly trivial ones, like getters and setters
  - Can inherit documentation from superclass or interface
    - /** {@inheritDoc} */
- Describe what the method does
  - Not how it does it
- Choose first sentence fragment carefully
  - Shows up in the summary

Notes:

<ipf>L,8: Javadoc</ipf>

The case study does the inheritDoc. See, for example, FixedDiscountPromotion in the “complete”. Can also leave it out, in which javadoc will do it but maintainers wont know if you meant it that way or you just forgot …
Poor Coding Style

Avoid writing cryptic code

- Simple statements in loop blocks, with empty statements unmarked
- Confusing indentation
- Unclear what code is trying to do

```java
int i;
for (i=0; i < promotions.length && !promotions[i].isValid(); ++i);
    i = i-1;
```

WARNING! The i=i-1 is poorly indented. Note the semicolon at the end of the second line.
The next slide has this bad code written well.

This is the only example of good/bad code they’ll see. Tell them that we will look at higher-level best practices and are not out to be code convention nazis.
**Good Code Is Self-Documenting**

**Write simple, readable code**
- Compound statements in loop blocks
- Clear indentation
- Logic is easier to follow

```java
int firstValid = -1;
for (int i=0; i < promotions.length; ++i){
    if (promotions[i].isValid()){
        firstValid = i;
        break;
    }
}
```

Write simple, easy-to-follow code, even if it is less brief or less efficient.

---

**Notes:**

<i>R,10: Good Code Is Self-Documenting</i>

Same code as previous slide. Tries to find the first valid promotion in the list
Effective Programming in Java

Key Characteristics of High-Quality Software

- Eclipse and Tomcat

Hands-On Exercise 1.1

Notes:

<ipf>L,11: Effective Programming in Java</ipf>
Eclipse

- Eclipse provides an open-source platform for building software

Notes:

<ipf>L,12: Eclipse</ipf>

Supports C++, Ruby besides Java

Feel free to make this into a demo where you run through the points on the next two slides
Eclipse and Java

- The Java perspective in Eclipse has several useful panes
  - Editor
    - To view and modify Java files
  - Package Explorer
    - Package-by-package view of the source files in application
  - Navigator
    - File-by-file view of the project folder
  - Problems
    - Shows compiler errors and warnings
  - Tasks
    - Shows lines marked TODO in the Java code
  - Console
    - Shows output of applications run from within Eclipse
  - Ant
    - Integrates with Java build tool

Notes:

<R,13: Eclipse and Java</R>
Tasks view does not handle TODOs in non-Java files such as in the build.xml
The first three are on the left
Next three are on the bottom
Last one is on the right
Eclipse Tips and Techniques

- **A few productivity tips for Eclipse:**
  - “Source | Format” will apply consistent format to Java code
  - Ctrl + Space will bring up context-specific completion
    - Typing a period in Java code brings method completions
  - Eclipse finds syntax errors as you type
    - Click red “x” or yellow “bulb” to get suggestions
  - Compare two sets of files side by side
    - Select any two items (files/packages/directories) in Package Explorer
    - Right-click and select Compare | With Each Other
  - To run an application:
    - Right-click a Java file with a main method
    - Select Run As | Java Application
  - Select variable or class name, right-click, and select References | Project
  - Jump to compiler error or TODO by clicking it in the bottom pane
  - Save Java file to compile the code and create a .class file

**Notes:**

<ipf>L,14: Eclipse Tips and Techniques</ipf>

Context-sensitive completion: If you do it inside a starting tag of an Ant XML file, will get list of allowed attributes
Tomcat

- Tomcat is an open-source servlet container from Apache
  - Reference implementation of Java Server Pages (JSPs) and servlets
  - Widely used as a production environment
- In exercises, need to deploy servlets into Tomcat
  - As a Web Application archive (.war) file
  - Tomcat needs to be running in order to deploy .war files and test them
  - Leave it running throughout this week

Notes:

<ipf>R.15: Tomcat</ipf>
Effective Programming in Java

Key Characteristics of High-Quality Software

Eclipse and Tomcat

Hands-On Exercise 1.1

Notes:

<i>IPF>L,16: Effective Programming in Java</i>
The Class Project

- **Build data mining software for a wholesaler**
  - Examines past purchases and sends out coupons to retail outlets
  - The coupons are tailored to the type of customers a retailer attracts

- **Two applications**
  - Internal application for analyzing current and potential promotions
  - Web application used by retailers to obtain current promotions

---

**Notes:**

<ipf>R,17: The Class Project</ipf>

Existing promotion is first tab of Swing GUI; scripting is the means to analyze potential promotions and is the second tab.

The servlet is the external application. All are built on the same backend.

They will design and build all of the backend (except the datamining computational engine, but no need to tell them that now).
Hands-On Exercise 1.1: Data Mining for a Wholesaler

- In this exercise, you will explore the completed course project in order to
  - Understand the case study

Please refer to the Exercise Manual

Notes:

<ipf>L,18: Hands-On Exercise 1.1: Data Mining for a Wholesaler</ipf>

\There is no database of promotions by retailer_id.

This information is generated dynamically by the application by looking at past purchases at the retailer.
Review

1. Name some key characteristics of good software.

2. Why is it important to follow coding conventions?

Notes:

<ipf>R,19: Review</ipf>

(1) Usability, reliability, robustness – accept things they suggest and relate it to course chapters.
(2) Readability
Summary: Effective Programming in Java

- High-quality software meets user requirements
  — Extensible, maintainable, testable, scalable, performant
- Following coding conventions improves maintainability
- Write simple, easy-to-read code

Notes:

L,20: Summary: Effective Programming in Java

The best practices slide at the end of each chapter is the true summary slide. Spend time here. Breeze through the standard one.
Chapter Summary

In this chapter, we have

- Learned key characteristics of high-quality software
- Discussed code-level best practices
- Examined the class project

Notes:

L21: Chapter Summary

A quick chapter presenting concepts that we will build on throughout the course.
Notes:

<ipf>B.1: Optimizing software development with proven techniques</ipf>
Chapter Objectives

In this chapter, you will learn how to

- Simplify project build, test, and deploy environment using Apache Ant
- Apply test-driven development
  — Create unit tests to ensure that classes were coded correctly
  — Create functionality tests to ensure application behavior
- Configure runtime logging settings

Notes:

Motivate them by telling them that if they get started wrong, they may never recover.
Good practices can be self-reinforcing.
This chapter will show them how to get started “right”.

Optimizing Software Development

Automating Build Management With Ant
Leveraging Advanced Features of Ant
Hands-On Exercise 2.1
Unit Testing
Hands-On Exercise 2.2
Functionality Testing
Logging
Hands-On Exercise 2.3

Notes:

<ipf>R,3: Optimizing Software Development</ipf>
Steps of a Typical Build

- Compiling and deploying Java applications and J2EE components is quite involved
  - Typically involves writing configuration files
  - Specifying names and mapping
  - Placing classes and configuration files in a specified hierarchy
  - Creating a .jar, .war, or .ear file
  - Telling the container about it

Notes:
L.4: Steps of a Typical Build
Java Build Process

- For example, the Java build process for RainForest's data mining servlet:

Notes:

<i>R,5: Java Build Process</i>
Need for Automation

- Just deploying the application to test it can become quite a chore
  - Several Integrated Development Environments (IDEs) help
  - But you don’t want your deployment scripts tied up in a developer’s IDE settings!
- Need to automate your build, test, and deploy environment

Notes:

<i>IPF>L,6: Need for Automation</i>

Note why using an IDE’s deployment tool is a bad idea for a shop with more than one developer with more than one IDE
Also, even with a standard IDE being used, the IDE settings are not part of the project, they are part of an individual developers IDE setup.
Relate to previous exercise difficulties.
Ant for Automation

- Apache’s Ant is a Java solution to build automation
  - A powerful build tool
  - Written in Java, so cross-platform
  - Extensible: can adapt it to your needs
  - Uses XML syntax

Don’t rely on point-and-click for builds. Automate them using Ant, so that every installation follows the same process.

Notes:

<ipf>R,7: Ant for Automation</ipf>

Make clear that the best practice is “Use Ant”. The rest of this section is just a tutorial On using Ant.
XML Syntax

- **Highlights of XML syntax**
  - Self-describing
  - Hierarchical
    - One root element
  - Attributes in opening tag
  - Special syntax for elements that are empty

```xml
<employee id="35687">
  <name first="Jane" last="Doe" />
  <contact>
    <office building="MichaelFaradayBuilding" room="A2034" />
    <phone>
      <mobile>15553422431</mobile>
      <work>15553422431</work>
    </phone>
  </contact>
</employee>
```

*Notes:*  
<ipf>L.8: XML Syntax</ipf>
What Can Ant Be Used For?

- **Ant can be used to**
  - Compile Java code
  - Execute Java applications
  - Create, copy, and delete files
  - Unit-test components
  - Deploy to server environments

- **Ant integrates with**
  - Source-code management systems
  - IDEs
  - Application servers

**Notes:**

<ipf>R,9: What Can Ant Be Used For?</ipf>

Make clear that the best practice is “Use Ant.” The rest of this section is just a tutorial on using Ant.
Example Ant Build File

```xml
<project basedir="." default="build" name="so21">
  <property file="../build.properties" />  
  <path id="project.classpath">
    <pathelement location="${build.dir}" />  
    <fileset dir="${common.lib.dir}"/>\include name="*.jar" /></fileset>
  </path>
  <target name="init">
    <tstamp />
    <mkdir dir="${build.dir}" />
  </target>
  <target name="clean">
    <delete dir="${build.dir}" />
  </target>
  <target name="build" depends="init,copyconfig">
    <javac destdir="${build.dir}" srcpath="src" classpathref="project.classpath" />
  </target>
  <target name="copyconfig" depends="init">
    <copy todir="${build.dir}" fileset="config">\include name="*" /></fileset>
  </target>
</project>
```

Notes:

<ipf>L,10: Example Ant Build File</ipf>

Keep this up as you go through the following slides. The stuff in bold is referenced in the slides.
The Build File

- **Ant’s build files are written in XML**
  - The default name is **build.xml**
  - Launch by typing **ant** in directory that contains **build.xml**
  - Double-click target in Eclipse

- **Ant has a number of built-in tasks for common Java build operations**
  - **javac**
  - **copy**
  - **war**

- **Full list of core Ant tasks can be found in Ant manual**

Do Now

1. Navigate to course home page
2. Click link for Ant manual
3. Navigate to core Ant tasks

Notes:

<ipf>R,11: The Build File</ipf>
An Ant Project

- The root element is called `project`

- An Ant project consists of a number of targets
  - These specify a set of tasks to execute
  - Specify one of them as the default target for the project
    - Launch other targets as `ant clean`, for example

- Ant tasks are placed inside a target
  - Example: the `delete` task in the `clean` target

Notes:

<ipf>R,12: An Ant Project</ipf>
Dependencies

- **A target can depend on other targets**
  - A target isn’t executed until all of its dependencies have been completed
  - Regardless of the order in which the targets are specified in the build file
  - In the order that the dependencies are specified
  - Example: the build target depends on the init, copyconfig targets

- **Ant does dependency checking on files**
  - On compile, jar, copy, etc.
  - Speeds up tasks
  - tstamp task must be part of target sequence
    - The example has placed this in the init target

**Notes:**

<ipf>L,13: Dependencies</ipf>
fileset

- fileset is used to process sets of files
  - Can be built using patterns

```xml
<target name="copyimages">
  <copy todir="${web.dir}/images">
    <fileset dir="${image.dir}">
      <include name="**/*.png"/>
      <exclude name="*internal/**.png"/>
    </fileset>
  </copy>
</target>
```

- Patterns include:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Wildcard</td>
</tr>
<tr>
<td>**</td>
<td>Subdirectories wildcard</td>
</tr>
<tr>
<td>?</td>
<td>Optional</td>
</tr>
</tbody>
</table>

Notes:

<ipf>L,14: fileset</ipf>
Reusing fileset

Reuse a fileset by incorporating it into a path
— For example, the project.classpath

<path id="project.classpath">
  <pathelement location="${build.dir}" />
  <fileset dir="${common.lib.dir}" include="*.jar" /></fileset>
</path>

<target name="build" depends="init.copyconfig">
  <javac destdir="${build.dir}">
    <src path="src" />
    <classpath refid="project.classpath" />
  </javac>
</target>

• A pathelement is a single location
— Unlike a fileset, which may consist of multiple files matched by a pattern

Notes:

<ipf>R.15: Reusing fileset</ipf>
XXX: break into two slides
Properties

- **Properties are like variables:** They can be set and used later
  - **Avoids repetition**
    - Allows easy changing
      - <property name="deploy.dir" location="/projects"/>
    - Use as
      - <delete dir="${deploy.dir}"/>
  - **Properties can be loaded from a file**
    - <property file="../build.properties"/>

*Notes:*

<ipf>L,16: Properties</ipf>
Built-in Properties

- Some properties are built in
  - `${ant.project.name}` is provided by Ant
  - The Java `System.getProperty()` is also available
    - `${user.home}` and `${user.name}`, for example

Notes:

<R,17: Built-in Properties</R>
Common use: `<property file="build_${os.name}.properties"/>` to have OS-specific property files
Properties Are Immutable

- Ant tasks will use the value the first time a property is set
  - The second and subsequent attempts get ignored

A common idiom in build files is:

```xml
<property file="${user.home}/build.properties"/>
<property file="./build.properties"/>
<property name="deploy.dir" location="../deploy"/>
```

- Look to see if user has a preferences file
- Then load and use project settings
- If project settings don’t have that property set, use some hardcoded default
  - Make sure the hardcoded default is a relative path

Notes:

R,18: Properties Are Immutable

If someone writes a custom Task and calls setProperty(), they can change it – so the immutability is more of a convention than something inherent to Ant.

Custom tasks should call setNewProperty() to honor the “first-set” rule.
Optimizing Software Development

Automating Build Management With Ant

Leveraging Advanced Features of Ant

Hands-On Exercise 2.1

Unit Testing

Hands-On Exercise 2.2

Functionality Testing

Logging

Hands-On Exercise 2.3

Notes:

<ipf>R,19: Optimizing Software Development</ipf>
filterchain

• Filter chains allow you to apply a sequence of tasks on files
  — As part of loading, copying, or moving a file

```xml
<target name="distributesource">
  <tstamp/>
  <copy todir="${dist}/src">
    <fileset dir="src">
      <include name="**/*.java"/>
    </fileset>
    <filterchain>
      <concatfilter prepend="copyright.txt"/>
      <tokenfilter>
        <replaceregex pattern="@@DATE@@" replace="${DSTAMP}:${TSTAMP}"/>
      </tokenfilter>
    </filterchain>
  </copy>
</target>
```

• Like UNIX pipes

Notes:

<ipf>L,20: filterchain</ipf>

@@DATE@@ will get replaced when part of a source file – use this for
Help: About functionality

May want to mention that they don’t want to overdo this. There is no preprocessor in Java, for a reason.
Extending Ant With `exec`

- May need functionality that is not part of Ant’s core tasks
- Ant can call external scripts or executables
  - `exec`: run OS script or executable

```xml
<exec executable="ipconfig">
  <arg value="-all"/>
</exec>
```

Is there a disadvantage to doing this?

---

**Notes:**

<ipf>R,21: Extending Ant With exec</ipf>
Calling Java Programs Instead of \texttt{exec}

- For portability, you could launch Java programs from Ant

\begin{verbatim}
<java classname="com.ltree.crs516.PruneDatabase">
  <classpath refid="project.classpath"/>
</java>
\end{verbatim}

- Java application needs to be available on your machine
- Cannot easily make use of Ant properties or \texttt{fileset}

\textbf{Notes:}

\begin{verbatim}
<L,22: Calling Java Programs Instead of exec</verbatim>
Ant Is Extensible

- **Possible to write custom Ant tasks**
  - Custom Ant tasks can use Ant properties and filesets
  - For example, create custom task to update database whenever application is deployed to record
    - The name of the user
    - The name of the war file
    - How many Java files it was built from
    - The time of deployment

- **Many custom tasks have been contributed to Ant**

http://ant.apache.org/external.html

**Notes:**

<ipf>R,23: Ant Is Extensible</ipf>
Implementing an Ant Custom Task

- Steps to implement an Ant custom task:
  1. Implement the new Ant task in Java
     a. Write a Java class that extends Task
     b. Override the `execute()` method
  2. Use custom task in `build.xml`
     a. Define an XML element name for the task
     b. Use new task in targets

Notes:

<ipf>R,24: Implementing an Ant Custom Task</ipf>
Step 1: Implement the New Ant Task in Java

- **Extend** [org.apache.tools.ant.Task](http://example.com) — Override the `execute()` method
  - Should throw a `BuildException` on error

- **Consider which fields the task will require**
  - Provide setter methods for them
  - Ant will call these setter methods based on attributes in the `build.xml`

```java
public class DeployDBTask extends Task {
    private String username = null;
    private String war = null;
    public void setUsername(String username) {
        this.username = username;
    }
    public void setWar(String war) {
        this.war = war;
    }
    public void execute() throws BuildException {
        // do actual work here
    }
}
```

**Notes:**

> <ipf>L.25: Step 1: Implement the New Ant Task in Java</ipf>
Can do it without inheriting, but you lose a lot of the functionality

Put the next slide up so that the properties make sense
Step 2a: Define Task

- Define an XML element name for the task
  - From within the build file, using `taskdef`:
    ```xml
    <taskdef name="deploydb"
      classname="com.ltree.crs516.anttasks.DeployDBTask">
      <classpath refid="project.classpath"/>
    </taskdef>
    ```
  - Alternatively, load `taskdef`s from a properties file:
    ```xml
    deploydb=com.ltree.crs516.anttasks.DeployDBTask
    
    <taskdef file="dbtasks.properties">
      <classpath refid="project.classpath"/>
    </taskdef>
    ```

Notes:

R,26: Step 2a: Define Task

Alternatively, put class in a .jar file and add it to Ant’s classpath

The jar file must have a `defaults.properties` file containing the taskdefs
Step 2b: Using Custom Task

- Once the task has been defined, just use it

```xml
<deploydb username="${user.name}" war="${ant.project.name}.war">
  <fileset dir="src"> <include name="**/*.java"/> </fileset>
</deploydb>
```

**Notes:**

<ipf>R,27: Step 2b: Using Custom Task</ipf>
Ant Best Practices

☀️ Use common names for key targets
   — clean, init, prepare, compile, build, deploy, run, test, all

☀️ Use standard names for directories
   — src, build, lib, classes/bin, doc, dist

☀️ Use vendor-provided Ant tasks
   — For example, don’t copy files into Tomcat’s auto-deploy directory
     – Instead, use the Tomcat deploy, undeploy tasks

• Ant allows you to call external executables or scripts
  ❌ Avoid using exec to call platform-specific scripts
  ✔️ Write Ant tasks or launch Java programs

☀️ Create a classpath once, and use it by reference in the rest of the file

Notes:

<ipf>L17: Ant Best Practices</ipf>
Last point for maintainability, segue to next slide
Create file with common targets, taskdefs, etc., and import them (Ant 1.6+)

```xml
<import file="../common_targets.xml" />
```

— In earlier versions of Ant, use XML entity syntax

```xml
<!ENTITY commontargets SYSTEM "file:../common_targets.xml">
<project>
  &commontargets;
</project>
```

**Notes:**

<ipf>R,29: import</ipf>
antcall

Call other targets within the same build file using antcall

```
<target name="deploy" depends="war">
  <antcall target="undeploy" />
  <deployTomcat url="${tomcat.manager.url}"
  username="${tomcat.manager.username}"
  password="${tomcat.manager.password}"
  war="file:${user.dir}/${warfile}" path="/${ant.project.name}" />
  <antcall target="backup" />
</target>
```

- Unlike depends, antcall can invoke targets after main work
  - Such as the backup target

Notes:

<br/>
<ipf>R,30: antcall</ipf>

Also, parameters can be passed from the caller which makes it great for configuring targets.

```
<target name="build">
  <antcall target="show-message">
    <param name="param1" value="hello world…"/>
  </antcall>
</target>
```

```
<target name="show-message">
  <echo message="showing: ${param1}"/>
</target>
```

Path references can also be passed in a similar fashion – see the ant documentation for details.
Optimizing Software Development

Automating Build Management With Ant
Leveraging Advanced Features of Ant
▶ Hands-On Exercise 2.1
   Unit Testing
   Hands-On Exercise 2.2
   Functionality Testing
   Logging
   Hands-On Exercise 2.3

Notes:
<ipf>R,31: Optimizing Software Development</ipf>
Hands-On Exercise 2.1: Simplifying Compilation and Deployment With Ant

- In this exercise, you will learn how to
  - Automate your compilation and deployment environment using Ant
  - Extend the functionality of Ant by incorporating a custom task

Please refer to the Exercise Manual

Notes:

<ipf>R,32: Hands-On Exercise 2.1: Simplifying Compilation and Deployment With Ant</ipf>
Optimizing Software Development

Automating Build Management With Ant
Leveraging Advanced Features of Ant
Hands-On Exercise 2.1

Unit Testing
Hands-On Exercise 2.2
Functionality Testing
Logging
Hands-On Exercise 2.3

Notes:
R,33: Optimizing Software Development
How Can Software Be Tested?

- **Systematic testing helps make projects more manageable**
  - Integral part of all engineering projects

- **How are systems tested?**
  - The traditional engineering process (such as building a bridge) involves
  - A design stage where the entire system is designed
  - To meet user requirements and performance goals
  - A development stage where the system is built
  - Based on well-understood designs and models
  - A testing/evaluation stage where the built system is tested
  - Using rigorous, industry-standard methods

Can software be tested using the same approach?

**Notes:**

<ipf>R,34: How Can Software Be Tested?</ipf>
Testing as Part of the Development Process

Test at each stage of development

- Example: in a spiral development process
  1. Analyze feedback
  2. Design
  3. Change existing code
  4. Test new design
  5. Build new feature
  6. Test that it works
  7. Demonstrate to users
     - Gather feedback
     - Repeat

Notes:

<ipf>L.35: Testing as Part of the Development Process</ipf>

One loop (design to acceptance) of the spiral is for one vertical slice. Works with the next slide.

From previous acceptance test, you know the problems. Design so that you can fix those problems and implement

Next feature. Refactor the design to enable this. Unit-test the changes. Make sure the system behavior hasn’t changed.

Now implement the changes, unit-test and functionality-test. Then show to users (acceptance test). Repeat.

Build and unit test at the same time.
Functionality and integration test often at the same time.
Forms of Testing

- **Unit test**
  - Tests individual classes
  - Ensures classes have been coded correctly

- **Functionality test**
  - Tests vertical slices of the application
  - Ensures high-level behavior is as expected

- **Integration test**
  - Ensures the application works with other applications
  - Tests and measures performance
  - Determines if the architecture scales

- **Acceptance test**
  - Ensures that customers get what they want
  - Acceptance tests are not covered in this course

Notes:

<ipf>R,36: Forms of Testing</ipf>

The four forms of testing refer back to the previous page.
We’ll look at integration test in the performance chapter
Unit Testing

• When adding new features to application
  — Ensure that the changes have not broken the system
  — Testing classes in isolation is called *unit testing*
  — Confidence to make changes

• Unit testing tests an object through its public methods
  — Test validation on setters
  — Always test methods in the manner they are used
  — *Example:* `login()` must be called before `getSubject()`

• Write the unit test at the time you write the class
  — That’s when you understand the class
  — Extreme programming recommends writing the test before the class

  ✨ Automate the unit tests
  — Can’t visually examine each unit test

Notes:

<ipf>L.37: Unit Testing</ipf>
JUnit

- **Unit testing framework for Java**
  - By Erich Gamma and Kent Beck
  - A set of classes to aid in the creation of unit tests
  - `Assert` enables the automation of unit tests
    - Don't have to visually examine results
- **Create one `testCase` per class**
  - Test all non-trivial methods of the class under test
  - JUnit uses reflection to find test methods

---

**Notes:**

<i>L.38: JUnit</i>

From Ian Darwin:

http://www.cs.princeton.edu/~bwk/testing.html

Interesting paper by Brian Kernighan (of UNIX, Awk and C book fame) on "testing in teaching".

The slide shows JUnit 3.8. In JUnit 4, you don't need to extend `TestCase` (the test class is simply a POJO). The setup and teardown methods are specified by annotations, as is the test method(s).

In rev B.1 or C.1, I’ll update the course load to use JUnit 4 and then add sample code in the annotations section (Ch. 3) to show JUnit 4. We’ll leave the course load to use JUnit 3.8 code for the foreseeable future.
An Example Test Case

```java
public class PurchaseDAOTest extends junit.framework.TestCase{
    public static void main(String[] args){
        junit.textui.TestRunner.run( suite() );
    }
    public static Test suite(){
        return new TestSuite(PurchaseDAOTest.class);
    }
    PurchaseDAO dao = null;
    public void setUp(){
        dao = new PurchaseDAO();
    }
    public void tearDown(){
        dao.dispose();
    }
    public void testGetLatestPurchaseByCategory() throws Exception {
        String category = "Jazz";
        Purchase p = dao.getLatestPurchase(category);
        assertEquals( p.getCategory(), category );
    }
    public void testGetLatestPurchaseByBadCategory() throws Exception {
        Purchase p = dao.getLatestPurchase("WeDontCarryThis");
        assertTrue( p == null ); // should not throw exception
    }
}
```

**Notes:**

<ipf>R,39: An Example Test Case</ipf>

The initializations you do in the setUp are called “fixtures”

Point out that you also need to check for “null” passed in, etc.
**TestCase and TestSuite**

- **TestCase** tests all non-trivial methods of a class
  - Part of a TestSuite
  - Executing a test suite calls all the test methods of its test cases

- **In the TestCase, provide:**
  - `main()` to enable running this test individually
  - `suite()` to enable easy incorporation into another TestSuite
  - `setUp()` to perform initialization
    - Done before every test method
    - So that there are no side effects
  - `tearDown()` to perform cleanup after test

---

**Notes:**

<i>Q: Having a main in every testcase is very wasteful - some of my projects have hundreds of testcases; I am NOT going to copy/paste main (and the then-required suite) into them all - this is not something Mr BPG should be recommending :-)
Use RecursiveTestSuite to drive them all from one starting point (top level AllTests) - have a main there if you really really must.</i>

<i>A: This is the best practice guidance for new test cases.
For new test cases, click the button on Eclipse that’ll generate the main() and suite() methods for you.

If you have lots of test cases without main(), don’t treat this as a recommendation to copy-paste in the entire project, but do consider that you may need to run the tests individually. For example, when you add a method to a class, it is helpful to be able to run the unit test of just that class while developing it, not the unit test of the entire project. So, at that point, add the main() and suite() methods to its unit test. If you won’t run the unit test individually, there is no need for main()</i>
Test Methods

- For each method of the original class to be tested:
  - Have a corresponding test method in the `TestCase`
  - Use the method in the normal manner
  - Also consider bad parameters
    - Test the resulting values and/or internal state of the object
  - Do not combine lots of assertions in a single method
    - An assertion throws an exception, so the rest of the test is not performed

Notes:

<ipf>L.41: Test Methods</ipf>

Two tests – one normal behavior, and the other is error handling.

Spend time here.
Test All Required Behavior

Make sure to test all branches and all required behavior
— For example, if the specification for our DAO is that “no exception shall be thrown if the category doesn’t exist”
— If a future change breaks this requirement, the unit test will fail
— Client code can depend on the DAO not throwing a runtime exception

```java
public void testGetLatestPurchaseByBadCategory() throws Exception {
    Purchase p = dao.getLatestPurchase("WeDontCarryThis");
    assertTrue(p == null); // should not throw exception
}
```

Notes:

<ipf>R,42: Test All Required Behavior</ipf>

Note that requirements are explicitly captured in the unit tests.
Test Exception Throwing

If a method needs to throw an exception, check that as well

```java
public void testGetLatestPurchaseByNullCategory() throws Exception {
    try{
        Purchase p = dao.getLatestPurchase(null);
        fail( "Should have thrown exception on input of null" );
    } catch (NullPointerException e){
        return; // correct behavior
    }
}
```

Explicitly check bad parameters. Many software bugs show up because someone changed the way errors are handled. If you write your tests correctly, users of your code will not be surprised by changes in implicit behavior.

Notes:

<ipf>R.43: Test Exception Throwing</ipf>

Note that requirements are explicitly captured in the unit tests.

Assert calls fail; so the end behavior is the same as saying `assertTrue(false)`
Ant and Unit Tests

- Can run the unit test as a standalone application
  — Has a main() method

 Invoke junit from Ant

```xml
<target name="test" depends="build">
  <junit description="XMLUtils" fork="yes">
    <test name="com.ltree.crs516.ProjectTestSuite"/>
    <formatter type="xml" usefile="yes"/>
    <formatter type="plain" usefile="no"/>
    <classpath>
      <path refid="project.classpath" />
    </classpath>
  </junit>
</target>
```

- Can choose where the output goes
  — Plain text files, XML files, and console output are supported
  — XSL stylesheet to convert XML to HTML available

Notes:

<ipf>L,44: Ant and Unit Tests</ipf>
Composing a Project-Level Test Suite

- Can explicitly create a test suite from all the test cases in a project
  - Hierarchically, with one test suite per package
  - Run the master test suite on checked-in code as part of the automated build
    - Ensures that new check-ins have not broken your build

```java
public class ProjectTestSuite {
   public static void main(String[] args){
      junit.textui.TestRunner.run(suite());
   }
   public static Test suite(){
      TestSuite suite = new TestSuite("Rainforest");
      suite.addTest(PromotionPackageTest.suite());
      suite.addTest(DatabasePackageTest.suite());
      suite.addTest(DataminingPackageTest.suite());
      return suite;
   }
}
```

Notes:

<ipf>R,45: Composing a Project-Level Test Suite</ipf>

Lots of work and discipline
Using Ant Instead of a Test Suite

- You can use Ant to find all the test files

```xml
<target name="test" depends="build">
  <junit description="AllTests" fork="yes">
    <batchtest todir="."/>
    <fileset dir="src">
      <include name="**/*Test.java"/>
      <exclude name="**/*InContainerTest.java"/>
    </fileset>
    <formatter type="xml" usefile="yes"/>
    <formatter type="plain" usefile="no"/>
    <classpath>
      <path refid="project.classpath"/>
    </classpath>
  </junit>
</target>
```

What convention are we assuming for the unit test names?
- How to organize tests is not standardized
- Choose a convention and stick to it

Notes:

L,46: Using Ant Instead of a Test Suite

May want to mention that some people put unit tests in separate directory parallel to source. Any convention will work as long as it is consistent.
How Often Should You Unit Test?

- **Run unit test often as you develop a class**
  - Helps ensure that methods are correctly implemented

- **A unit test should be maintained throughout a project’s lifecycle**
  - It serves as an example of how a class should be used
  - It serves as proof that the class still works

- **Create an Ant task to unit test all checked-in code at least once a day**

Notes:

/ipf>R,47: How Often Should You Unit Test?/ipf>

On the last point, could work in discussion on continuous integration and Cruise Control if you like.
Maintaining a Unit Test

If someone reports a bug in an application
   — See if the bug is tested for in the relevant class
       – Run the unit test for that class
       – Does the test still pass?
           – If it fails, why did you ship the code?
           – If it passes:
               – The condition was not tested for
               – Or the bug is in another class
   — Add the example usage that triggers the bug to the unit test
       – Add assertions for this usage pattern
       – The test should now fail
   — Fix the bug
       – Ensure that the test now succeeds

• Do this every time a bug is reported

Notes:

<ipf>L.48: Maintaining a Unit Test</ipf>
If the test fails, the code should never have been shipped. Bad Programmer!
Emphasize that the bug doesn’t get fixed unless the unit test fails for this test …
Using a Class That Has a Unit Test

- **If you use a class that has an associated unit test**
  - Look at the unit test to find the anticipated ways of using that class
  - Try to use the class the same way
    - That way, your code will not break when the class is changed

- **If you use a class in a different manner from what was intended**
  - Add your usage pattern to the class’s unit test
  - The class maintainer should ensure that your assertions are not broken when the class is modified

*Over time, with bugs and reuse, your unit tests will grow, making the unit tests more and more useful. Good software practices can be self-reinforcing!*

*The time you invest in creating unit tests will be paid back many times over.*

**Notes:**

<i>Using a Class That Has a Unit Test</i>

Stress the last point. Unit tests usually start out small. They don’t have to anticipate all the conditions. With discipline, the unit tests will end up capturing the system.
Optimizing Software Development

Automating Build Management With Ant
Leveraging Advanced Features of Ant
Hands-On Exercise 2.1
Unit Testing

Hands-On Exercise 2.2

Functionality Testing
Logging
Hands-On Exercise 2.3

Notes:

<ipf>R,50: Optimizing Software Development</ipf>
Hands-On Exercise 2.2:
Applying Test-Driven Development

• In this exercise, you will learn how to
  — Integrate a JUnit test into your development environment
  — Write a new class and its unit test in tandem

Please refer to the Exercise Manual

Notes:
<i>Hands-On Exercise 2.2: Applying Test-Driven Development</i>
Optimizing Software Development

Automating Build Management With Ant
Leveraging Advanced Features of Ant
Hands-On Exercise 2.1
Unit Testing
Hands-On Exercise 2.2
▶ Functionality Testing
Logging
Hands-On Exercise 2.3

Notes:

<ipf>R,52: Optimizing Software Development</ipf>
Functionality Testing

Do not test servlets by simply visiting a Web page
— Visually examining results is very subjective
— Need an automated way to ensure that pages conform to requirements

• A functionality test is used to test use cases
  — Ensures that the individual classes in a use case work together correctly
  — Tests that a user will obtain the expected results
  – Not isolated methods of a class
  – Job of a unit test

Notes:
<ipf>L.,53: Functionality Testing</ipf>
HttpUnit

HttpUnit simplifies functional testing of Web applications

- Write servlets and web.xml as usual
- HttpUnit provides a mock servlet environment
- Use JUnit framework to test application behavior during development
- Allows parsing the returned response
  - As text
  - As XML—response.getDOM()

http://httpunit.sourceforge.net/

Notes:

<ipf>L,54: HttpUnit</ipf>
Testing a Servlet

- To use HttpUnit, create a JUnit test case:
  1. Create a mock Web conversation using HttpUnit
  2. Specify request parameters
  3. Set session variables if required
  4. Invoke servlet with specified request and obtain response
  5. Perform assertions on the response

Notes:

<ipf>R,55: Testing a Servlet</ipf>
public class CurrentPromotionsOutsideContainerTest extends TestCase {
   // main, suite, setUp methods go here
   public void testGet() throws Exception {
      // Step 1
      ServletRunner sr = new ServletRunner( new File("web.xml") );
      ServletUnitClient client = sr.newClient();
      WebRequest req =
         new GetMethodWebRequest("http://localhost/PromotionAnalyzer");
      // Step 2
      req.setParameter("retailer_id", "2");
      // Step 4
      WebResponse response = client.getResponse( req );
      String txt = response.getText();
      // Step 5
      assertTrue(txt.contains("Final score"));
   }
}

Notes:
R,56: Testing a Servlet (continued)
HttpUnit

1. **Open the Eclipse project dm21_httpunit**
   - Examine the servlet code (CurrentPromotionsServlet)
   - Examine the HttpUnit code that tests the servlet
     - CurrentPromotionsOutsideContainerTest
   - Run the Ant test task

2. **Change CurrentPromotionsServlet.java to introduce a bug**
   - Comment out the line that sets the retailer_id attribute in the XML:
     ```java
     root.setAttribute("retailer_id", new Integer(retailer_id).toString());
     ```

3. **Rerun the Ant test target**
   - ☒ Does the test now fail?
   - ☐ Did the test target deploy any .war file to Tomcat?

---

**Notes:**

<ipf>L,.57: HttpUnit</ipf>
Functionality Tests of GUI Applications

- To test GUI applications
  - Design GUIs to be thin
  - Make calls to application classes
    - Unit-test these using junit

- Ideally, the service completely hides the underlying classes
  - Called the Façade design pattern

Notes:

<ipf>R,58: Functionality Tests of GUI Applications</ipf>

We’ll be looking at more design patterns tomorrow.
Optimizing Software Development

Automating Build Management With Ant
Leveraging Advanced Features of Ant
Hands-On Exercise 2.1
Unit Testing
Hands-On Exercise 2.2
Functionality Testing
▼ Logging
Hands-On Exercise 2.3

Notes:

<R,59: Optimizing Software Development</R>
Logging Needs to Be Configurable

- Logging is essential
  - Debugging when the software produces wrong results
  - Audits of the software actions after the fact
  - Searching for potential security breaches

- Log messages should indicate their “level”
  - Debug messages
  - Significant actions (for audits)
  - Errors (for security breaches and environment problems)

- It should be possible to change logging level without recompiling software
  - For example: Debug messages in development but not in deployed software

- `System.out.println()` doesn’t meet these requirements!

One of the best ways to track a bug that occurs under rare circumstances is to selectively enable logging on a single class.

Notes:

<ipf>R,60: Logging Needs to Be Configurable</ipf>
log4j

- **log4j is an open-source Apache project**
  - Started by Ceki Gulcu
  - Now widely used
  - Optimized for speed

- **log4j allows logging code to indicate logging level**
  - Debug, Info, Warn, Error, and Fatal
  - Each log call has a level associated with it

Notes:

<ipf>R,61: log4j</ipf>
log4j Terminology

- **Logging level**
  - Only log calls above specified level
  - Can also specify ALL to log everything
  - Log messages below this threshold are ignored

- **Appender**
  - Where the message is logged
  - Console, database, file, etc.

- **Layout**
  - Format of the logged message

Notes:

<ipf>L.62: Log4j Terminology</ipf>
Logging With log4j

- Each log call is made on a logger object
  ```java
  private Logger log = Logger.getLogger( this.getClass() );
  log.info( "Computing quality of promotion" );
  ```

  The convention is to have a separate logger for each class
  - Pass the class into the logger factory
  - The programmer decides the level of each message
    - Info, in the above case

- log4j logging levels
  - Debug
  - Info: Significant action undertaken by software
  - Warn: Problem that software can work around
  - Error: Environment problem that user may be able to fix
  - Fatal: Application is going to stop
    - Throw exceptions instead

Notes:
<ipf>R,63: Logging With log4j</ipf>
Loggers Form a Hierarchy

- The name of the logger used to form a hierarchy
  - Can configure groups of classes at the same time
  - Log messages bubble up the logger hierarchy
- Use `log4j.rootLogger` to specify properties common to all logs
  - Then, specify any different properties for other loggers

Notes:

<ipf>L,64: Loggers Form a Hierarchy</ipf>
The attached circles are loggers and reference the slides following.
log4j Configuration File

- The configuration file indicates what should be logged and where
  - Specify an appender associated with the log file
  - The configuration file is a properties file
    - Default name is log4j.properties
    - log4j also allows setting in code or with an XML file

- The configuration file is placed in classpath

<ipf>R,65: log4j Configuration File</ipf>
Notes:

L,66: log4j Configuration File (continued)

Make sure to point to them that the logging code is no different whether you are logging to a console or to a database.
Additivity of Appenders

- Logs from the GUI application will go to two places
  - Console (rootLogger)
  - JDBCAppender (Ltree)
  - Log messages bubble up the logger hierarchy

- Turn off *additivity* if you want only the most specific log setting
  
  ```
  log4j.additivity.com.ltree.crs516=false
  log4j.additivity.<name-of-logger>
  ```

- Other appenders include FileAppender, SMTPAppender, SocketAppender
  - Look at API for complete list
  - Possible to write your own, for custom requirements

**Notes:**

 L,67: Additivity of Appenders
Java Logging API

- Java 1.4 introduced a hierarchical logging API (`java.util.logging`)
- Similar to log4j in many ways
  - An application gets a logger and calls its log methods
    - Typically one logger associated with every class
    - A root logger that other loggers inherit from
  - Several logging levels, although with different names:
    - FINEST, FINER, FINE, CONFIG, INFO, WARNING, SEVERE
    - ALL, OFF for convenience
  - Handlers are similar to log4j's appenders
    - LogRecord objects are sent to handlers
    - ConsoleHandler, FileHandler, SocketHandler, MemoryHandler, etc.
  - Formatters are similar to log4j's layouts
    - Log messages (called LogRecord) are formatted
      - In plain text or XML, internationalized, etc.
- Configuration files are different

Notes:

<ipf>R,68: Java Logging API</ipf>

4 ways that the two are similar
log4j vs. java.util.logging

• Which logging API should you use?
  — Reasonable people can differ
  — And argue endlessly

• log4j is recommended
  — More widely adopted
  — Optimized for speed
  — Fewer, better named levels
  — PatternLayout is more powerful than SimpleFormatter
  — No RuntimeException thrown if an appender encounters an exception
  — Configuration order of loggers does not matter
  — Resource bundles are also inherited from parent handlers

log4j:
  • Supports more appender (handler) types
  • Is more popular
  • Will run on pre-Java 1.4 virtual machines, too

java.util.logging (JUL)
  • Is a standard
  • May catch up in popularity and features
  • Supports remote setting of levels using Java Management Extensions

Notes:
/ipf>L,69: Log4J vs. java.util.logging</ipf>
Jakarta Commons Logging

- Ideally, logging is done through a lightweight, generic API
  - To hook into any logging implementation, like JDBC for database drivers

- Jakarta commons provides a generic logging API
  - Developers simply use the wrapper classes
  - The wrapper determines which logging implementation to use at runtime
    - Based on which configuration file it finds

    ```java
    import org.apache.commons.logging.*;
    Log log = LogFactory.getLog( this.getClass() );
    log.warn( "Reloading data", exception );
    ```

- Jakarta commons has a few disadvantages
  - The commons API is much more basic than log4j or java.util.logging
  - The dynamic classpath discovery causes hard-to-trace bugs
  - The wrapper adds a runtime cost to logging

- All the Apache software uses Jakarta commons
  - Since we are using Tomcat, we’ll use Jakarta commons

http://jakarta.apache.org/commons/logging

Notes:

<ipf>R,70: Jakarta Commons Logging</ipf>

Minimize the disadvantages – we’re going to use it the rest of the week!
Minimize Logging Overhead

- Logging is expensive
  - The cost of the log call is quite minor if below logging level
  - The major expense is in creating the parameters to the log method

```java
log.info( "Last purchase: " + customer.getLastPurchase() );
```

- Avoid creating the parameter to the log method if you don’t have to
  - Use a code guard

```java
if (log.isInfoEnabled()){
    log.info( "Last purchase: " + customer.getLastPurchase() );
}
```

Notes:

<ipf>L,71: Minimize Logging Overhead</ipf>
Logging Exceptions

- Don’t log when throwing or rethrowing a checked exception
  — Caller has to deal with exception anyway

```java
catch(FileNotFoundException e){
  log.error( "Unable to open " + file + " for reading");
  throw new RainForestDataUnavailableException(e);
}
```

- When logging an exception, pass exception to logger
  — Stack trace goes into log for debugging
  — The `e.getMessage()` only for end user

```java
catch(FileNotFoundException e){
  log.error( "Unable to open " + file + " for reading", e);
}
```

Notes:

<ipf>L,72: Logging Exceptions</ipf>

First point also avoids duplicate log messages
Optimizing Software Development

Automating Build Management With Ant
Leveraging Advanced Features of Ant
Hands-On Exercise 2.1
Unit Testing
Hands-On Exercise 2.2
Functionality Testing
Logging
Hands-On Exercise 2.3

Notes:

<ipf>R,73: Optimizing Software Development</ipf>
Hands-On Exercise 2.3:
Configuring and Controlling Logging at Runtime

- In this exercise, you will learn how to
  - Set up your application to use runtime log settings
  - Change runtime log settings in order to make the application more verbose
  - Send all logs to C:\crs516\logs

Please refer to the Exercise Manual

Notes:

<ipf>L,74: Hands-On Exercise 2.3: Configuring and Controlling Logging at Runtime</ipf>

Debrief:

printStackTrace problems: cannot be turned off, messages not persisted to file, NT service / Unix Daemon processes lose printed output.

Show them the Log4J config file and explain the syntax – you could use the “Loggers Form a Hierarchy” slide’s diagram along with this.

Stress the benefit of being able to log independent classes at different levels.

Show that the build.xml file copies the Log4J config file into the classpath (the build folder) – explain that Log4J looks for the config file in the classpath.

Logging much better!

Q: Why have a code guard when no string concatenation or expensive call is being made (such as in the debug statement)?

A: The idea here is similar to always using curly braces -- allows a maintainer to quickly add more information to the logging statement without remembering to add a code guard at that later stage.
Review

1. Why might you use Ant rather than Eclipse’s deployment wizards?

2. If you get a bug report, what should you fix first—the unit test that failed to catch the bug or the class that actually has the bug?

3. How would you test servlets?

4. Why are code guards recommended?

Notes:

<ipf>R,75: Review</ipf>

(1) don’t be tied into one developer’s IDE settings
2 – Unit test
3 – Use Mock Objects / Make servlet code thin and test the service façade rather than the servlet
4 - performance
Summary: Optimizing Software Development

- **Use Ant for building so as to avoid dependence on IDEs**
  - Write Ant custom tasks instead of invoking the `exec` command of Ant
  - Move absolute paths to Ant properties files
  - Avoid repetition in Ant build files by using references and imports
  - Take advantage of vendor-provided Ant tasks
  - Standardize on directory and target names

- **Unit tests should capture all the ways a class is currently used**
  - Write unit tests at the same time as the class
  - Add tests for identified bugs
  - Explicitly check class behavior on bad parameters
  - Perform functional tests of servlets outside a container

- **Use a logging API that lets you selectively enable class logging**
  - Use code guards to reduce logging overhead
  - Log or rethrow checked exceptions; don’t do both

**Notes:**

<ipf>L,76: Summary: Optimizing Software Development</ipf>

The best practices slide at the end of each chapter is the true summary slide. Spend time here. Breeze through the standard one.
Chapter Summary

In this chapter, you have learned how to

- Simplify the project build, test, and deploy environment using Apache Ant
- Apply test-driven development
  — Create unit tests to ensure that classes were coded correctly
  — Create functionality tests to ensure application behavior
- Configure runtime logging settings

Notes:

Motivate them by telling them that if they get started wrong, they may never recover.
Good practices can be self-reinforcing.
This chapter will show them how to get started “right”.
Chapter 3

Improving Code Quality

Notes:

<ipf>B,1: Improving Code Quality</ipf>
Chapter Objectives

In this chapter, you will learn to

- Improve code quality with key design patterns
- Eliminate runtime errors using generics
- Enforce type safety with canonical objects
- Increase state and behavior encapsulation
- Simplify application logic using inner classes
- Increase programmer effectiveness
- Choose between framework uses of interfaces, reflection, and class loaders

Notes:

Lots of advanced language features that they may have already heard about.
We’ll show them the best ways to use those features.
Improving Code Quality

Key OO Design Patterns

- Hands-On Exercise 3.1
- Attaining Type Safety
- Enforcing Encapsulation
- Hands-On Exercise 3.2
- Learning From Java Experts
- Creating Flexible Frameworks

Notes:

<ipf>R,3: Improving Code Quality</ipf>
Object-Oriented Analysis and Design

- When designing classes, consider both extensibility and maintainability

🌟 Consider extensibility method by method
- Which methods should be overridden by subclasses?
  - Ideally, such methods are abstract
  - Provide the typical implementation in another class
  - Classes with abstract methods are more extensible
- Which methods should not be overridden?
  - Such methods should be final

🌟 Consider maintainability for the class as a whole
- Which methods need to be public? protected? package-friendly? private?
  - The fewer the public methods, the more maintainable the class is
- Too many fine-grained getter methods may weaken encapsulation
  - Makes it hard to change the class’s implementation later

Notes:
<ipf>L.4: Object-Oriented Analysis and Design</ipf>
Designing to Interfaces

• **Abstract methods can make a class more extensible**
  — By allowing subclasses to specialize functionality
  — Appropriate subclass selected at runtime

  ```java
  CheckingAccount ca = new OverdraftCheckingAccount(accountNumber);
  boolean ok = ca.debit(amount); // what does this do?
  ```

  — This works because `CheckingAccount` specifies the `debit()` contract
  – And `OverDraftCheckingAccount` honors that contract

• **Ideally, make every method of `CheckingAccount abstract`**
  – `CheckingAccount` should be an interface
  — Makes it possible to create special implementations for any of its methods

**Notes:**

<R,5: Designing to Interfaces</ipf>

Make sure they understand polymorphism and set appropriate expectations for students who don’t quite get it.

Technically, `debit()` doesn’t need to be non-abstract, just not final. However, an abstract method is a good design practice because it is light-weight.
Design Patterns and Refactoring

- **Object-oriented design patterns** are design ideas
  - Help you think about the structure of your programs
  - Popularized in a 1995 book by Gamma, et al., a.k.a. the *Gang of Four* (GoF)
    - *Design Patterns: Elements of Reusable Object-Oriented Software*

- **Improving a design incrementally is called refactoring**
  - Changing the design without changing the functionality of an application
  - Design patterns can make this easier

That’s so 1995! The GoF book has code examples in Smalltalk and C++. Ugh!

A good book on design patterns in Java is *Head First Design Patterns.*

Notes:
<ipf>L.6: Design Patterns and Refactoring</ipf>
Refactoring Using OO

How would you refactor this repetitive code?

```java
public class TenOffPromotion {
    private ArrayList categories;
    public double getSalePrice(Item item) {
        if (categories.contains(item.getCategory())) {
            if (item.getPrice() > 20) {
                return item.getPrice() - 10;
            }
        }
        return item.getPrice();
    }
}

public class HalfOffPromotion {
    private ArrayList categories;
    public double getSalePrice(Item item) {
        if (categories.contains(item.getCategory())) {
            return item.getPrice() / 2;
        }
        return item.getPrice();
    }
}
```

Notes:

<ipf>R,7: Refactoring Using OO</ipf>

No need to step through the code. Tell them that the lines not in bold are repeated. TenOff is a fixed discount; HalfOff is a fractional discount.

I’m using direct numbers (and doubles) here, rather than BigDecimal. Otherwise, the main point will be hard to see.
With inheritance, you can reuse common code

- Put common code in the super class
- Make specializable methods abstract

Called Template Method pattern

- `getSalePrice()` is the Template Method
- The other methods hook into it

```
public abstract class Promotion {
    private ArrayList categories;
    protected abstract double getDiscountedPrice(Item item);
    public final double getSalePrice(Item item) {
        if (categories.contains(item.getCategory())) {
            return getDiscountedPrice(item);
        }
        return item.getPrice();
    }
}
```

```
public abstract class Promotion{
 private ArrayList categories;
 protected abstract double getDiscountedPrice(Item item);
 public final double getSalePrice(Item item){
     if (categories.contains(item.getCategory())){
         return getDiscountedPrice(item);
     }
     return item.getPrice();
 }
}
```

Notes:

<ipf>L.8: Refactoring With Inheritance</ipf>

Point out is-a relationship in UML here
Refactoring With Inheritance
(continued)

public class TenOffPromotion extends Promotion {
    protected double getDiscountedPrice(Item item) {
        return item.getPrice() - 10;
    }
}

public class HalfOffPromotion extends Promotion {
    protected double getDiscountedPrice(Item item) {
        return item.getPrice() / 2;
    }
}

Notes:

<ipf>R,9: Refactoring With Inheritance (continued)</ipf>
Refactoring With Aggregation

- Promotion, Discount may need to interfaces
  - New types of promotions
  - New types of discounts

🌟 Move common code to a helper object
  - Called the Strategy pattern

```java
public class CategoryBasedDiscount implements Discount {
    private ArrayList categories;
    public boolean isDiscounted(Item item) {
        return categories.contains(item.getCategory());
    }
}
```

Notes:

L,10: Refactoring With Aggregation

Point out UML has-a relationship here. Put the next slide up.
Refactoring With Aggregation (continued)

```java
public class TenOffPromotion implements Promotion{
    private Discount discount;
    public double getSalePrice(Item item){
        if (discount.isDiscounted(item)){
            if (item.getPrice() > 20){
                return item.getPrice() - 10;
            }
        }
        return item.getPrice();
    }
}
```

```java
public class HalfOffPromotion implements Promotion{
    private Discount discount;
    public double getSalePrice(Item item){
        if (discount.isDiscounted(item)){
            return item.getPrice() / 2;
        }
        return item.getPrice();
    }
}
```

Notes:

<i>Refactoring With Aggregation (continued)</i>

The problem is that “isDiscounted” in this case is only one line of code (insert joke here about being able to fit things onto a slide), so the advantage is not really obvious.

So be prepared to get a question here on why this is nearly as repetitive as the original code.
Favor Delegation Over Deep Inheritance

⭐ Delegation is more flexible than inheritance
- Can easily incorporate multiple types of discounts

• Inheritance is rigid and not easily changeable
- End up with many classes with copy–paste code
  - HalfOffJazzPromotion
  - HalfOffVolumePurchasePromotion
  - TenOffJazzPromotion
  - TenOffVolumePurchasePromotion

Notes:
<ipf>L,12: Favor Delegation Over Deep Inheritance</ipf>
Deep Hierarchies Are Hard to Maintain

- **Design more generic classes**
  - FractionalDiscountPromotion
    - Pass the discount percentage into its constructor
    - Instead of HalfOffPromotion and TwentyPercentOffPromotion
  - FixedDiscountPromotion

- **Create objects as needed**
  - Reduces number of classes that need to be maintained

```
Promotion halfOffPromotion = new FractionalDiscountPromotion("Jazz", 0.5);
```

---

**Notes:**

<ipf>R,13: Deep Hierarchies Are Hard to Maintain</ipf>
You can ask what if you need 50% or $10 off as a segue to Composite.
Write down the is-a and has-a relationship between BestOfPromotion and Promotion on this slide – the students may not be familiar with UML!
The next slide has code for how to use the Composite pattern.
The Composite Pattern in Code

Can easily form custom groupings of objects at runtime

```java
Item item = ...;
BestOfPromotion promo = new BestOfPromotion();
promo.add( new FractionalDiscountPromotion(50, Category.JAZZ) );
promo.add( new FixedDiscountPromotion(10, 20, Category.JAZZ) );
double salePrice = promo.getSalePrice(item);
```

Notes:

<R,15: The Composite Pattern in Code</R>
Ways of Creating Complex Objects

Which of the following snippets of code is more maintainable and extensible?

- Compose object every time it is needed

```java
Item item = ...;
BestOfPromotion promo = new BestOfPromotion();
promo.add( new FractionalDiscountPromotion(50, Category.JAZZ) );
promo.add( new FixedDiscountPromotion(10, 20, Category.JAZZ) );
double salePrice = promo.getSalePrice(item);
```

- Allow constructor to compose the object
  — Call constructor

```java
Promotion promo = new JazzInJunePromotion();
double salePrice = promo.getSalePrice(item);
```

- Compose the object within a factory, and obtain object by name

```java
Promotion promo = PromotionFactory.getPromotion(promotionCode);
double salePrice = promo.getSalePrice(item);
```

Notes:

<ipf>R,16: Ways of Creating Complex Objects</ipf>
Ways of Creating Complex Objects—Assessed

This is bad because creation code will be scattered all over

```java
Item item = ...;
BestOfPromotion promo = new BestOfPromotion();
promo.add( new FractionalDiscountPromotion(50, Category.JAZZ) );
promo.add( new FixedDiscountPromotion(10, 20, Category.JAZZ) );
double salePrice = promo.getSalePrice(item);
```

- This is better because creation code is only in the constructor of the class

```java
Promotion promo = new JazzInJunePromotion();
double salePrice = promo.getSalePrice(item);
```

★ This is best because users can get the promotion they need by name

```java
Promotion promo = PromotionFactory.getPromotion(promotionCode);
double salePrice = promo.getSalePrice(item);
```

Usage depends only on PromotionFactory and Promotion, making this more maintainable.

Notes:

<ipf>L,17: Ways of Creating Complex Objects—Assessed</ipf>
Factory Design Pattern

Creating classes without explicitly calling the constructor is called the **Factory pattern**
— Class.forName() makes factories easy to implement

```java
// In PromotionFactory.java
Promotion getPromotion(String couponCode)
{
    String className = config.getClassName(couponCode);
    Promotion promo = (Promotion) Class.forName(className).newInstance();
    return promo;
}
```

**Notes:**

<ipf>R,18: Factory Design Pattern</ipf>
LoyalCustomer can add another 10% off
Proxy Design Pattern

- The factory can decide which class to create
  - Doesn't need to return a JazzInJunePromotion
  - Can add a LoyalCustomerPromotion transparently, for example
    - The BestOfPromotion is a proxy to the JazzInJunePromotion

```java
// In PromotionFactory.java
Promotion getPromotion(String name){
    String className = config.getClassName(name); // what user expects
    Promotion promo = (Promotion) Class.forName(className).newInstance();
    BestOfPromotion bestof = new BestOfPromotion();
    bestof.add( promo );
    bestof.add( new LoyalCustomerPromotion(…) ); // what user gets
    return bestof;
}
```

**Notes:**

- R,19: Proxy Design Pattern</ipf>

LoyalCustomer can add another 10% off

A question I had here was “Why is this a good thing?” The developer asks for “JazzInJune” and gets back something else!

My answer: what gets returned is an enhanced (“better”) JazzInJune. Sort of like someone who clicks a “search” button expecting a search using Google, but instead gets back a search result that combines results from Google and Yahoo.
Improving Code Quality

Key OO Design Patterns

- Hands-On Exercise 3.1
- Attaining Type Safety
- Enforcing Encapsulation
- Hands-On Exercise 3.2
- Learning From Java Experts
- Creating Flexible Frameworks

Notes:

<i>L,20: Improving Code Quality</i>
Hands-On Exercise 3.1: Creating Extensible Designs With Design Patterns

- In this exercise, you will learn how to implement the following key design patterns:
  - Delegation/Strategy
  - Composite
  - Factory

Please refer to the Exercise Manual

Notes:

<i>R,21: Hands-On Exercise 3.1: Creating Extensible Designs With Design Patterns</i>
Improving Code Quality

Key OO Design Patterns

Hands-On Exercise 3.1

Attaining Type Safety

Enforcing Encapsulation

Hands-On Exercise 3.2

Learning From Java Experts

Creating Flexible Frameworks

Notes:

<L,22: Improving Code Quality</L>
Enforcing Limits on Parameter Values

- Often have methods whose input range is limited

```java
// PurchaseDAO.java
List<Purchase> getPurchasesByCategory(String category);
```

How can PurchaseDAO enforce:
- Input string can only be: “Jazz”, “Hip-Hop”, “Rock”, etc.?
- Not just any arbitrary string

Notes:

<i>ipf>R,23: Enforcing Limits on Parameter Values</i>
The Java `enum` Pattern

Use canonical objects to enforce a limited set of values

```java
public class Category {
    public static final Category HIPHOP = new Category("Hip-Hop");
    public static final Category JAZZ = new Category("Jazz");
    public static Category getCategory(String category) {
        if (category.equals(HIPHOP.category)) {
            return HIPHOP;
        } else if (category.equals(JAZZ.category)) {
            return JAZZ;
        } else {
            throw new IllegalArgumentException("Unknown category: "+ category);
        }
    }
    private String category;
    private Category(String category) {
        this.category = category;
    }
}
```

**Notes:**

<ipf>L.24: The Java enum Pattern</ipf>

Explain the stuff in bold.

(a) Limited set of objects

(b) Factory method

© private constructor
Java `enum` Keyword

- The `enum` keyword in Java 5 handles some of the drudgery
  Provides a `values()` method that can simplify coding

```java
public enum Category {
    HIPHOP("Hip-Hop"),
    JAZZ("Jazz");
    public static Category getCategory(String category) {
        for (Category cd : values()){
            if (cd.category.equals(category)){
                return cd;
            }
        }
        throw new IllegalArgumentException("Unknown category: " + category);
    }
    private String category;
    private Category(String category) {
        this.category = category;
    }
}
```

**Notes:**

<ipf>L,24: The Java enum Keyword</ipf>

What enum does … gives you the static final fields automatically and also creates a values() method
Using the Java enum Pattern

- **String validated once by factory**
  - Require type-safe Category objects everywhere else
  - Change the PurchaseDAO to accept only Category objects
  - This way, only valid categories will come to the DAO

```java
// PurchaseDAO.java
List<Purchase> getPurchasesByCategory(Category category);

// somewhere else
String userInput = ...;
Category category = Category.getCategory(userInput);
List<Purchase> purchaseList =
    purchaseDAO.getPurchasesByCategory(category);
```

**Notes:**

<ipf>R,25: Using the Java enum Pattern</ipf>
Implications of Canonical Objects

 Canonical objects do not need to override equals() — The == operator is enough

 Why?

 Canonical objects should be designed to be immutable — Otherwise, nasty side effects between methods

Notes:

<ipf>L,26: Implications of Canonical Objects</ipf>

Also makes sure that Canonical objects are not serializable
And that all fields are declared final.
Downcasts Are Unsafe

• Prior to Java 1.5, code using collections needed an explicit downcast

```java
ArrayList purchases = new ArrayList();
purchases.add( new Purchase() );

// later
Purchase p = (Purchase) purchases.get(0);
int retailer_id = p.getRetailerId();
```

• What happens if someone puts a Promotion into the ArrayList?

 hann Will the code compile?

 hann What happens at runtime?

Notes:

<ipf>R,27: Downcasts Are Unsafe</ipf>
Java 5 Generics

- **Java 1.5 introduced generics that provided type safety**
  - Generics are also part of other OO languages, such as C++ and C#

- **Generics allow Java 5 collections to be used without casting**
  - And ensure that Promotion objects are not added to this list by mistake

```java
// when creating:
ArrayList<Purchase> purchases = new ArrayList<Purchase>();
purchases.add( new Purchase() );

// later
Purchase p = purchases.get(0);
int retailer_id = p.getRetailerId();
```

**Notes:**

<R,28: Java 5 Generics</R>

Show them generics in a context they may already be familiar with.
Then say they should think about where generics would be useful in their
Applications.
Type Safety When Operating on Generic Types

- Java is an object-oriented language
  - Supports inheritance and polymorphism
  - Permits the design of classes that operate on generic types

```java
// In Purchase.java
void applyPromotion(Promotion p){
    this.salePrice = p.getSalePrice( this );
    this.promotion = p;
}
// somewhere else
FixedDiscountPromotion p = new FixedDiscountPromotion(...);
purchase.applyPromotion(p);
```

- However, polymorphism can lead to loss of type information

```java
// somewhere else
Promotion p = purchase.getPromotion(); // don't know what type this is!
```

Notes:

<ipf>L,29: Type Safety When Operating on Generic Types</ipf>

Ask them why item.getPromotion() has to return a Promotion object. Why not a FixedDiscountPromotion?
Type Safety and OO

- Ideally, user should not have to know exact type of object
  - Promotion should declare every method that users could possibly need
    - Subclasses implement/override these methods
    - “Design to interfaces”

```java
Promotion p = purchase.getPromotion(); // don't need to know exact type!
Date startDate = p.getStartDate();
boolean isValid = p.isValidForItem( ... );
```

- Sometimes, users need to treat items differently
  - Based on the exact promotion applied to that item

```java
// use-case dictates cast will succeed
FixedDiscountPromotion p = (FixedDiscountPromotion) purchase.getPromotion();
double discountAmount = p.getCouponAmount();
```

Notes:

<ipf>R,30: Type Safety and OO</ipf>

The first bullet point, of course, is “design to interfaces”

No getCouponAmount() if the promotion is for 30% off …
Specialization Using Generics

- Purchase can be specialized according to the type of Promotion

```java
// when creating:
Purchase<FixedDiscountPromotion> item = new Purchase<FixedDiscountPromotion>(...);

// somewhere else
purchase.applyPromotion(new FixedDiscountPromotion(...));

// later
FixedDiscountPromotion p = purchase.getPromotion();
```

- To enable specialization, Purchase is defined as follows:

```java
public class Purchase<X extends Promotion>{
    private X promotion;
    public void applyPromotion(X promo){ ... }
    public X getPromotion(){ return promotion; }
}
```

- X, T, and E are commonly used to represent the template parameter
- Can be specialized by any class that extends/implements Promotion

Notes:

<i>ipf>L.31: Specialization Using Generics</i>

I like to look at the usage first before figuring out how to get there.
If you like to show them how it is, and then how it works, simply cover the second point first.

If you simply say Purchase<X> then X extends Object

XXX: split over two slides
What Happened to the Casts?

- Generics in Java are only about type safety
  - Everything that can be done with generics can be done without them
  - At the cost of introducing downcasts into the code

- Generics do not get rid of the casting
  - The casting is still happening, just not in your source code
  - The Java compiler inserts the dynamic casts
  - Done this way so that the Java byte code format is not changed

Notes:

<ipf>R,32: What Happened to the Casts? </ipf>
Generics and Inheritance

- Inheritance can get tricky
  - Cannot pass `List<TenOffPromotion>` to method that expects `List<Promotion>`

```java
// in Purchase.java
void applyPromotions( List<Promotion> promos ){
    // stuff here
}
// somewhere else
List<Promotion> promos = new List<Promotion>();
promos.add( new TenOffPromotion() );
purchase.applyPromotions(promos);

// in Purchase.java
void applyPromotions( List<Promotion> promos ){
    // stuff here
}
// somewhere else
List<Promotion> promos = new List<Promotion>();
promos.add( new TenOffPromotion() );
purchase.applyPromotions(promos);
```

Notes:

<i>ipf>R,32: Generics and Inheritance </ipf>

So, declare list as `List<Promotion>`, not `List<TenOffPromotion>`
Generics Do Now

1. Run the Ant target `run_generics_buggy` in Eclipse project `dm31_uniqueList`
   a. Examine `PlainMain.java` to find the problem
      — In package `com.ltree.crs516.demo.generics`
      — Is the stack trace informative?
   b. Open `GenericsMain.java` and try to introduce the same bug
      — Does the code compile?

2. Examine `UniqueList.java`
   — An example collection of Purchases or Promotions
      — Maintains unique elements (a set) in the order they were added
      — Like `LinkedHashSet` in the collections API, except not by `hashCode`
      — Shows how to implement custom generic class
   — Note the interface called “ID”
      — Why is this required?

Notes:

<ipf>L,33: Generics Do Now</ipf>

Answer to last question: no `hashCode()` method, so how to know something is unique?
Improving Code Quality

Key OO Design Patterns

Hands-On Exercise 3.1

Attaining Type Safety

Enforcing Encapsulation

Hands-On Exercise 3.2

Learning From Java Experts

Creating Flexible Frameworks

Notes:

<ipf>R,34: Improving Code Quality</ipf>
Encapsulation

- One of the major benefits of an OO language is encapsulation
  - Make all fields private
  - Permit modifications only through methods
    - Helps retain the ability to change the structure of your program
    - Helps retain data integrity

```java
public class BrokerageAccount {
    private BigDecimal cash;
    private BigDecimal margin;

    public void setCash(BigDecimal a) {
        // validate, log, etc.
        this.cash = a;
    }

    public void setMargin(BigDecimal a) {
        // validate, log, etc.
        this.margin = a;
    }

    public BigDecimal getCash() {
        return this.cash;
    }

    public BigDecimal getMargin() {
        return this.margin;
    }
}
```

**Notes:**

<ipf>L,35: Encapsulation</ipf>

Tell them how great these setter methods are. Next slide is the downsides
Fine-Grained Getter and Setter Methods

- Fine-grained getter/setter methods
  - Make code hard to maintain

- For example, consider that the application needs to show the user the effect of a trade
  - Save current state of object
  - Perform trade
  - If user cancels, set state back

What is the problem with this code?

```java
BrokerageAccount acct = …;
BigDecimal cash = acct.getCash();
BigDecimal margin = acct.getMargin();
acct.performTrade(trade);
boolean ok = promptUser();
if (!ok){
    acct.setCash(cash);
    acct.setMargin(margin);
}
```

**Notes:**

<ipf>R,36: Fine-Grained Getter and Setter Methods</ipf>

Answer: (a) code hard to maintain
User has to know the two properties – the class’ data structure is not encapsulated
What if we also want to add the number of trades to the BrokerageAccount so that we can set commission levels?
Use Inner Classes to Encapsulate State

- A static inner class can be employed to encapsulate state
  - Other classes can get/set the state, but can’t modify the internals
    - Encapsulating state this way is called the Memento pattern

```java
public class BrokerageAccount {
  private Balances balances;
  public static class Balances {
    private BigDecimal cash;
    private BigDecimal margin;
    private Balances(BigDecimal cash, BigDecimal margin) {
      this.cash = cash; this.margin = margin;
    }
    public Balances getBalances() { return balances; }
    public void setBalances(Balances other) { balances = other; }
  }
  BrokerageAccount.Balances balances = acct.getBalances();
  acct.performTrade(trade);
  boolean ok = promptUser();
  if (!ok){
    acct.setBalances(balances);
  }
}
```

**Notes:**

- Use Inner Classes to Encapsulate State

Point out the benefits for client code – they deal with the black box Balances.
The outer class implementor can change Balances however and client code will work okay.
Yes, private fields and constructors of inner class can be invoked from outer class

Why static? It allows you to import the state of one object to another.
Just ‘cos someone is holding on to a copy of your state doesn’t mean you don’t get GC’ed
Use Inner Classes to Retain Encapsulation

Use inner classes for coarse-grained getter/setter methods
  — An inner class is defined within the scope of another class
  — The inner class and the outer class have mutual access to the private fields and methods of the other
    – Preserve encapsulation

• Can use either static or non-static inner classes to preserve encapsulation
  — Use static inner classes when different state will be externalized
  — Otherwise, tie to the instance of the outer object they were created in
    – By not declaring the inner class as static

Inner classes give you all the benefits of using a separate object without any of the disadvantages involved in breaking encapsulation.

Notes:
<ipf>L.38: Use Inner Classes to Retain Encapsulation</ipf>
Encapsulation and Interfaces

- Often a class needs to implement interface to plug in to framework
  - The Strategy pattern

- For example:
  - An exit button may need to call `exit()` method on main application

```java
public class MainFrame extends JFrame {
    private MainFrame() {
        JButton exitButton = new JButton("Exit");
        getContentPane().add(exitButton);
        exitButton.addActionListener(
            new ExitActionListener(this,"separate class");
        }
    }
    /** Have to make this public so that external class can call it. */
    public void exit() {
        this.dispose();
        System.exit(0);
    }
}
```

Notes:

<ipf>R,39: Encapsulation and Interfaces</ipf>
Encapsulation and Interfaces (continued)

The MainFrame must expose its `exit()` method
— Must pass reference to itself to listener

```java
public class ExitActionListener implements ActionListener {
    MainFrame frame;
    String message;
    public ExitActionListener(MainFrame frame, String message) {
        this.frame = frame;
        this.message = message;
    }
    /** exits application */
    public void actionPerformed(ActionEvent evt) {
        System.out.println(message);
        frame.exit();
    }
}
```

Notes:

<ipf>R.40: Encapsulation and Interfaces (continued)</ipf>
Use Inner Classes to Reduce Visibility

- **By making ExitListener an inner class:**
  You can reduce visibility of the `exit()` method

```java
public class MainFrame extends JFrame {
    private MainFrame() {
        super("Static inner");
        JButton exitButton = new JButton("Exit");
        getContentPane().add(exitButton);
        exitButton.addActionListener(new ExitActionListener(this, "static inner class"));
    }
    private void exit() {
        this.dispose();
        System.exit(0);
    }
    private static class ExitActionListener implements ActionListener {
        // as in previous slide
    }
}
```

*Notes:*

<ipf>R,41: Use Inner Classes to Reduce Visibility</ipf>
Increasing Maintainability

- The inner class maintains a reference to `MainFrame`
  — So that it can call the `exit()` method

🌟 Java will do this for us automatically, which is less error-prone

```java
public class MainFrame extends JFrame {
    private MainFrame() {
        // as before
    }
    private void exit() {
        // as before
    }
    private class ExitActionListener implements ActionListener {
        String message;
        public ExitActionListener(String message) {
            this.message = message;
        }
        public void actionPerformed(ActionEvent arg0) {
            System.out.println(message);
            exit();
        }
    }
}
```

Notes:

<i>ipf>L.42: Increasing Maintainability</ipf>
Further Increasing Maintainability

• Can streamline the code further
  ★ The inner class is not needed anywhere else
  – No need to give it a name (anonymous)

```
public class MainFrame extends JFrame {
    private MainFrame(){
        super("anonymous inner");
        JButton exitButton = new JButton("Exit");
        getContentPane().add(exitButton);
        exitButton.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent evt) {
                System.out.println("anonymous inner");
                exit();
            }
        });
    }

    private void exit() {
        this.dispose();
        System.exit(0);
    }
}
```

Notes:

<ipf>R,43: Further Increasing Maintainability</ipf>
Summary: Best Practices for Inner Classes

- All types of inner classes can be used to:
  - Reduce visibility of outer class methods
  - Retain encapsulation

- Static inner class has no reference to outer class
  - Use to encapsulate state (Memento pattern)
    - Can transfer state between different objects

- Non-static inner class has automatic reference to outer class
  - Adapt a class to an interface expected by frameworks

- Anonymous inner classes are non-static inner classes with no name
  - Use when inner class will not be reused
    - Can always refactor into named inner class later
  - Simplifies code and makes it more readable

- Local inner classes, in most cases, are just poor programming practice

Notes:

<ipf>L.44: Summary: Best Practices for Inner Classes</ipf>
Improving Code Quality

Key OO Design Patterns

Hands-On Exercise 3.1

Attaining Type Safety

Enforcing Encapsulation

Hands-On Exercise 3.2

Learning From Java Experts

Creating Flexible Frameworks

Notes:

<ipf>R.45: Improving Code Quality</ipf>
Hands-On Exercise 3.2:
Retaining Encapsulation With Inner Classes

- In this exercise, you will improve the encapsulation of the returned values from the PurchaseDAO
  - This will make it possible to change the return type of the DAO in later exercises

Please refer to the Exercise Manual

Notes:

Q: In this exercise you encourage them to replace the use of a parameterised Collection with an inner class. I can't understand how this can be presented as best practice. Would you do this in your projects? I've never done it and would find it hard to encourage students to adopt this technique. Another downside of this approach becomes evident in later exercises where you cannot use the new Java 5 loop to iterate through purchases as you would have been able to before this change was made!

A: The exercise is not REPLACING the List<Purchase> with an inner class. It is ENCAPSULATING the List<Purchase> with an inner class. Besides the benefit of being able to change the return type completely, another benefit of doing this is to be able to easily add more return values. As another example of this, see the AnalysisResult class in the datamining package. There too, it is not that we are replacing "double" with an AnalysisResult. We are encapsulating the return value so that it is easier later on to add other return values besides the quality. And yes, you do lose a little bit of the syntactical sugar of Java 5 loop syntax. However, the maintenance benefits of encapsulating return values is enormous. I do this all the time for core logic, not necessarily with inner classes of course, but encapsulating results is a definite best practice. If you've never done this, may I suggest that you start to do so? :)

<ipf>L.46: Hands-On Exercise 3.2: Retaining Encapsulation With Inner Classes</ipf>
Improving Code Quality

Key OO Design Patterns
Hands-On Exercise 3.1
Attaining Type Safety
Enforcing Encapsulation
Hands-On Exercise 3.2
Learning From Java Experts
Creating Flexible Frameworks

Notes:

<ipf>R,47: Improving Code Quality</ipf>
When and What to Throw

 Exceptions are only for exceptional cases
 — For which caller would have to break out of control flow
 — In routine circumstances, don’t throw exceptions
 — For example, consider java.io
   — FileNotFoundException
   — BufferedReader: At end of file, simply returns null

 Wrap low-level exceptions in higher-level ones
 — DataUnavailableException, not FileNotFoundException
 — Preserves layer abstraction

Notes:

<ipf>L.48: When and What to Throw</ipf>
Another example: DOM parsing
Asking for an attribute that doesn’t exist simply returns null
Parsing an invalid XML document throws an exception
Runtime vs. Checked Exceptions

🌟 Throw RuntimeException for things from which caller can’t recover
   — For example: unable to connect to enterprise database
   — Goes all the way up the call stack

🌟 Throw checked exception only for recoverable problems
   — Clutters up code otherwise

Notes:

<ipf>R,49: Run Time vs. Checked Exceptions</ipf>
Sensible middle ground between
C#: no checked exceptions
Sun: only JVM is allowed to throw RuntimeException
clone() Is Broken

- **No easy way to ensure correct clone() behavior**
  - Need to call super.clone()
    - Then fix anything not handled by super class
  - But should behave like constructor, not calling overridden methods
    - Otherwise original and copy may be different
    - Not easy to do this
  - One way may be to serialize object, then read it back in
    - Problem with transient fields
    - And references to canonical objects
  - Do not implement Cloneable
    - Rarely worth the trouble

**Notes:**

<!--lint:ok:style=xml -->
<ipf>L.50: clone() Is Broken</ipf>
See Joshua Bloch (Effective Java) on this subject
Providing Copy Behavior

- **To provide copying behavior for a class**
  - Provide a static factory method
    - Invoke constructor in method
    - With copies of objects if necessary
  - Can also provide a constructor that takes original object

```
public static BestOfPromotion newInstance(BestOfPromotion original){
    BestOfPromotion result = new BestOfPromotion();
    // The Collections classes provide constructors that take
    // original object.
    // If Promotion is designed to be immutable, no need to make
    // defensive copies
    result.promotions = new ArrayList<Promotion>(original.promotions);
    return result;
}
```

- **Disadvantage: Static methods and constructors cannot be overridden**
  - No way to get clone of a Promotion without knowing its type

**Notes:**

<ipf>R,51: Providing Copy Behavior</ipf>

The collections API takes the copy constructor route. May want to point that out to them.
Characteristics of a Correct `equals()` Method

A correct `equals()` implementation should be:

- Symmetric
  - If A equals B, then B should equal A
- Transitive
  - If A equals B and B equals C, then A should equal C
- Consistent
  - As long as A and B are not modified, A equals B should keep returning the same value

The implementation should also handle these two special cases:

- Parameter value of `null`
- Object identity

Notes:

<ipf>L.52: Characteristics of a Correct equals() Method</ipf>
Implementing `equals()`: An Example

- Usually, these requirements are no problem
  — Just follow the steps in this template:

```java
// Purchase.java
@override
public boolean equals(Object other){
    // Step 1: Check whether same object
    if ( this == other ){
        return true;
    }
    // Step 2: Check type and cast
    if ( !(other instanceof Purchase) ){
        return false;
    }
    Purchase purchase = (Purchase) other;
    // Step 3: Check all fields
    return ( purchase.listPrice.equals(listPrice) &&
            purchase.retailerId.equals(retailerId) );
}
```

**Notes:**

<R,53: Implementing `equals()`: An Example</R>

Will talk about `@Override` later

Can use `other.getClass().equals( this.getClass() )` to get consistently “false” when a subclass is provided.
Overriding an `equals()` Implementation

- **Overriding an `equals()` implementation cannot be done right**
  - Symmetry and transitive conditions not possible to meet
    - Comparing `Purchase` to `DiscountPurchase` will invoke `Purchase`'s method
    - Comparing `DiscountPurchase` to `Purchase` will invoke `DiscountPurchase`'s method

- **Solutions:**
  - Use delegation, not inheritance
    - `DiscountPurchase` has a `Purchase` and a `Discount`
  - Never mix `Purchase` and `DiscountPurchase` objects in the same collection
  - Use `getClass()` to ensure consistent result

```java
if ( !((other.getClass()).equals(this.getClass()))) {
    return false;
}
```

**Notes:**

- Overriding an `equals()` Implementation
- Easy to enforce `List<DiscountPurchase>` should not contain plain `Purchase`
- Otherway round is impossible
Override `hashCode()`

- The usual reason to provide `equals()` is for use in collections
  - Many collections also make use of `hashCode`
  - Therefore, provide `hashCode()` whenever you provide `equals()`

- Equal objects should have equal hash codes
  - Use the same fields as `equals()`

Notes:

<ipf>R,55: Override `hashCode()`</ipf>
Example `hashCode()` Implementation

- Follow this template when implementing `hashCode`:

```java
// Purchase.java
@Override
public int hashCode(){
    int result = 0;
    final int FACTOR = 13; // traditional to choose a prime number > 3
    // do this for all primitives
    result = result * FACTOR + retailerId;
    // do this for all Object references, shown here on BigDecimal
    if ( listPrice != null ){ // do this for object references
        result = result * FACTOR + listPrice.hashCode();
    } else {
        result = result * FACTOR;
    }
    // repeat for other fields
    return result;
}
```

Notes:

<ipf>L.56: Example `hashCode()` Implementation</ipf>
Improving Code Quality

Key OO Design Patterns

Hands-On Exercise 3.1

Attaining Type Safety

Enforcing Encapsulation

Hands-On Exercise 3.2

Learning From Java Experts

Creating Flexible Frameworks

Notes:

<ipf>R,57: Improving Code Quality</ipf>
Writing Framework Code

- **Frameworks provide support for software development**
  - Libraries, tools, execution environments, etc.
  - Context for software reuse

- **When writing frameworks, you need to create flexible software**
  - May need to work with arbitrary objects
  - May need to execute client-supplied code after application startup
  - Users may need to control and configure framework

- **Java provides mechanisms to aid in the creation of flexible frameworks**
  - Interfaces
  - Reflection
    - JavaBeans
    - Annotations
  - Dynamic class loading

Notes:

<ipf>L,58: Writing Framework Code</ipf>
What we’re going to look at in this chapter.
XDoclet is temporary
Best Practices in Designing and Using Frameworks

• This section examines best practices when:
  — Designing a framework
    – Catalog of Java capabilities available
    ☑️ When to use what
  — Making use of a framework
    ☑️ Understanding a designer’s motivation can enable effective framework usage

• Not focused on how to use these tools
  — Consult documentation on these topics if not familiar with them

Notes:

<ipf>R,59: Best Practices in Designing and Using Frameworks</ipf>
Notes:

<i>L,60: Design to Interfaces</i>

Draw diagram here
Leveraging Reflection

⚠️ Take advantage of reflection when interfaces are impractical
- Define naming convention
- Load up object and look for methods that meet naming convention
- For example: JUnit and testX() methods

// in framework, operating on user-supplied object "obj"
Method[] methods = obj.getClass().getMethods();
for (Method m : methods) {
    if (m.getName().startsWith("test") ){
        // invoke method with no parameters
        m.invoke(obj, (Object[])null);
    }
}  

⚠️ Don't overdo reflection
🌟 Interfaces should always be your first option

Notes:
<ipf>R.61: Leveraging Reflection</ipf>
Impractical to have lots of inner classes each implementing one test method each
Reflection for Getters and Setters

If framework needs to look for getters and setters, use JavaBeans
   — Allows users flexibility in naming

A JavaBean is simply a class that follows certain conventions
   — Must have no argument constructor
   — Typical naming convention for:
     — Getters and setters
       — void setSalePrice(double salePrice)
       — double getSalePrice()
     — Listeners
       — addXListener()
   — Must implement java.io.Serializable

A class that doesn’t follow naming conventions can be a JavaBean if it has a BeanInfo class. For a JavaBean X, provide an XBeanInfo class.

Just use Java naming conventions.

Notes:
<ipf>L,62: Reflection for Getters and Setters</ipf>
Extracting Properties From a JavaBean

- Framework can introspect a JavaBean to get properties of that class
  - JavaBeans were originally designed for GUI builders and IDEs
  - Introspector works on bean properties
  - Name of getter method does not need to be hard-coded

```java
// Step 1
Object data = ...;
BeanInfo info = Introspector.getBeanInfo(data.getClass(), Object.class);
PropertyDescriptor[] props = info.getPropertyDescriptors();
for (int i=0; i < props.length; ++i){
  // Step 2
  String name = props[i].getName();
  Method m = props[i].getReadMethod();
  Object value = m.invoke(data, (Object[]) null);
  map.put(name, value);
}
```

Notes:

<ipf>R,63: Extracting Properties From a JavaBean</ipf>

java.beans.Introspector: be familiar with it
Annotations

If framework provides facility not central to class, use metadata
- Information about the code
- Javadoc metadata such as deprecated

- Java 5 provides the ability to annotate code
  - Can be read from the source code or from runtime
  - Depending on retention policy

```java
@CouponCode(name="JJ23xY4W", validUntil="yesterday")
public class JazzInJunePromotion extends FractionalDiscountPromotion{
    @Override
    public double getSalePrice(Purchase p){
        ...
    }
}
```

Method annotation  Class annotation  Compile-time constants only

Notes:

<i>ipf>R,64: Annotations</i>

Keywords such as transient, volatile are arguably metadata too
Not central to class behavior – instrumentation, documentation, etc
How Frameworks Can Use Annotations

• To create a custom annotation, define an annotation interface

```java
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface CouponCode {
    String name();
    String validUntil() default "tomorrow";
}
```

• Find out if a class has a runtime annotation and gets its value via reflection

```java
// somewhere else
Method[] methods = obj.getClass().getMethods();
for (Method m : methods){
    if (m.isAnnotationPresent(CouponCode.class)){
        CouponCode couponCode = m.getAnnotation(CouponCode.class);
        String validUntil = couponCode.validUntil();
        // do something with these ...
    }
}
```

Notes:

<ipf>R,66: How Frameworks Can Use Annotations</ipf>

RetentionPolicy.SOURCE annotations are completely ignored by Java. Essentially stuff like @author and @version

RetentionPolicy.CLASS annotations are used only during compilation. Essentially stuff like @override

Finding annotations via reflection works only for the third type
@Override Can Catch Subtle Bugs

@ovErriDe is used by the compiler to verify that the method has the same signature as method in superclass

— Extremely useful
— If superclass’s method is changed, the subclass won’t compile anymore
— Can be a very subtle bug to catch if you are not using @Override

Notes:

<ipf>R,65: @Override Can Catch Subtle Bugs</ipf>
When Is Code Generation Useful?

- **Using frameworks often requires many support classes**
  - A code generation engine can help reduce drudgery

- **XDoclet is a code-generation engine**
  - Open-source project
  - Commonly used with EJBs to create
    - Deployment descriptors
    - Home interfaces
    - Transfer objects

- **XDoclet can be used for custom code generation, too**
  - From custom tags (annotations) on Java files

---

Notes:

<i>When Is Code Generation Useful?</i>
How Code Generation Works

package com.ltree.crs516;
/**
 * @promotion couponCode="NewYearSale"
 */
public class Promotion {
...
}

<XDtClass:forAllClassTags tagName="promotion">
package <XDtPackage:packageName/>;
class <XDtClass:classTagValue tagName="promotion" paramName="couponCode"/>Promotion
extends <XDtClass:className/>{
}
</XDtClass:forAllClassTags>

package com.ltree.crs516;
class NewYearSalePromotion extends Promotion{
}

Notes:
<i>ipf>L.68: How Code Generation Works</i></ipf>
The top two are inputs
The third is an output
Dynamic Loading of Classes

- Frameworks often have to execute client-supplied code at runtime
  - For example:
    - Servlet engines execute servlets
    - Ant executes custom tasks
  - Must load client-supplied class from persistent store
  - The class is not already part of framework’s class path

- Java supports dynamic loading of classes
  - `Class.forName(className)` if byte code is already present in classpath
  - More complicated if the byte code is not in classpath
    - Framework has to find and read the byte code

Notes:

<ipf>R,69: Dynamic Loading of Classes</ipf>
The Principles of Class Loading

1. Classes are loaded as they are needed

2. Certain classes are loaded by predefined class loaders
   - The bootstrap class loader loads all classes that are part of the core Java Runtime Environment (JRE)
     - Boot class path
   - The system class loader loads classes that are in the CLASSPATH
   - Can never have more than one definition for such classes in a JVM

3. Loading of other classes has to be initiated by the code
   - Must be loaded by a custom class loader
   - Specify where to load the classes from
     - Could be loaded from a network location
     - AppletClassLoader loads classes from hosting Web site

4. A class loader gets to load a class only if its parent cannot
   - First bootstrap class loader gets a shot
   - Then system class loader
   - Finally, custom class loaders in framework-specified order

Notes:

Make sure to make explicit the distinction between the classpath of Tomcat (for example)
   -- the classpath OF the servlet engine
And the directories/places where Tomcat looks for classes
   -- not in Tomcat’s classpath e.g: common/lib, common/classes etc.
   -- custom class loaders that Tomcat can arrange however it wants
Multiple Versions of the Same Class

- Can have multiple versions of the same class in the JVM
  - If not loaded by bootstrap or system class loaders
  - Classes loaded by separate custom class loaders can never interact
    - Used by servlet engines to run different Web applications

- Most common confusion in class loading by servlet containers:
  - Classpath of Tomcat (bootstrap and system class loaders)
  - The path that Tomcat searches to find classes (custom class loader)

Notes:
<i>Multiple Versions of the Same Class</i>
Best Practices When Using Dynamic Frameworks

- Safeguard against class-loading problems when using frameworks that load classes at runtime
  - Application servers, servlet engines, Ant, Eclipse, etc.

🌟 Keep the framework’s classpath clean
  - For example: classpath of Tomcat when it starts
  - Don’t add extraneous .jar files to the classpath
  - If your application references a .jar file that is in framework’s classpath:
    – Some classes will load classes from framework’s .jar file
    – Some classes will load classes from yours
    – Classes loaded by separate class loaders cannot communicate

🌟 Never set classpath as an environment variable
  - Framework may pick it up
  - Start applications by explicitly providing classpath as command-line option
    – Or by using Ant

Notes:

May not think about Ant as a framework, but adding jar files to Ant’s lib will pose same problems.
Class Loader Tunnel

- What if you need classes loaded by different class loaders to interact?
  - Need a class loader tunnel

- All classes extend `java.lang.Object`
  - Which has to be loaded by the bootstrap class loader: Why?
  - All objects can be upcast to `Object` and used by any other class
  - Use reflection to call methods on classes loaded by some other class loader

Notes:

<ipf>R,73: Class Loader Tunnel</ipf>

Draw the is-a relationship here to form the “tunnel”
Extending Factory

- Can extend factory to look for new Promotions on a Web site
  - Load these classes at runtime
  - Apply them to a purchase using reflection

- Your instructor will now point out the salient points of a custom class loader
  - Open Eclipse project dm32_classloader

Notes:

<ipf>R,74: Extending Factory</ipf>

Be quite familiar with this. Lots of tricky things (I had to learn as I went along; didn’t realize much of this.). You could point out

A) Ant build file to have two locations for .class files. One of them outside classpath
B) Can’t cast the returned object to Promotion (since Promotion has been loaded by the current project’s classloader)
C) Have to use reflection to call methods
D) can’t pass primitives; so getSalePrice() has to take Double.
E) the implementations of getSalePrice() in PromotionA, B, C demonstrate Java 5 autoboxing
Review

1. Why are design patterns important to know?

2. What is the main benefit of generics in Java?

3. When should you use reflection?

Notes:
<ipf>L,75: Review</ipf>
(a) capture expert knowledge
(b) type safety
© when interfaces are impractical or if framework is not central to class behavior
Summary: Improving Code Quality

• Consider both extensibility and maintainability in your design
  — For extensibility: Supply interfaces for all business objects
  — For maintainability: Restrict public access and use coarse-grained methods

• Apply design patterns to solve common design problems
  — Favor delegation to deep inheritance
  — Reuse code within an inheritance hierarchy using Template Method
  — Move common code off hierarchy into helper classes (Strategy pattern)
  — Create flexible groupings of subclasses with the Composite design pattern
  — Centralize object creation in a factory for future flexibility
  — Add behavior transparently within factories using a Proxy

• Design for type safety and encapsulation
  — Incorporate generics to avoid runtime errors
  — Enforce limits on parameter values using canonical objects
  — Use static inner classes to provide coarse-grained setters (Memento)
  — Use inner classes to reduce the visibility of methods
  — Simplify adaptation to interfaces using anonymous inner classes

Notes:

<ipf>L,76: Summary: Improving Code Quality</ipf>
Summary: Improving Code Quality (continued)

- **Throw exceptions only when caller has to stop routine processing**
  - Reserve checked exceptions for the case when caller can fix problem
  - Preserve layer abstraction

- **Be aware of implicit contracts in the Java core API**
  - Prefer factory copy to implementing `Cloneable`
  - Overriding `equals()` cannot be done right, so use delegation
  - Override `hashCode()` when overriding `equals()`

- **When designing frameworks, use interfaces to provide extension points**
  - Employ reflection only when interfaces will be impractical
  - For reflecting on getters and setters, use JavaBeans
  - For providing facility not central to code, reflect on annotations
  - Use dynamic class loading to execute client-supplied behavior

- **When using frameworks**
  - Follow JavaBean naming conventions
  - Keep classpath clean

Notes:

<ipf>L,77: Summary: Improving Code Quality (continued)</ipf>
Chapter Summary

In this chapter, you have learned how to
- Improve code quality with key design patterns
- Eliminate runtime errors using generics
- Enforce type safety with canonical objects
- Increase state and behavior encapsulation
- Simplify application logic using inner classes
- Increase programmer effectiveness
- Choose between framework uses of interfaces, reflection, and class loaders

Notes:

Lots of advanced language features that they may have already heard about.
We’ll show them the best ways to use those features.
Chapter 4

Tuning for Performance

Notes:

<B,1: Tuning for Performance</B>
Chapter Objectives

In this chapter, you will learn how to

- Measure different performance aspects of a Java application
  - Identify bottlenecks
  - Tune performance
  - Reduce memory usage
- Avoid coding habits that can lead to poor performance
- Take advantage of operating system optimizations

Notes:

A long hard slog. Try to get a weave a thread that weaves all the elements of this chapter together. There are lots of concepts here, so you don’t want to get too side-tracked.

This is mine:

We’ll look at what to measure – there’ll be three things we’ll be most interested in. We’ll then look at how to measure them. We’ll use three different tools for those. Then, we’ll look at how to improve performance.

-- start with the easy stuff – JVM settings and the like
-- get your memory usage down as much as you can
-- make sure while coding that you are not employing any bad habits
-- as a last resort, see if the native OS can bail us out
Tuning for Performance

Overview

Measuring Performance
Improving Response Time
Hands-On Exercise 4.1
Reducing Memory Footprint
Hands-On Exercise 4.2
Leveraging the Operating System

Notes:

<Tuning for performance>
First step is to measure performance
Know When and What to Optimize

- Don’t optimize prematurely
  - The goal is to create software that works
  - Optimization should be done only after the application has been tested
  - Optimized code can be hard to maintain
  - Optimization takes time away from development
  - You may end up optimizing code that isn’t a “bottleneck”

- Don’t try to optimize everything
  - Consider which resource you want to optimize
    - Time of execution (the most common)
    - Network, memory, database, disk I/O
  - Tackle the parts of the code that will give you the greatest payoff
    - Methods that use up lots of resources
    - Methods that are called very often
  - These are called the bottlenecks of the application

Notes:

<ipf>R,4: Know When and What to Optimize</ipf>

Emphasize this.

Lots of optimizations are bad code; you don’t want to have too much of it.
Performance Improvement Strategy

1. Measure the performance of the application
   — If the application meets requirements, you are done
     ✮ Measure use case by use case
     — Measure CPU time taken method by method (called profiling)

2. Identify the bottlenecks
   ✮ Profiling tools can identify time hogs
   — Identifying other resource hogs requires understanding the application
   — If possible, create a test to bypass the bottleneck
     – See if the performance improves
     – You have correctly identified the bottleneck

3. Alter the application to improve the performance of the bottleneck
   ✮ Regression test to ensure application behavior hasn’t changed

4. Measure performance again
   — Decide whether to further optimize
   — May need to focus on different bottleneck

Notes:

<i>ipf>L.5: Performance Improvement Strategy</i>
Techniques to Improve Performance

- These are ways you can alter the application to improve performance
  - Change JVM settings to better fit your application
  - Change the algorithm used
  - Change the data structure used
    - Arrays and collections
  - Reuse objects
    - Spurious object creation can be expensive
  - Cache information locally
    - Instead of refetching or recomputing
    - Take advantage of operating system optimizations
- We’ll look at these techniques in the next chapter
  - Parallelize your operations (use threads)
  - Appear quicker by providing status to user

Notes:

<ipf>R,6: Techniques to Improve Performance</ipf>

The last two we’ll look at in the next chapter.
All the others we’ll look at now.
Tuning for Performance

Overview

Measuring Performance
Improving Response Time
Hands-On Exercise 4.1
Reducing Memory Footprint
Hands-On Exercise 4.2
Leveraging the Operating System

Notes:

<ipf>L,7: Tuning for performance</ipf>
First step is to measure performance
What to Measure

Measure different aspects of performance differently

- Response time, using a client-side timer (JMeter)
- CPU usage, using a profiler (JIP)
- Memory usage, using garbage collector information (verbosegc/JMX)
- May also need to measure network and database performance

Performance is a loose term. Find out what aspect your users are interested in.

Notes:

<ipf>R,8: What to Measure</ipf>

Three different performance aspects with three ways of measuring them. We’ll do all three. Network and database performance may be needed to eliminate the Java code as the source of the bottleneck!
How to Measure

Carry out timing tests from a different machine
   — The timing application itself consumes resources
   — Minimize variability caused by network traffic by using an isolated LAN

Notes:
<ipf>R,9: How to Measure</ipf>
Three different performance aspects with three ways of measuring them. We’ll do all three.
Network and database performance may be needed to eliminate the Java code as the source of the bottleneck!
Where to Measure

- If you are building a servlet, you already have servlet unit tests
  - These are functionality tests
  - You could time these, but …

**Don't measure performance using HttpUnit on a developer workstation**
- The functionality test runs outside the container
- The container does many optimizations
- You may end up tuning code that doesn't need to be tuned

- **Measure performance in the deployed environment**
  - In the JVM in which it will ultimately run
  - The container configuration files will allow you to set JVM parameters to launch the measurement tools
    - We'll use Tomcat as an example of how to do this
    - Your favorite container will have something similar

Notes:

<ipf>L,10: Where to Measure</ipf>
Test performance inside the container
Timing Measurements

- Measure three separate things for each functional test
  - Response time
    - Time taken to get a response from the application on the use case
    - Assuming only one user
  - Network overhead (latency)
    - Time to get any response from the application regardless of use case
  - Throughput (hits per second)
    - Number of simultaneous users who can run the use case at a time

Notes:

<ipf>R,11: Timing Measurements</ipf>
JMeter

- **JMeter is an open-source Java performance measurement tool**
  - Designed for testing performance of use cases
    - Timing tests
    - Concurrent users

- **JMeter provides reports in several formats**
  - Graphs
  - CSV files (can be exported to Excel or other spreadsheet program)
  - XML files
    - Can apply style sheets and view in browser
    - Or generate custom reports

- **Usually used from graphical user interface (GUI)**

- **JMeter can also be launched from an Ant build file**
  - Continuous integration of performance tests

http://jakarta.apache.org/jmeter/

CSV = comma-separated value

Notes:

<ipf>L,12: JMeter</ipf>
JMeter Terminology

- **JMeter has some interesting terminology**
  - Test plan
    - Set of steps that JMeter should execute
    - Such as the list of Web sites that JMeter should visit
    - Useful if you have a login page, for example
    - Can have logic to choose among various steps
    - Can place regular expression assertions on the returned values
  - Thread group
    - Number of threads represents the number of concurrent users
    - Specify how long to take to “ramp” up to total number of users
    - Specify how many times user will make query
    - Can schedule the test to run at a specific time
  - Sampler
    - HttpRequest sampler will send HTTP requests to a Web site
    - Also FTP, JDBC, SOAP, etc.
  - Listener
    - Graphical, data file, etc.

*Notes:*

R,13: JMeter Terminology

Keep this slide up as you go through the following slides
An Example Load Test

Why many users is this test plan simulating?
How often will those users execute the steps?

Notes:
<ipf>L,14: An Example Load Test</ipf>
An Example Test Plan

What are the steps that a user takes?
Where does the output go?

Notes:

<ipf>R,15: An Example Test Plan</ipf>
Interpreting the Results

- Median value is the response time experienced by a typical user
  - Equal numbers of users experience more and less than median
  - Average affected by atypical events
  - Use 90-percent line for confidence intervals

Notes:

<ipf>L,16: Interpreting the Results</ipf>
What Do You Do Now?

- You've measured the application performance
  - Now what?
    - The answer depends on what the problem was
- Throughput?
  - Multithread the application
    - If you are using significant I/O and CPU
    - Or if you have multi-CPU servers
    - Does not help Web applications, though
  - Buy more hardware
  - Go asynchronous if possible
- Latency
  - Measured by difference in response time between isolated LAN and normal network
  - Upgrade network
  - Do more things locally

Notes:


Spend time discussing the first two; we won’t look at them anymore in this chapter.
Notes:

Tuning for Performance

First step is to measure performance
Addressing Response Time Problems

- If the problem is response time, you need to speed up the application

- Which parts of your application should you speed up?
  - The bottlenecks
    - Concentrate your efforts on methods that take more than 10 percent of the time
    - Methods that take less than 5 percent or so are poor targets for optimization

If all your methods take less than 5 percent, consider radical changes in the way your application works, because simple tuning is not going to help much.
For example, consider creating batch reports rather than dynamic pages.

Notes:

<ipf>L,19: Addressing Response Time Problems</ipf>
5%, 10% are just my rules of thumb
Profilers

- **Find the bottlenecks using a profiler**
  - Measure the amount of time spent in every method
  - Rank the methods in terms of the amount of time

- **Many free and commercial profilers exist**
  - JProbe, Optimizelt, etc.
  - The JVM ships with its own profiler

- **Java 5 has built-in profiling mechanisms**
  - Avoid dependence on OS capabilities

---

**Notes:**

<i>ipf>L,20: Profilers</i>
Java Interactive Profiler

- The Java Interactive Profiler (JIP) makes use of Java 5 hooks
  - Hooks into the system class loader to profile the application
  - Does not depend on native OS timing applications
  - Does not slow down your application significantly
  - Can be switched on or off while your program is running
  - Works only with Java 1.5

- For earlier versions of Java, use hprof (ships with the JDK)
  - Can cause significant slowdown of your application
  - Cannot be switched on or off

- Specify JIP as JVM argument to Tomcat (or your application server)
  - For Tomcat, set the JAVA_OPTS environment variable
    - -javaagent:profile.jar -Dprofile.properties=filename
  - The profile.properties specifies options for JIP

Notes:

There’s a white paper on JIP. Read it. It’s quite cool.
Profiling In-Container With JIP

- Exercise your test case a few times with your browser
  - To get the startup costs out of the way

- Perform the profiling test
  - Start profiling
  - Execute your test case
  - Finish profiling
  - In our class, Ant tasks exist to perform the start and finish of profiling

- Output summarizes most expensive methods
  - Count: number of times method was called
  - Time: total time the method took
  - Percent: percent of total response time taken by method

- Typical to concentrate on the top 10 or so methods
  - Reduce number of calls or reduce time taken by method

Notes:

<ipf>L,22: Profiling In-Container With JIP</ipf>
### Writing Fast Code Routinely

- **Get into the habit of writing good, fast code**
  - Best practices address common Java bottlenecks that you need to avoid

- **Design to interfaces so that you can avoid casts**
  - Overridden methods are faster than casts

```java
Promotion p = ...;
double discount = ((FixedDiscountPromotion) p).getDiscountAmount();
purchase.applyFixedDiscount(discount);
```

```java
Promotion p = ...;
Discount discount = p.getDiscount();
purchase.applyDiscount(discount);
```

### Notes:

<ipf>R,23: Writing Fast Code Routinely</ipf>
Write Fast Loops

How can this loop be made faster?

```java
for (int i = 0; i < arr.size(); ++i){
    Purchase p = (Purchase) arr.get(i);
    double discount = getTodaysDiscount( new Date() );
    p.setSalePrice(p.getSalePrice() * (1 - discount));
}
```

Notes:

<i>L,24: Write Fast Loops</i>
Write Fast Loops (continued)

Move all one-time calculations out of the loop

```java
int len = arr.size(); // compiler would probably do this anyway
double discount = getTodaysDiscount( new Date() );

for (int i = 0; i < len; ++i){
    Purchase p = (Purchase) arr.get(i);
    p.setSalePrice(p.getSalePrice() * (1-discount));
}
```

Notes:

<ipf>R,25: Write Fast Loops (continued)</ipf>

It might also get rid of a subtle bug – it’s unlikely that the intent is to change the discount in the middle of array

Simply because the date crossed over.
Built-in Arrays vs. Collection Classes

🌟 You will often get much better performance by using a Purchase[] instead of an ArrayList
  — arr[3] can be many times faster than arr.get(3)
  — arr[3] does not need to be downcast into a Purchase

🌟 Do this only if the loop turns out to be a bottleneck
  — And if you will know the size of the array beforehand
  🌟 Or, create ArrayList, and once the full list is created, convert to array
    — Much numerical software does this

Notes:

<ipf>L,26: Built-in Arrays vs. Collection Classes</ipf>
This is demonstrated in a sample code package. Feel free to demo this if you’d like
Analyze and Reorder Loops

Reorder loops so that expensive operations are done fewer times
— For example, if code needs to notify all monitoring services about purchases where a credit transaction was denied, it:
  — Could loop through monitoring services first, and then through purchases
  — Or loop through purchases first, and monitoring services second

```java
for (int j=0; j < monitoringServices.length; ++j){
  for (int i = 0; i < purchaseList.length; ++i){
    if (!creditChecker.isPurchaseValid(purchaseList[i])) {
      notifyFraudAlert(monitoringServices[j]);
    }
  }
}
```

```java
for (int i = 0; i < purchaseList.length; ++i){
  if (!creditChecker.isPurchaseValid(purchaseList[i])) {
    for (int j=0; j < monitoringServices.length; ++j){
      notifyFraudAlert(monitoringServices[j]);
    }
  }
}
```

Notes:
<ipf>R,27: Analyze and Reorder Loops</ipf>
Spend time on this. This is the optimization they’ll do in Ex 4.1.
It’s not that obvious.
Use String Wisely

- Any literal String (e.g., "hello") is very efficient
  — Because it is interned by the compiler
- The substring operation is also very efficient
- Other methods (replace, toUpperCase, etc.) are very expensive
  — Need to create a new String
  — Need to account for internationalization
  — Consider whether you can avoid uppercasing
- Never use replaceAll(regex) if you will replace in more than one string
  — Do Pattern.compile(regex) and use the matcher to replace

Notes:

<ipf>L,28: Use String Wisely</ipf>

Substring is efficient because it refers to the original char array, but with a different start and length.
One way to avoid trimming – if the string is going to land up in an html page, no need to trim.
One way to avoid uppercasing – if the string came from a database, perhaps uppercase on insert.
Similarly, upper case when you get from user.
Then, all future comparisons and sorts can happen with plain equals()
String VS. StringBuilder

🏆 Concatenate literal Strings using the + operator
— The compiler will automatically create one large String

```java
String sonnet =
    "Shall I compare thee to a summer's day?\n" +
    "Thou art more lovely and more temperate:\n" +
    "Rough winds do shake the darling buds of May,\n";
```

🏆 In runtime, use a StringBuilder
— Avoids the creation of too many temporary objects
— Many compilers now do this automatically

```java
StringBuilder sonnetBuf = new StringBuilder();
while ( (line = reader.readLine()) != null ){
    sonnetBuf.append(line);
}
String sonnet = sonnetBuf.toString();
```

🏆 Same principle applies for any immutable class with a mutable counterpart

Notes:

<i>R,29: String vs. StringBuilder</i>

Compile time vs run-time
Be aware that sonnetBuf.toString() doesn’t create a separate copy of the string
Until the stringbuffer is subsequently modified.

StringBuilder is to StringBuffer what ArrayList is to Vector
i.e. faster when thread safety is not required.
Minimize Logging Overhead

- **Log only what you need to know**
  - Logging may involve synchronized disk I/O and can be pretty slow
  - Use a logger that can be switched off at runtime—`java.util.logging`, `log4j`

- **Surround logging with a code guard**

- **Log checked exceptions only once**
  - Multiple log messages may also confuse user
  - Either log a message or throw a wrapped exception—don’t do both

```java
catch(FileNotFoundException e){
    if ( log.isInfoEnabled() ){ // code guard
        log.info(userConfig + " not found. Trying " +
                  systemConfig);
    }
    // no rethrow
    return init(systemConfig);
}
catch(SAXException saxException){
    // no log
    throw new RainForestDataException(saxException);
}
```

**Notes:**

<ipf>L,30: Minimize Logging Overhead</ipf>

The subtle problem is what happens if the wrapped exception’s class cannot be found on other end
Input/Output (I/O)

- Execute I/O in separate thread
  - I/O can often be performed in a background thread
  - Gives user a perception of being lively
- BufferedReader is much faster than just FileReader
  - Much, much faster
- Can trade off networking with CPU by reading/writing compressed data

Notes:

<ipf>L.31: Input/Output (I/O)</ipf>
Choose Data Structure

Use an appropriate collection class
- ArrayList is usually the fastest collection
- LinkedList is faster if you need to insert or remove in the middle
- HashMap is the fastest associative collection
  - IdentityHashMap is even faster, if it will work for your application
- Vector/HashTable are slightly faster than synchronized ArrayList/HashMap

May be able to redesign so that HashMap can be replaced by ArrayList
- If you can map the key String to a sparse index or a unique number

Set the initial capacity higher if you know you'll need lots of items
- Constructor parameter for ArrayList

Notes:

<ipf>R,32: Choose Data Structure</ipf>

If you have a map that goes from SocialSecurityNumber to Person,
if you can redesign software to use EmployeeNumber internally
and EmployeeNumber goes from 0 to 1000, then instead of using
HashMap, you get to use ArrayLists (except in the initial database call)
IdentityHashMap compares based on == instead of on equals()

Synchronized ArrayList, HashMap are proxies to an underlying collection object which provide
synchronized versions of the methods, there is a minor overhead associated with a method call. In an
experiment adding 1000000 objects to a Vector and a synchronized ArrayList, the Vector was in the
region of 10-20% faster. Without synchronization applied to the array list, the same test shows that the
array list is ten times faster than a vector. Note that these differences are only significant when adding
large numbers of objects.
Object Reuse With a Factory

Use the factory pattern instead of a constructor

```java
PurchaseVerifier p = Factory.newPurchaseVerifier();
```

The factory can use a pool to avoid object creation

- **Sometimes, copying an object is more efficient than creating a new one**
  - The factory could copy fields from the master rather than recomputing them
  - Do this when constructor makes expensive calls
  - This is the GoF *Prototype Factory* design pattern

One of the best ways to improve Java application performance is to avoid garbage collection altogether. You can do this by reusing objects whenever possible.

Notes:
<ipf>L,33: Object Reuse With a Factory</ipf>

clone() avoids calling chained constructors and is therefore faster if the constructors do significant computations or DB access.

ThreadLocal in next chapter
But clone() has other problems re. Ch. 3
Canonical Objects

• If you have a limited number of read-only objects
  ✨ You can improve performance by pre-creating them

```java
public class PurchaseVerifier {
    public static final CreditChecker VISA = new CreditChecker(...);
    public static final CreditChecker AMEX = new CreditChecker(...);
    public static final CreditChecker DINERS = new CreditChecker(...);
    // ...
}

— Use VISA or DINERS wherever needed instead of recreating each time
— You can also improve the speed of comparisons by using ==

```java
if ( purchase.creditCheckedBy == PurchaseVerifier.VISA ){
    // get 3-digit security code
}
```

😍 This is also called the Java enum pattern

⚠️ Do not allow canonical objects to be mutable

Notes:

<ipf>R,34: Canonical Objects</ipf>

Be prepared for a question on startup costs. Because Java classes are loaded (and static final fields created) as needed, all the CreditCheckers will be created at once even if a particular application uses only VISA,
but this is not application startup, only the first time that PurchaseVerifier is used.

To avoid creating all the CreditCheckers at once, the solution is to make each of the CreditCheckers a Singleton, but that’s probably overkill.
Tuning for Performance

Overview
Measuring Performance
Improving Response Time
Hands-On Exercise 4.1
Reducing Memory Footprint
Hands-On Exercise 4.2
Leveraging the Operating System

Notes:
<i>ipf>L,35: Tuning for Performance</i>
First step is to measure performance
Hands-On Exercise 4.1: Measuring Performance

In this exercise, you will apply the response time measurement and optimization techniques you've learned to our case study application

- Measure the response time of the use case using JMeter
- Profile the method usage with JIP
- Identify one of the bottlenecks
- Investigate a fix to the bottleneck
- Quantify the improvement in response time

Please refer to the Exercise Manual

Notes:

Tell them that these are the major steps of any performance tuning strategy.

Exercise comments:

If they miss out using ant to rebuild the project after changing the code, the config files are not copied correctly and the unit tests will fail.
Tuning for Performance

Overview
Measuring Performance
Improving Response Time
Hands-On Exercise 4.1
▶ Reducing Memory Footprint
Hands-On Exercise 4.2
Leveraging the Operating System

Notes:
<Tuning for Performance>
First step is to measure performance
The Java Virtual Machine

- Determine the type of JVM you are using
  - Just-in-time compiler
    - Compiles Java code into native code as it executes
    - Does not optimize the Java code
      - So source code optimizations help a great deal
    - Method calls are cheap
    - Object creation is expensive
  - Hot-spot compiler
    - Optimizes the heavily used areas (bottlenecks) of your application
    - Source code optimizations may not help much
    - Try using `-Xcomp` to optimize all the code, not just bottlenecks
      - Very expensive on startup, but gets much faster

Notes:

<ipf>L.38: The Java Virtual Machine</ipf>
Project-Wide Optimizations With a JIT Compiler

⚠️ A just-in-time compiler does not do much Java optimization
   — Therefore, it is a good idea to get into good habits when writing code
   — Don’t rely on the compiler to optimize your code

• Reduce the size of your code base
   — Remove unused methods and classes
   — Avoid checking unnecessarily for null
   — Remove unused fields from objects
     – Definitely don’t read them from the database!

Notes:

<ipf>R,39: Project-Wide Optimizations With a JIT Compiler</ipf>
Measuring Memory Usage

- One of the major costs of any application is its memory usage
  - Check the memory usage and set JVM options appropriately
  - Ensure that garbage collection is being efficient
  - Ensure that your program is not growing in memory (*memory leak*)

- To measure memory usage
  - Start the JVM with the option `verbosegc`
  - Prints out Garbage Collection (GC) statistics
  - Write a script to parse the GC output
    - The GC output format depends on the JVM version
    - So, you need to create a custom one

- Alternately, use Java Management EXtensions (JMX)
  - Start application with `-Dcom.sun.management.jmxremote`
  - View memory usage using JConsole
    - Part of Java 5 SDK

*Notes:*

<ipf>R.40: Measuring Memory Usage</ipf>
verbosegc Output

- The verbosegc output may look like this:

```java
[Full GC 1613K->1557K(2240K), 0.0958451 secs]
[GC 2069K->2068K(3176K), 0.0076571 secs]
[GC 2580K->2578K(3176K), 0.0085226 secs]
[GC 3090K->3089K(3688K), 0.0085991 secs]
[Full GC 3089K->3089K(3688K), 0.1024315 secs]
[GC 3601K->3600K(5728K), 0.0070861 secs]
[GC 4112K->4110K(5728K), 0.0085919 secs]
[GC 4622K->4621K(5728K), 0.0085399 secs]
[GC 5133K->5132K(5728K), 0.0077700 secs]
[GC 5644K->5643K(6240K), 0.0085399 secs]
```

- This application is growing in memory and spending lots of time in GC
  — This may be OK in terms of what the application is doing
  — But we may need to set starting heap size appropriately

- GC/FullGC refers to generation

If you see the total heap size increasing, it’s usually bad news.

Notes:

<i>verbosegc Output</i>
Garbage Collection JVM Options

- Tuning the JVM can result in better performance
  - Choose starting and maximum heap sizes
  - Experiment with incremental and concurrent garbage collectors

- Try changing these settings
  - Maximum heap size (-Xmx)
    - Set larger than maximum size of application (reported by verbosegc)
    - Garbage collection runs less often, but takes longer
  - Starting heap size (-Xms)
    - Try a number between 50 percent and 100 percent of maximum heap size
    - Set to be the minimum size of application (reported by verbosegc)
  - Turn off garbage collection
    - If your application is very short-lived
  - Non-standard flags
    - Turn off garbage collection of classes (-Xnoclassgc)
    - Incremental garbage collection (-Xincgc)
    - Concurrent garbage collection (-Xconcgc)
    - Fewer pauses but longer garbage collection

Notes:

<ipf>L.42: Garbage Collection JVM Options</ipf>

Only the first two are likely to help, in my experience and only if the heap size was ridiculously mismatched to start with.

We list all these because some one in your class may have heard something about incgc etc.
Generational Garbage Collectors

- **Most modern garbage collectors are generational**
  - Assumes newly allocated objects are also those most likely to need GC

- **A minor garbage collector runs often**
  - On newly created objects that are no longer referenced
    - Example: local variables that fall out of scope
  - New objects still referenced get promoted
  - Very cheap

- **A major garbage collector runs less frequently**
  - When the machine is running low on resources
  - Does garbage collection on “long-lived” objects
  - Does copying or “mark-and-sweep”
  - Takes up lots of time to run
    - Can often cause pauses in your application

---

**Notes:**

<ipf>R,43: Generational Garbage Collectors</ipf>

The colors in the figure refer to the tri-color algorithm, if you want to go there.

Copying, rather than mark-and-sweep, seems to be more commonly used because it results in less fragmentation.
Working With a Generational Garbage Collector

- Optimization strategies with generational garbage collectors
  - Ensure temporary objects don’t get promoted
    - Put local variables in the most local scope possible
    - Assign variables you no longer need to “null”

- Don’t use finalizers in your classes
  - A class with a finalizer is automatically pushed to the full garbage collection
  - It is not collected by the minor collector

- Increase size of young space
  - -XX:NewSize (JVM-dependent)

- With any garbage collector
  - Set the maximum and starting heap sizes appropriately
    - Ensure that the time spent in GC is no more than 10 percent of total time
    - Otherwise, look into object reuse strategies

Notes:

<ipf>L.44: Working With a Generational Garbage Collector</ipf>
Analyzing verbosegc on Generational GC

• Consider the memory usage at each stage

• The full GC should not be releasing too much memory
  — If it is, then young objects have been inappropriately promoted
    ☀ Try increasing the heap size for young objects
    ☀ Look at code to see if objects are being kept around inappropriately

• Try to keep full GC under 10 percent of application time
  ⚠ If you increase the heap size, GC time will increase

Notes:

<ipf>R,45: Analyzing verbosegc on Generational GC</ipf>

In my experience, most applications take 5-10% of time in full GC (vl)
Reducing Memory Usage

- One of the major costs of any application is its memory usage
  - Programs with larger memory are slower
    - The OS may use fragmented memory segments that need to be tied together somehow
    - The OS may be swapping memory to disk
    - Garbage collection has to go through more space
  - Reducing an application’s memory usage pays off at multiple levels

- Take advantage of generational garbage collectors
  - By releasing memory as quickly as possible
  - Try to declare variables in the most local scope
  - Set references you no longer need to “null”

Notes:

<ipf>R,46: Reducing Memory Usage</ipf>
Use Reference Types to Aid the Garbage Collector

- Objects are often cached in a collection for later retrieval
  - Applications add entries to the cache
  - But do not specify when they no longer need the cache entry
  - If they are not garbage-collected, application exhibits a memory leak

💡 Provide hints to the garbage collector
  - About cached objects that can always be recreated if they are garbage-collected
  - The garbage collector will remove them if it has no other option

- Hints to the GC are provided using reference types

Notes:

<ipf>L.47: Use Reference Types to Aid the Garbage Collector</ipf>
Types of References

- Different reference types are garbage-collected differently
  - WeakReferences in the first round
  - SoftReference if there are no WeakReferences and the JVM is still short on memory
  - Decide whether something should be Weak or Soft depending on how expensive it is to recreate

- A WeakHashMap makes reference types of its keys automatically

- Explicit use of a reference type

```java
Reference<PurchaseVerifier> ref = new WeakReference<PurchaseVerifier>(new PurchaseVerifier(…));

// later
PurchaseVerifier p = ref.get();
if (p == null){
    p = new PurchaseVerifier(…); // recreate if GC’ed
}
```

Notes:

<ipf>R,48: Types of References</ipf>

The third type of reference is a Phantom reference, this is from the Javadoc on this class:

Phantom reference objects, which are enqueued after the collector determines that their referents may otherwise be reclaimed. Phantom references are most often used for scheduling pre-mortem cleanup actions in a more flexible way than is possible with the Java finalization mechanism.
Online Processing of Data

One way to avoid creating large arrays is to process data **online**
- Don’t create an array of the data
- Make the DAO retrieve data only when it is needed by the application

```
public class Iterator implements java.util.Iterator<Purchase> {
    public boolean hasNext() {
        try {
            boolean hasMore = rs.next(); // move to next row
            if (!hasMore) { dispose(); /* close conn */
                return hasMore;
            }
        } catch (SQLException e) { log.error(e); dispose(); return false; }
        return hasMore;
    }
    public Purchase next() {
        try {
            return new Purchase(rs.getInt(1), rs.getString(2));
        } catch (SQLException e) {
            throw new RainForestDataException(e);
        }
        // remove() should throw UnsupportedOperationException
    }
}
```

**Notes:**

<ipf>L.49: Online Processing of Data</ipf>

Ideally, it does not implement java.util.Iterator because any thing that implements java.util.Iterator should support simultaneous
Traversal of collections by multiple iterators. This one doesn’t, but that’s just too hard to explain. Few people will catch it.
Use Immutable Objects

• It can be tempting to avoid creating new `Purchase` objects by reusing them

```java
private Purchase myPurchase;
public Purchase next(){
    myPurchase.reinit(rs.getInt(0),rs.getString(1),rs.getInt(2) );
    return myPurchase;
}
```

• However, a client may choose to keep a `Purchase` item for future reference

```java
Purchase p = iterator.next();
if ( p.getSalePrice() > 35.0 )
    expensivePurchases.add( p );
```

⚠️ What effect would the `reinit()` have on this client?

🌟 The `Purchase` objects should be immutable to avoid this problem
— Even though using `reinit()` will improve performance
— Could specify that clients need to `clone()` the object before keeping it

Notes:

<ipf>R,50: Use Immutable Objects</ipf>
Tuning for Performance

Overview
Measuring Performance
Improving Response Time
Hands-On Exercise 4.1
Reducing Memory Footprint
Hands-On Exercise 4.2
Leveraging the Operating System

Notes:
L,51: Tuning for Performance
First step is to measure performance
Hands-On Exercise 4.2:
Online Processing

• In this exercise, you will change `PurchaseDAO.Purchases.Iterator` to
  get the data as needed
  — `PurchaseDAO` shouldn’t create large arrays
  — Monitor memory usage by application
    – Has total memory usage decreased?
    – Has the pattern of memory use changed to suit generational garbage
      collection better?

Under what circumstances is this a worthwhile optimization?

Please refer to the Exercise Manual

Notes:

<ipf>R,52: Hands-On Exercise 4.2: Online Processing</ipf>
Tuning for Performance

Overview

Measuring Performance

Improving Response Time

Hands-On Exercise 4.1

Reducing Memory Footprint

Hands-On Exercise 4.2

Leveraging the Operating System

Notes:

<i>Tuning for Performance</i>

First step is to measure performance
Java NIO

- **The Java New Input Output APIs (Java NIO)**
  - Since Java 1.4
  - Bring some scalable I/O operations to Java

- **NIO brings performance improvements in specific situations**
  - Operating on data from large (gigabyte-size) files
  - Read from multiple sockets simultaneously
  - Operate at the binary data level
    - Such as to transfer files, but not to parse them
    - Or to read into an array of primitives

- **NIO is meant to supplement regular I/O**
  - Most Java programs should still use `java.io` classes
  - The `java.io` classes have been modified to support NIO operations

---

**Notes:**

<ipf>R.54: Java NIO</ipf>
What Is JNI?

- The Java Native Interface (JNI) allows you to invoke C code from Java
  - JNI rarely improves performance
  - Lots of high-performance visualization software now written in Java

Notes:

<i>What Is JNI?</i>

XXX: break into two slides
When Is JNI Useful?

- **Use JNI to**
  - Invoke legacy code in C, C++, or Fortran
  - Call to native Operating System (OS) functionality

- **For example, on Linux, there is a mechanism called `inotify`**
  - Can select and wait on a directory or file
    - Just as NIO provides ability to select and wait on a socket
    - Notified whenever the file or directory is written to or read from
      - No need to poll the directory, list its files, and look for changes!
  - Can drastically improve efficiency
  - `inotify` is a UNIX device; it has no counterpart on Windows
    - Useful when you know your server environment is Linux
  - You need to access it via a C interface

- **Disadvantages of JNI**
  - Not portable
  - Outside control of Java security manager

---

**Notes:**

<ipf>L.56: When Is JNI Useful?</ipf>
Example JNI and NIO Application

- Open the Eclipse project `dm41_niojni`
  - This illustrates several operations using NIO
    - Counting the number of words in a file
    - Replacing words in a file and writing out a new file
  - The word count operation is also illustrated using JNI

- The Ant “test” task will run a set of timing tests
  - On Shakespeare’s *The Merchant of Venice*
    - The file is only 130 KB
    - Does NIO seem to help?
    - How about JNI?
    - What does this tell you about the speed of the JDK core classes?

Notes:

<ipf>L,57: Example JNI and NIO Application</ipf>

NIO is good for pure, binary transfers of data. As soon as you need to “peek” inside the data to do something,
Java I/O is just as fast.
Review

1. When should you optimize?

2. What should you optimize?

3. Why should you place code guards around logging statements?

4. When are weak references needed?

Notes:

<ipf>R,58: Review</ipf>
Best Practices: Tuning for Performance (Strategy)

- Follow a strategy in performance tuning
  - Measure performance use case by use case
  - Regression test to ensure application behavior hasn’t changed
  - Measure response time, CPU usage, and memory usage against targets
  - Measure performance in deployed environment from different machine after application has had time to get over initialization
  - Diagnose problem: throughput, latency, or response time
  - Tackle only parts of an application that offer a reasonable payoff

- Understand your environment and its limitations
  - Take advantage of the compiler-type and garbage-collector characteristics
  - Set heap size appropriately
  - Process data online to reduce memory footprint
  - Don’t use finalizers
  - Release memory as quickly as possible
  - Use reference types to aid the garbage collector

Notes:

<ipf>R,59: Best Practices: Tuning for Performance (Strategy)</ipf>
Best Practices: Tuning for Performance (Ideas)

• **Best practices for faster, lighter code**
  — Design to interfaces to avoid downcasts
  — Use the `String +` operator for compile-time concatenation
  — Use `StringBuilder` for runtime concatenation
  — Surround logging with a code guard
  — Use the factory pattern instead of a constructor (and reuse objects)
  — Prefer `BufferedReader` to bare `FileReader`
  — Perform I/O in separate thread
  — Canonical objects (Java enum) also optimize frequent equality checks
  — Declare variables in most local scope

• **Optimize loops and collections**
  — Move all one-time calculations out of loop
  — Use built-in arrays rather than collections when size is known
  — Reorder loops so that expensive operations are done fewer times
  — `ArrayList` and `HashMap` are usually the fastest collections

Notes:

<ipf>R,60: Chapter 4: Tuning for Performance</ipf>
Chapter Summary

In this chapter, you have learned how to

- Measure different performance aspects of a Java application
  - Identify bottlenecks
  - Tune performance
  - Reduce memory usage

- Avoid coding habits that can lead to poor performance

- Take advantage of operating system optimizations

Notes:

<ipf>R,61: Chapter Summary</ipf>
Chapter 5

Taking Full Advantage of Threads

Notes:

<B,1: Taking Full Advantage of Threads</B>
Chapter Objectives

In this chapter, you will learn how to
- Improve response time by parallelization
- Bulletproof a threaded application
- Share data between threads
- Manage the implications of synchronization
- Increase user impression of responsiveness using threads

Notes:
<ipf>L,2: Chapter Objectives</ipf>
Taking Full Advantage of Threads

Parallel Execution Using Threads

Bulletproofing a Threaded Application

Managing Implications of Synchronization

Hands-On Exercise 5.1

Notes:
<ipf>R,3: Taking Full Advantage of Threads</ipf>
There may be some people who don’t realize that this is relevant. Tell them servlets are automatically Threaded and you will get their attention.
What Is Threading?

- An application may be made to execute multiple tasks simultaneously
  - Multiple threads of execution
  - Similar to how an operating system runs several applications simultaneously
  - A sequential program executes tasks one by one
  - The scheduler allows a threaded program to execute tasks in parallel

**Notes:**

<i>What Is Threading?</i>

4 threads running in parallel. The first thread is waiting on user input
Why Thread?

- Threading can
  - Provide parallelization
    - On machines with multiple CPUs or hyperthreading
    - With one CPU if different threads use different system resources
  - Make an application seem responsive
    - Because one thread can be assigned to just receive user input
    - And other threads can do actual processing

Notes:

<ipf>L.5: Why Thread?</ipf>

4 threads running in parallel. The first thread is waiting on user input
Programming With Threads

• Steps:
  1. Write a class that implements Runnable
  2. Create a Thread object, passing in the Runnable
  3. Call start() on Thread

• Consider, for example, this scenario:
  — Customer approaches Clerk at a counter
  — Requests to operate Account

Notes:

<i>Programming With Threads</i>

Example illustrates a banking scenario

  Customer approaches Clerk at a counter and requests to operate Account

Explain banking scenario a little here

XXX: break into two slides
Programming With Threads: An Example

```java
public class ThreadedBanking implements Runnable {
    private Clerk clerk;
    private Customer customer;
    public ThreadedBanking(Clerk clerk, Customer customer) {
        this.clerk = clerk;
        this.customer = customer;
    }
    public void run() {
        clerk.performBanking(customer);
    }
}
```

Thread task1 = new Thread(new ThreadedBanking(clerk1, cust1));
Thread task2 = new Thread(new ThreadedBanking(clerk2, cust2));
task1.start();
task2.start();
task1.join();
task2.join();

Notes:

<ipf>R,7: Programming With Threads: An Example</ipf>

Example illustrates a banking scenario

Customer approaches Clerk at a counter and requests to operate Account

Explain banking scenario a little here

XXX: break into two slides
Threaded Banking Demo

- Open the Eclipse project dm51_threads
  - Execute the run_sequential Ant target
    - How long does this take? ______________________
  - Execute the run_parallel Ant target
    - How long does this take? ______________________

- Examine the classes in the application
  - We will build on this example further
  - Account has withdraw(), deposit() that involve some network traffic
    - Simulated here by a random delay
  - Customer has an Account
  - Clerk can operate an Account on behalf of a Customer
  - ThreadedBanking and Main
    - These were on the previous slides

Notes:

<ipf>L,8: Threaded Banking Demo</ipf>
Keep the instructor demo machine going on one projector. You’ll do another demo in a few minutes.
When Does the Scheduler Switch?

- Can set priority for threads
  - Higher-priority threads are scheduled more often

- The scheduler will switch away from a thread if the thread
  - Calls Thread.sleep(...)
  - Calls Thread.yield()
  - Is blocked waiting for input
  - Is passed over because a higher-priority thread wants the CPU

- Cannot predict when the switching happens
  - Don’t rely on particular scheduling mechanism
    - Threads are not notified when they are suspended or resumed

Notes:

<ipf>R,9: When Does the Scheduler Switch?</ipf>
Taking Full Advantage of Threads

Parallel Execution Using Threads

Bulletproofing a Threaded Application

Managing Implications of Synchronization

Hands-On Exercise 5.1

Notes:

There may be some people who don’t realize that this is relevant. Tell them servlets are automatically Threaded and you will get their attention.
Race Condition

- Threaded programming is easy, right?
  — Not so fast!

- Suppose we have two Customers who have a joint Account
  — And both of them visit the bank at the same time

```java
Thread task1 = new Thread( new ThreadedBanking(clerk1, cust1) );
Thread task2 = new Thread( new ThreadedBanking(clerk2, cust2) );
task1.start();
task2.start();
task1.join();
task2.join();
```

Notes:

<ipf>R,11: Race Condition</ipf>
Race Condition: Demo

- Run the Ant target `run_sharedacct in dm51_threads`
  - Are the results reliable?
  - What happened?
- This is called a `race condition`

Notes:

<ipf>R,12: Race Condition: Demo</ipf>
Keep the instructor demo machine going on one projector. You’ll do another demo in a few minutes.

Use the run-sequential target to illustrate that the bank account balance should be 120
Then run the run-parallel target to show that the balance is non-deterministic when using threads
When Do Race Conditions Happen?

- Race conditions happen when
  - Two threads access a common resource
  - The scheduler happens to switch between the two threads
  - The threads process data in an inconsistent state

If threads are so finicky, can I just not use them?

Notes:

<i>When Do Race Conditions Happen?</i>
Java Environments and Threading

• In Java, it’s hard not to use threads
  — Java regular I/O blocks until there is input
    – Applications that do lots of I/O need to do the I/O in separate thread
    – Otherwise, poor responsiveness
  — Or use NIO with Selector
  — Swing GUI components execute in a separate non-daemon thread
    – Need to perform resource-intensive operations in some other thread
    – Or GUI won’t be responsive

• J2EE applications are inherently threaded
  — Servlet engine could run same instance of servlet in different threads
  — Think of servlet’s `doGet()` as being in a `Runnable`
  — J2EE container-managed components are also inherently threaded

Notes:

<ipf>R,14: Java Environments and Threading</ipf>
Simple and Safe Programs

- **Start with a no-shared-data approach**
  - Use local variables as much as possible
  - For example, in a servlet:
    - No instance variables
    - Anything in an HttpSession is shared
    - Recreate objects or get from request as needed

- **Avoid the use of non-final static variables**
  - Static variables are shared between object instances
  - Static variables need to be synchronized

Notes:

*L,15: Simple and Safe Programs*L

The simple workaround. Tell them that the problem with the previous demo was that the Account was shared. If you can design to have no shared data, no sync needed, no race Conditions possible. Quite pleasant.

Put a (*) on this slide. This is what they should do as often as they can.
Synchronization

Synchronization can prevent access to data in an inconsistent state

- One thread obtains a monitor (lock) on an object
- No other thread can get the lock until this thread releases it
- If the scheduler switches between the two threads, the second thread waits

```java
public void performBanking(Customer customer) {
    Account account = customer.getAccount();
    synchronized(account) {
        // do stuff with account here
        double balance = account.getBalance() + 100;
        account.setBalance(balance);
    }
}
```

Notes:

<ipf>L.16: Synchronization</ipf>

dm53_sync: show them Clerk.java
Synchronization, Explained

- **Every object has a lock**
  - Synchronizing on the object waits for the lock if object is in use
  - Synchronization protects the state of objects
  - Ensures objects are not in inconsistent state

- **Two different pieces of code can be synchronized on same object**
  - No two threads can be in either segment of code at same time

```
No sync
methodA(object)
methodB(object)

Both methods sync on object that they share
methodA(object)
methodC()
methodB(object)
```

**Notes:**
R,17: Synchronization, Explained
**Synchronized Methods**

- **Synchronized blocks place the burden of correct programming on the client code of the `Account` class**
  - Why is this a problem?

- **Synchronized methods move the burden back to the `Account` class**
  - The `withdraw()` and `deposit()` methods have to be synchronized
  - Just like calling `synchronized(this)`

```java
public class AccountImpl implements Account {
    public synchronized void withdraw(int amount) {
        this.setBalance(this.getBalance() + amount);
    }
    private synchronized double getBalance() {
        return balance;
    }
    private synchronized void setBalance(int amount) {
        balance = amount;
    }
}
```

**Notes:**

- (a) maintainability
- (b) point them to previous slide. Synchronizing only the `getBalance()` and `setBalance()` methods do nothing
- If the scheduler switches right between them.
- If you make `getBalance` and `setBalance` private, then no need to sync them. But don’t want to get into all that.
When to Synchronize

To decide whether to synchronize a method, ask:

— Are objects of this class shared?
— Will this method be called by clients in different threads?
— Does the method require an instance variable?

```java
public class AccountImpl implements Account {
    public synchronized void withdraw(int amount) {
        this.setBalance(this.getBalance() + amount);
    }
    private synchronized double getBalance() {
        return balance;
    }
    private synchronized void setBalance(int amount) {
        balance = amount;
    }
}
```

Notes:

<ipf>R,18: When to Synchronize </ipf>

(a) maintainability
(b) point them to previous slide. Synchronizing only the getBalance() and setBalance() methods do nothing
If the scheduler switches right between them.

If you make getBalance and setBalance private, then no need to sync them. But don’t want to get into all that.
Synchronized Wrappers

- To avoid the problem of synchronizing objects even for clients who don't need synchronization
  ➡️ Create a synchronization wrapper

```java
public class SynchronizedAccount implements Account{
    private Account account; // set by constructor
    ...
    public synchronized void withdraw(int amount){
        account.withdraw(amount);
    }
}
```

Synchronization essentially makes the access to an object sequential (i.e., slow), so leave Account unsynchronized for customers who don't have shared accounts.

Notes:

<ipf>L.19: Synchronized Wrappers</ipf>

May want to tell them about Vector, ArrayList and synchronized ArrayList
Synchronized Wrapper

- **Open the Eclipse project dm51_threads**
  1. Complete the implementation of SynchronizedAccount
     — Complete the steps marked TODO
  2. In SharedAccountMain.java, replace acct1 by a new instance of SynchronizedAccount that wraps the original AccountImpl
     — Follow the steps marked TODO
  3. Run the Ant target run_sharedacct
     🎉 Are the results reliable now? Why?
  4. Note that acct2 is not a SynchronizedAccount
     🎉 What benefit does this provide?

**Notes:**

<ipf>R,20: Synchronized Wrapper</ipf>
Taking Full Advantage of Threads

Parallel Execution Using Threads
Bulletproofing a Threaded Application
Managing Implications of Synchronization
Hands-On Exercise 5.1

Notes:
<ipf>R,21: Taking Full Advantage of Threads</ipf>
There may be some people who don’t realize that this is relevant. Tell them servlets are automatically Threaded and you will get their attention.
Be Careful When Obtaining Multiple Locks!

- **A deadlock could happen if**
  - Two locks are acquired in different orders in two different threads

  ```java
  // somewhere
  synchronized(acct){
    synchronized(clerk){
      clerk.handle(acct);
    }
  }
  // somewhere else
  synchronized(clerk){
    synchronized(acct){
      clerk.handle(acct);
    }
  }
  ``
  
  - Application “hangs”

Notes:

<ipf>L,22: Be Careful When Obtaining Multiple Locks!</ipf>
Avoiding Deadlocks: Best Practices

- **Deadlocks are the hardest threading problems to diagnose**
  - Depends on the timing being "just wrong"
  - Get into the habit of ordering locks correctly

- **If synchronization can be avoided, deadlocks go away**
  - Start with a "no shared data" approach
  - Avoid premature optimizations
  - The performance benefit of sharing data may be outweighed by increased maintenance cost due to synchronization and deadlock issues

- **Several other suggestions to avoid synchronization:**
  - `volatile`
  - `atomics`
  - `ThreadLocal`

**Notes:**

<ipf>L,23: Avoiding Deadlocks: Best Practices</ipf>
Avoiding Synchronization Using volatile

• The keyword volatile has now changed
  — Pre-Java 1.4
    – Every thread was guaranteed to see latest value of volatile fields
    – Even when the data are shared between different threads
    – But other fields of object might still be wrong
        – Cause of many subtle bugs in threaded programs
  — After Java 1.4
    – Every thread guaranteed to see latest value of volatile fields
        – And all other fields of object

• Performance implications of volatile now close to that of synchronization
  — But deadlock cannot happen
  — Many JVMs still do not implement volatile correctly
  — It is difficult to correctly use volatile

Notes:

<ipf>L.24: Avoiding Synchronization Using volatile</ipf>
See Brian Goetz’ article on new Java memory model

Also there are some volatile demos in the samplecode project.
They illustrate how hard volatile programming can be.
You may even end up on a chipset where even the Test program
won’t work.
Avoiding Synchronization Using Atomics

- Java 5 extends the volatile concept to other types
  - Thread-safe programming without locks
  - AtomicReference, AtomicLong, etc.
  - Part of java.util.concurrent.atomic

- Ensures that object is modified in one operation
  - So that object is not left in inconsistent state

- Atomics do not provide an alternative to synchronization
  - References to single field are atomic
  - Nothing guaranteed about references to multiple atomic fields in class

Notes:

<ipf>R,25: Avoiding Synchronization Using Atomics</ipf>

Again, sample code illustrates this if you get a challenge.
Avoiding Synchronization Using ThreadLocal

- Can avoid synchronization if each thread has separate copy of data
  - A ThreadLocal provides this capability
    - Whereas static provides one value per class loader

```
private ThreadLocal<DAOCache> daoCaches = new ThreadLocal<DAOCache>();
public IndexDAO getDAO(String source){
    DAOCache daoCache = daoCaches.get();
    if (daoCache == null){
        daoCache = new DAOCache();
        daoCaches.set(daoCache); // for this thread
    }
    return daoCache.getDAO(source); // implements actual caching
}
```

- This is a tradeoff: with a separate DAOCache per thread
  - DAOCache does not have to be a singleton
  - No race conditions or threading issues
  - But have more instances of variable

Notes:

<ipf>R,26: Avoiding Synchronization Using ThreadLocal</ipf>
First two are not best practices as regards avoiding synchronization. ThreadLocal is. There are now separate DAOs per thread.

daoCaches is now an instance variable that looks like a local variable.
Smarter Synchronization

- If data need to be shared, synchronization cannot be avoided
  — But can be done smarter

- Performance implications of synchronization can be reduced
  🌟 Use appropriate locking mechanism (Java 5 and higher)
    - Semaphore
    - ReentrantReadWriteLock
    - Many others, consult API documentation
  🌟 Use thread notification mechanisms to “wake” up waiting threads

Notes:

<ipf>R,27: Smarter Synchronization</ipf>
Semaphores Allow Statement-Level Synchronization

- A semaphore maintains a set of permits
  - Allows statement-level synchronization
    - If no permits are available, the thread blocks

```java
private Semaphore sm = new Semaphore(1); // only one permit: Step 1
public void deposit(int amount) {
    sm.acquireUninterruptibly(); // Step 2
    synchronized(this){
        super.deposit(amount); // Step 4
    }
    sm.release(); // Step 5
}
```

1. Create a semaphore with the preset number of permits
2. Clients acquire a permit
3. Obtain any other locks as needed
4. Perform operation on resource
5. Release permit

**Notes:**

<ipf>R,28: Semaphores Allow Statement-Level Synchronization</ipf>
When Reading Is More Common Than Updates

```java
class TelephoneDirectory{
    private HashMap<String, Phone> map;
    private ReentrantReadWriteLock rwLock =
        new ReentrantReadWriteLock();
    public Phone getPhone(String name){
        Lock readLock = rwLock.readLock();
        readLock.lock();
        try{
            return map.get(name);
        }finally {
            readLock.unlock();
        }
    }
    public void setPhone(String name, Phone ph){
        Lock writeLock = rwLock.writeLock();
        writeLock.lock();
        try{
            map.put(name, ph);
        }finally {
            writeLock.unlock();
        }
    }
}
```

- **ReentrantReadWriteLock**
  - Optimized for single writer but multiple readers

- **For example**
  - Telephone directory
  - `getPhone()` could be called by multiple threads
    - The threads don’t wait unless there’s a `setPhone()` going on

Notes:

<R,29: When Reading Is More Common Than Updates</R>

Try, finally needed in case of run-time exception
**Wait and Notify**

- **java.lang.Object** has **wait() and notify() methods**
  - `wait()` waits for some other thread to call `notify()`
  - `notify()` wakes up any one waiting thread
  - Can only be called after the lock on that object is obtained

```java
public synchronized void withdraw(int amount) {
    try {
        super.withdraw(amount);
    } catch (OverdrawnAccountException e) {
        try {
            this.wait(1000L);
        } catch (InterruptedException e1) {
            super.withdraw(amount);
        }
    }
}

public synchronized void deposit(int amount) {
    super.deposit(amount);
    this.notifyAll();
}
```

**Notes:**

<ipf>L,30: Wait and Notify</ipf>
Need for Thread Pools

- Explicitly creating Thread objects and calling start is wasteful
  - Threads are quite heavy in weight
  - Should not be creating threads for short-lived tasks
  - Would be better if they could be reused

- A clerk may need to serve customers as they come in
  - Threads should not be assigned to just one Runnable
  - Need to have a pool of threads in which ThreadedBanking will run

Notes:

<ipf>L.31: Need for Thread Pools</ipf>
ExecutorService

- Java 5 provides mechanisms to interact with thread pools
  - Called ExecutorService
    
    ```java
    ExecutorService threadpool = Executors.newFixedThreadPool(3);
    Callable<Customer> c = new ThreadedBanking(customers[i], amt);
    Future<Customer> f = threadpool.submit(c);
    
    newCachedThreadPool for unlimited-size thread pool
    ```
  - A future can be used to
    - Determine whether method is done
    - Obtain result of method when it finishes
    
    ```java
    if (f.isDone()) { ... }
    Customer result = f.get();
    ```

Notes:

<pip>L.32: ExecutorService</pip>
Taking Full Advantage of Threads

Parallel Execution Using Threads
Bulletproofing a Threaded Application
Managing Implications of Synchronization

Hands-On Exercise 5.1

Notes:

<ipf>R,33: Taking Full Advantage of Threads</ipf>

There may be some people who don’t realize that this is relevant. Tell them servlets are automatically Threaded and you will get their attention.
Hands-On Exercise 5.1:
A More Responsive Application

- In this exercise, you will learn how to
  - Identify potential threading hazards
  - Implement a thread pool and tasks to execute within a thread pool
  - Build a more responsive application

Please refer to the Exercise Manual

Notes:

<ipf>L,34: Hands-On Exercise 5.1: A More Responsive Application</ipf>

Tell them that good design and best practice can help their code not have too many hazards in the first place.

In the ResultsPanel methods, instance variable numProcessing is being used. The other classes don't need sync.

Q: You are not using SwingUtilities to update the GUI from the application thread.

A: I don't think this goes onto the load without a proper discussion in the slides, and there is no time to do so. The code is correct as it now stands, because all the Swing text UI classes are thread safe.
Review

1. Can threads improve performance on a single-CPU machine?

2. Why are static variables problematic in threaded applications?

3. Why is it a best practice to create synchronization wrappers instead of making the original methods synchronized?

Notes:

<ipf>R,35: Review</ipf>
Best Practices: Taking Full Advantage of Threads

- **Threads improve performance through parallelization and responsiveness**
  - To program with threads, implement `Runnable` or `Callable`
  - Program as if switching could happen at any time

- **Design for minimal data sharing between threads**
  - Prefer local/instance variables to static variables
  - Use `ThreadLocal` to avoid static in resources such as factories
  - `volatile` and `atomic` are problematic replacements for synchronization

- **Identify potential threading hazards**
  - Create synchronization wrappers to prevent race conditions
  - Order locks correctly to prevent deadlocks
  - Use semaphores for statement-level synchronization
  - Use read/write locks in cases where there are few modifications

- **Prefer executors for short-lived tasks that need to run in separate thread**

**Notes:**
<ipf>R,36: Best Practices: Taking Full Advantage of Threads</ipf>
Chapter Summary

In this chapter, you have learned how to
• Improve response time by parallelization
• Bulletproof a threaded application
• Share data between threads
• Manage the implications of synchronization
• Increase user impression of responsiveness using threads

Notes:

<ipf>L,37: Chapter Summary</ipf>
Chapter 6

Extending Application Functionality

Notes:

<ipf>B,1: Extending Application Functionality</ipf>
Chapter Objectives

In this chapter, you will learn how to extend an application by

- Injecting dynamic behavior with proxies
- Remotely managing the application at runtime
- Scripting in changes to its rules engine

Notes:

<ipf>L,2: Chapter Objectives</ipf>

Next slide has same information
Extending Application Functionality in Java

- **OO ways of extending application functionality in Java:**
  - Write a new class and add it to the project
    - Could be a subclass of an existing class
    - Design to interfaces
  - Edit or add methods to an existing class
    - Not maintainable

- **Other ways of extending application functionality in Java**
  1. Add some behavior to a large number of classes at once
    - Called an *aspect*
  2. Permit remote control of existing class at runtime
    - Through Java Management Extensions (JMX), for example
  3. Allow users to specify the code in runtime
    - End-user programming can be achieved through *scripting*

- **We will examine these three techniques in this chapter**

**Notes:**

R,3: Extending Application Functionality in Java

Tell them that we looked at solution 1 extensively in Chapter 3.
We’ll look at aspects, JMX and scripting in this chapter.
Extending Application Functionality

- Injecting Behavior With Proxies
- Managing Applications With JMX
- Incorporating Scripting Into Rules Engines
- Hands-On Exercise 6.1

Notes:

<ipf>L,4: Extending Application Functionality</ipf>
Factory With a Proxy

- Recall that the factory can decide which class to create
  - Doesn't need to return the JazzInJune promotion
  - Can add a Loyal Customer Discount transparently, for example
  - ★ Called the Proxy pattern

```java
// In PromotionFactory.java
Promotion getPromotion(String name) {
    String className = config.getClassName(name); // what user expects
    Promotion promo = (Promotion) Class.forName( className ).newInstance();
    BestOfPromotion bestof = new BestOfPromotion();
    bestof.add( promo );
    bestof.add( new LoyalCustomerPromotion(...) ); // what user gets
    return bestof;
}
```

Notes:

<ipf>R,5: Factory With a Proxy</ipf>
reminder
The proxy intercepts calls to service
- Adds behavior transparently to specific methods of service
- By implementing those methods explicitly

What if we need to add behavior transparently to arbitrary methods?
- For example, count the number of times that any Promotion is analyzed

Notes:
<ipf>R,6: Factory With a Proxy (continued)</ipf>
reminder
Dynamic Proxy

Dynamic proxies allow addition of behavior transparently
- The Proxy object is created at runtime by the JVM
  - Uses a specified InvocationHandler to inject behavior
- The original service class doesn’t have to change

Notes:

<ipf>L,7: Dynamic Proxy</ipf>

They don’t need to write the Proxy, but do need to write the InvocationHandler.
So, what’s the point?
The same InvocationHandler works for any method of the service, and potentially for different services too
(next slide).
**InvocationHandler**

- **Implement InvocationHandler and its invoke method**
  - Call method on original service object through reflection
  - Works for any method of the service

```java
public class AnalysisCounter implements InvocationHandler {
    private PromotionAnalysisService service = new PromotionAnalysisServiceImpl();
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        for (int i=0; i < args.length; ++i){
            if (args[i] instanceof Promotion){
                incrementCount((Promotion) args[i]);
            }
        }
        return method.invoke(service, args);
    }
    // implementation of proxy-specific methods
}
```

**Notes:**

L,8: InvocationHandler
Dynamic Proxy

- The proxy creation itself is boilerplate code
  - Don’t place in client code—put it in a factory instead
    - Client code uses a factory to create an object
      - Does not need to be aware that it is using a proxy at all

```java
// In PromotionAnalysisServiceFactory.java
InvocationHandler analysisCounter = AnalysisCounter.getInstance();
PromotionAnalysisService service = (PromotionAnalysisService)
    Proxy.newProxyInstance
        (this.getClass().getClassLoader(),
         new Class[]{PromotionAnalysisService.class},
         analysisCounter);
```

- Can apply to any/all methods of the service interface!
  - Adds an aspect to an existing class

Notes:

<ipf>R,9: Dynamic Proxy</ipf>

Tell that they will get to play with dynamic proxies in the exercise.
The example assumes (as in the code they will write) that AnalysisCounter is a singleton.
Extending Application Functionality

- Injecting Behavior With Proxies
- Managing Applications With JMX
- Incorporating Scripting Into Rules Engines
- Hands-On Exercise 6.1

Notes:
<ipf>L,10: Extending Application Functionality</ipf>
What Is JMX?

- **Java Management Extensions (JMX)**
  - Tools for remote management of applications, networks, and devices
  - The JMX API is part of Java 5
    - `java.lang.management`
    - `javax.management`

- **Use JMX to control objects in applications remotely**
  - JMX is a standard
  - Don’t build custom solutions

- **Best choice wherever management and monitoring are needed**
  - Can use JMX tools to monitor memory and CPU usage of Java applications

---

**Notes:**


The best practice focus of this section is simply that if they need to remotely modify Behavior, they should use JMX, not build custom solutions.
How to Use JMX

- The code being managed needs to
  1. Implement interfaces
     - By convention, the name of the interface ends with `MBean`
     - “Managed Bean”
  2. Register bean with JMX server
  3. Be launched with remote manageability turned on

Notes:

<ipf>R,12: How to Use JMX</ipf>
Step 1: Implement an Interface

Create an interface for those methods that should be remotely callable
— Changes the property of your domain object

```java
public interface AnalysisCounterMBean {
    public void resetAllCounters();
    public void setCounter(String promoCode, int newValue);
    public int getCounter(String promoCode);
}
```

```java
public class AnalysisCounter implements AnalysisCounterMBean {
    ...
    public void resetAllCounters(){
        myCounters = ...;
    }
    public void setCounter(String promoCode, int newValue){
        myCounters.put( promoCode, newValue );
    }
    public int getCounter(String promoCode){
        ...
    }
    ...
}
```

Notes:

<ipf>L,13: Step 1: Implement an Interface</ipf>

Interfaces are a best practice anyway
Step 2: Register Bean With JMX Server

- Works regardless of the platform that the code is running on
  - Creates new MBeanServer if none currently running

```java
MBeanServer mbeanServer = ManagementFactory.getPlatformMBeanServer();
ObjectName mbean = new
  ObjectName("com.ltree.crs516.datamining:type=AnalysisCounter,name=counter");
mbeanServer.registerMBean(counter, mbean);
```

- Like package name, class name, instance variable name

Notes:

If you don’t use the naming convention shown in this example, need bean configuration files
Step 3: Remote Management On

- Enable manageability for an application with the system property:

  -Dcom.sun.management.jmxremote

- Can now remotely access properties and call MBean methods
  — Use JConsole—a Swing application that is part of the Java 5 SDK
  — Many application servers provide a JMX Web application
    – And have directions on enabling manageability for a Web application

Notes:

<ipf>L,15: Step 3: Remote Management On</ipf>
JConsole requires installation on a drive with user permissions
Step 3: Remote Management On (continued)

Name bean was registered with
Corresponds to methods in MBean interface
Managed application

Notes:

[ipf>L,16: Step 3: Remote Management On (continued)</ipf>
JConsole requires installation on a drive with user permissions
Extending Application Functionality

Injecting Behavior With Proxies
Managing Applications With JMX
Incorporating Scripting Into Rules Engines
Hands-On Exercise 6.1

Notes:
<i>Extending Application Functionality</i>
Adding Behavior to Deployed Applications

- Users often want to customize a deployed application
  - Programmers may not be available to make changes for them
  - The changes that users want to make are often quite simple
    - “Just try something out”

Can provide for user customization with scripting engines

Notes:

<ipf>L.18: Adding Behavior to Deployed Applications</ipf>
Quite convenient if the scripting engine were to be Java
BeanShell

- **BeanShell is a Java interpreter that supports scripting**
  - Will be part of J2SE in the future (JSR 274)
  - Access to all the Java APIs
    - JDBC, collections, etc.

**Use BeanShell to**
- Provide user interactivity to application
- Read complex configuration information
  - Configuration could be Java source code
- Execute remotely submitted jobs
- Embed Java expression evaluation in business logic
- Have two-way interaction between Java programs and user scripts

**Avoid using BeanShell as loosely typed Java**

---

**Notes:**

<ipf>R,19: BeanShell</ipf>

Ian Darwin reports:

Java 6 will include a mechanism for "scripting engines"
Allows you to plug in almost any scripting language as an extension mechanism for your application
  - Perl, Python, Ruby, JavaScript, bsf
A few such as BSF can be used in Java 1.4 or 1.5
Incorporating BeanShell

- **Launch a BeanShell interpreter**
  - And evaluate the source code that user supplied
    ```java
    String sourceCode = ...;
    Interpreter bsh = new Interpreter();
    Object o = bsh.eval(sourceCode);
    ```

- **To read and evaluate source code from a file:**
  ```java
  Interpreter bsh = new Interpreter();
  Object o = bsh.source("startup.bsh");
  ```

Notes:

<ipf>L,20: Incorporating BeanShell</ipf>
BeanShell Scripts

- The BeanShell script itself can be any Java code

```java
import java.util.ArrayList;
ArrayList files = new ArrayList();
files.add("sales_20060721.xml");
files.add("sales_20060628.xml");
Collections.sort(files);
return files;
```

- The evaluating code would then cast the returned object to an `ArrayList` Why?

```java
bsh.Interpreter bsh = new bsh.Interpreter();
ArrayList files = (ArrayList) bsh.source("customfiles.bsh");
```

**Notes:**

<i>Next> R.21: BeanShell Scripts</i>

Need to stress the second point: that the script evaluator has to cast back into the object returned by the script.

The scripts are means to an end in Java – the script is not completely arbitrary.
BeanShell Classes

Powerful behavior results if the script implements an interface
— User able to plug custom behavior into heavyweight classes at runtime
— The Strategy design pattern

class LargeFiles implements FileProvider
{
   public ArrayList getFiles(){
      ...
   }
}

• The evaluating code would then cast the returned object to a class

Why?

bsh.Interpreter bsh = new bsh.Interpreter();
Class ret = (Class) bsh.source("customfiles.bsh");
FileProvider fp = (FileProvider) ret.newInstance();
Files[] files = fp.getFiles().toArray(new File[0]);

The evaluation code depends on the script. Therefore, restrict the leeway that users have in writing scripts.

Notes:

<ipf>L,22: BeanShell Classes</ipf>
Extending Application Functionality

Injecting Behavior With Proxies
Managing Applications With JMX
Incorporating Scripting Into Rules Engines

Hands-On Exercise 6.1

Notes:
<ipf>L,23: Extending Application Functionality</ipf>
Hands-On Exercise 6.1: Scripting a Promotion

- In this exercise, you will learn how to extend the application functionality
  - Through user-driven scripting
  - With a dynamic proxy
  - By remotely changing its settings

Please refer to the Exercise Manual

Notes:

<ipf>L.24: Hands-On Exercise 6.1: Scripting a Promotion</ipf>
Make to demo the bonus in class.
Notes:

<ipf>R,25: Review</ipf>

(a) addition of behavior to arbitrary methods, not just the specific method
B) For remote interaction
(c) user customization
Best Practices: Extending Application Functionality

- To extend application functionality:
  - Design to interfaces
  - Inject behavior by applying aspects using a dynamic proxy
  - Permit remote control of objects by making them managed beans

- Can provide for end-user customization with scripting engines
  - Use BeanShell for user interactivity and expression evaluation
  - Avoid using BeanShell as loosely typed Java
  - Powerful behavior results if the script implements an interface

Notes:
<ipf>L,26: Best Practices: Extending Application Functionality</ipf>
Chapter Summary

In this chapter, you have learned how to extend an application by

• Injecting dynamic behavior with proxies
• Remotely managing the application at runtime
• Scripting in changes to its rules engine

Notes:

A quick chapter presenting concepts that we will build on throughout the course.
Chapter 7

Implementing Security Best Practices

Notes:

B,1: Implementing Security Best Practices
Chapter Objectives

In this chapter, you will learn how to

• Restrict access to protected resources
• Provide clearly circumscribed Java applications to end users
• Avoid security pitfalls

Notes:
<ipf>L,2: Chapter Objectives</ipf>
A quick chapter presenting concepts that we will build on throughout the course.
Implementing Security Best Practices

- What Java Security Entails
  - Authentication and Authorization
  - Hands-On Exercise 7.1
  - Securing Against Untrusted Applications
  - Securing Against Untrusted Code
  - Secure Coding

Notes:

Java Security: Who Is It For?

- **Addresses the security of the owner of the Java Virtual Machine**
  - Does not address security of the writer of Java software
  - To keep Java source code confidential, obfuscate the byte code
    - Limits its use, since code cannot be invoked from users’ applications
    - Encrypting byte code and using custom classloader doesn’t work
      - Someone with JVM source can easily access plaintext byte code

- **Java programmers and users of Java applications should**
  - Enable security features
  - Configure security properly
  - Ensure that trusted code does not leave itself open to malicious attack

*Notes:*

Java Security: What’s Provided

- Java security provides ways to address three scenarios
  - Limit what untrusted Java applications can do
  - Assign different levels of trust to different software and users
  - Limit what untrusted users can do

Notes:

<ipf>R,5: Java Security: What’s Provided</ipf>
Java Security Scenarios: Provided vs. TODO

- **Untrusted application on end user's machine**
  - Java security provides way to limit what software can do
  - Need to secure JVM installation to prevent end-run around those constraints

- **Application built from trusted and untrusted code**
  - Provides way to limit what distrusted packages in application can do
  - Need to prevent hackers from acting like a trusted provider

- **Untrusted user accessing or supplying input to Java application**
  - Java security provides way to limit what user can do
  - Need to avoid pitfalls by which hackers can do harm through their inputs

**Notes:**

The second situation is of course that in many commercial servlet environments where another web application could have been written by a malicious competitor. The Java application in that case is the servlet engine and plugin.jar the various War files.

May want to add this to favorites and refer back to this slide as you go through Chapter so that students have the big picture in context. Do it whenever you see One of the these pictures on the slide.
Implementing Security Best Practices

What Java Security Entails

► Authentication and Authorization

Hands-On Exercise 7.1

Securing Against Untrusted Applications

Securing Against Untrusted Code

Secure Coding

Notes:

JAAS Jazzes Up Java Security

- **Java Authentication and Authorization Service (JAAS)**
  - User-based permissions for Java security
  - Augments codebase security in Java 2
  - Hooks into existing standard authentication mechanisms

- **JAAS is configured with two configuration files**
  - Authentication configuration file
  - Authorization configuration file

- **Java code should check permission before it undertakes critical functions**

**Notes:**

*ipf>L,8: JAAS Jazzes Up Java Security</ipf>*

May want to talk about code-based security (first scenario) and its limitations
Configuration File 1:
java.security.auth.login.config

- The authentication (login) configuration file
  - Specifies ways to authenticate a principal
  - A subject can be represented by many principals
    - EmployeeID, Windows Logon, Database user name, etc.
  - Pre-existing LoginModules are available for various authentication scenarios (UNIX, Kerberos, JNDI, etc.)
    - Can also implement custom LoginModule

```
RainForest {
    com.sun.security.auth.module.NTLoginModule required debug=true;
}
```

- Specify location of file with system property

```
java.security.auth.login.config=C:/config/login_config.txt
```

Notes:

R,9: Configuration File 1: java.security.auth.login.config

Required states that the user needs to be authenticated. No “guest”
Debug=true simply makes the login module more verbose.
The authorization policy file
- Specifies what permissions are to be given to which users and applications

grant codebase "file:rainforest.jar" signedby "crs516"
Principal com.sun.security.auth.NTUserPrincipal "peter.piper" {
  permission java.io.FilePermission "*", "read,write";
  permission java.util.PropertyPermission "*", "read,write";
  permission java.net.SocketPermission "localhost:*", "connect, resolve";
};

Any permission not explicitly granted is denied
Specify location of file with system property

java.security.policy=C:\config\permissions.txt

Notes:
<ipf>L,10: Configuration File 2: java.security.policy</ipf>
Next slide ties this back to the three scenarios discussed at beginning of chapter;
AND of all conditions specified
There is an AllPermissions which is not a good idea
JAAS Addresses All Three Scenarios

- **Untrusted applications**
  - codebase gives permissions based on where code is installed

- **Untrusted users**
  - Principal gives permission based on who’s running the code

- **Trusted and untrusted code**
  - signedby gives permissions based on who the code is from

- codebase, signedby, and Principal are all optional

**Notes:**

<i>R,11: JAAS Addresses All Three Scenarios</i>
Security Manager

- By default, applications do not have a security manager
  - Set system property to enable one

```java
<java classname="com.ltree.crs516.gui.MainFrame" fork="yes">
  <jvmarg line="-Djava.security.manager"/>
  <jvmarg line="-Djava.security.auth.login.config=login_config.txt"/>
  <jvmarg line="-Djava.security.policy=permissions.txt"/>
  <classpath>
    <path refid="project.classpath">
    </path>
  </classpath>
</java>
```

Notes:

<R,12: Security Manager</R>

Can also do a System.setSecurityManager(…) to force a security manager even if the flag is not set at startup.
Can also pass in a subclass of SecurityManager here.
Verifying Permission

- To verify if codebase is allowed to do something
  - Call checkPermission() on SecurityManager
  - Method may throw SecurityException or silently return

```java
SecurityManager sm = System.getSecurityManager();
if ( sm != null ){
    sm.checkPermission(new FilePermission("commissions.txt", "read") );
}
```

Notes:

<ipf>R,13: Verifying Permission</ipf>
Authenticating a User

• To authenticate a user:

```java
CallbackHandler myCallback = new MyDialogWindow(...);
LoginContext loginContext = new LoginContext("RainForest", myCallback);
loginContext.login();
Subject subject = loginContext.getSubject;
```

— If no exceptions were thrown, Subject has been authenticated

Notes:

<ipf>L,14: Authenticating a User</ipf>

XXX break into two slides

The Solaris roots show here -- doAsPrivileged is a lot like sudo (switch user and do) on Unix
Checking Authorization on a User

To check authorization, run critical code inside `Subject.doAsPrivileged()`

```java
SecurityManager sm = System.getSecurityManager();
Permission p = new FilePermission("commissions.txt", "write");
Subject.doAsPrivileged(subject, new PermissionChecker(sm, p), null);

class PermissionChecker implements PrivilegedExceptionAction {
    // set sm, p in constructor
    public Object run(){
        sm.checkPermission(p);  // may throw SecurityException
        return null;
    }
}
```

Notes:

<ipf>L,15: Checking Authorization on a User</ipf>

Access control context: do we use the entire stack trace or just this method?
Usual to do “null” here, otherwise, in a servlet, you’ll be getting privileges of the person
Running Tomcat, not the person on behalf of whom the servlet is currently executing method.
The Solaris roots show here -- doAsPrivileged is a lot like sudo (switch user and do) on Unix
Custom Permission Class

1. Implement fine-grained security by creating a Permission subclass

   ```java
   public class AnalyzePermission extends BasicPermission {
       public AnalyzePermission(String name, String actions) {
           super(name, actions);
       }
       public AnalyzePermission(String name) {
           super(name);
       }
   }
   ```

2. Check authorization in restricted parts of application

   ```java
   // somewhere in business tier
   Permission p = new AnalyzePermission("RainForest");
   Subject.doAsPrivileged( subject, new PermissionChecker(sm,p), null );
   ```

3. User is allowed only if given the permission in authorization config file

   ```java
   grant Principal com.sun.security.auth.NTUserPrincipal "peter.piper" {
       permission com.ltree.crs516.AnalyzePermission "RainForest";
   }
   ```

**Notes:**

<ipf>R,16: Custom Permission class</ipf>
Implementing Security Best Practices

What Java Security Entails
Authentication and Authorization

Hands-On Exercise 7.1
Securing Against Untrusted Applications
Securing Against Untrusted Code
Secure Coding

Notes:
<i>R,17: Implementing Security Best Practices</i>
Hands-On Exercise 7.1: Restricting Access

- In this exercise, you will restrict the **Analyze** functionality to authorized users *only*, using JAAS

Please refer to the Exercise Manual

Notes:

Q: How should we distribute authentication files?

A: The authentication file is set by the system administrator and placed in a directory that is not modifiable by a regular user.

Q: How come log4j reports that it's not configured properly?

A: Log4J does not use privileged sections, so the only way to get it working is to give every caller of log4j (i.e. our application) java.security.AllPermission. Can't very well do that for this application. You can verify this by adding the following lines to permissions.txt:

```java
grant codeBase "file:./lib/log4j-1.2.13.jar" {
    permission java.security.AllPermission;
};
grant codeBase "file:./lib/commons-logging.jar" {
    permission java.security.AllPermission;
};
```

Log4J will now be able to find the logging file, but will fail with an access control violation. The only fix is to fix log4j's code or open up the entire application. I went with the compromise, of losing logging in the security exercise. java.util.logging is supposedly more friendly to the Java security model (I haven't used it though).
Implementing Security Best Practices

What Java Security Entails
Authentication and Authorization
Hands-On Exercise 7.1
- Securing Against Untrusted Applications
  Securing Against Untrusted Code
  Secure Coding

Notes:
Untrusted Applications

• If someone downloads an application from the Internet
  — Should the person be worried?

• What if the person could clearly specify
  — What files the application can read?
  — What machines the application can connect to?
  — What code the application is allowed to run?

• Java’s security allows users running Java applications to run them safely

Notes:
<ipf>L,20: Untrusted Applications</ipf>
Insert anecdote here.
Running Applications Securely

When running an untrusted Java application
— Always run it under a SecurityManager
— Find out what SecurityExceptions it throws
  – Grant only permissions that are safe to give

When distributing a Java application to end users
— Provide signed JAR files
— Publish certificates
— Create message digest checksums to prevent man-in-the-middle attacks
— State what permissions the application requires

In multi-user environments, secure the Java runtime installation
— Or rely on operating-system permissions on critical data
  – Never run a Java application as root or Administrator

Notes:
<ipf>L,21: Running Applications Securely</ipf>
How to Sign a .jar File

- **To sign a .jar file**
  - Get the private/public key pair for organization
  - Get public key signed by Certificate Authority (or self-sign)
  - Store keys in a keystore
  - Can be accomplished using the free keytool
    - Ships with the JDK

- **Use jarsigner to sign the JAR file using key in keystore**
  - jarsigner is a free tool that ships with JVM
  - If keystore contains a private key with the alias rainforest

```
jarsigner -keystore rainforest.keystore rainforest.jar rainforest
```

Notes:

<ipf>R,22: How to Sign a JAR File</ipf>
How to Assign Permissions to a .jar File

- The JAAS authorization config file permits granting permissions to code
  - In addition to or instead of principals

keystore file:/c:/mam.keystore;

grant codebase "file:rainforest.jar" signedBy "rainforest" {
  permission java.net.SocketPermission "promos.rf.com:8080", "connect, read";
};

- The keystore should contain a certificate (with public key) of RainForest
- Says that user trusts applications signed by RainForest in certain ways

Notes:

R,23: How to Assign Permissions to a JAR File
Securing JRE Installation in Multi-User Environments

- **For a secure JRE installation**
  - Make it hard for users to defeat security measures
  - Even if they run downloaded Java applications
  - Install JRE where user cannot modify settings

- **System administrators should not allow users to specify security policy**
  - Change `java.security` found in JRE installation directory
    - Delete the `${user.home}/.java.policy` line
    - `policy.allowSystemProperty=false`
    - `security.overridePropertiesFile=false`

- **System administrators should configure `security.policy`**
  - Found in JRE installation directory
  - Grant minimal permissions to everything
  - Grant critical resource access permissions only to signed codebase and authenticated users

*Notes:

<i>R.24: Securing JRE Installation in Multi-User Environments</i>
Implementing Security Best Practices

What Java Security Entails
Authentication and Authorization
Hands-On Exercise 7.1
Securing Against Untrusted Applications

- Securing Against Untrusted Code
  Secure Coding

Notes:

Securing Java in the Presence of Untrusted Code

- Sometimes trusted Java code may need to run alongside Java code from untrusted parties
  - In the same JVM

- This scenario occurs:
  - In commercial servlet-hosting environments
  - When one part of corporation is not allowed access to data used in another part of corporation
    - Employees’ health records, for example

Notes:

<ipf>L,26: Securing Java in the Presence of Untrusted Code</ipf>
Where relevant
Securing Java in the Presence of Untrusted Code (continued)

- Security is challenging when
  - You distribute signed, trusted code
  - To run in same JVM as untrusted code

- Have to rethink many issues
  - Affects every stage of design
  - Often affects extensibility of your code

- Avoid this situation as much as possible
  - Get a separate JVM to run applications with different trust levels
  - The Java APIs and Java security do not adequately handle this scenario

- Programming for this scenario is error prone
  - Will list means to address a few problem areas
  - Many more problem areas constantly being identified

Notes:

L.27: Securing Java in the Presence of Untrusted Code (continued)

Basically, avoid this situation as much as you can. The remaining 3 slides talk about this no-win scenario.
Build Security Façade

- **Add security mechanisms at entry to critical sections of code**
  - Create custom permissions and check against security policy
  - Ensure that all applications run with security policy
  - Keep privileged code short

- **At security façade**
  - Obtain complete call stack and verify that the call stack is expected
    - The package name alone does not uniquely identify a class
      - Use `==` to compare `Class` objects—not `Class.forName().equals()`
    - Mistrust all parameters to façade methods
      - Validate inputs
      - Make copies of mutable objects and arrays

- **Protect against external access to implementation packages**
  - Add `package.access` statement to `security.policy`
  - Give `RuntimePermission` to access package to façade only

**Notes:**

<ipf>R,28: Build Security Façade</ipf>
Java Pitfalls

- **Be careful with packages**
  - Use only private fields—not package-friendly, protected, or public fields
  - Don’t use inner classes in critical code
    - Inner classes are only a compile-time convenience
    - Inner classes are compiled into separate class
    - Compiler silently makes outer class fields package-friendly
  - Place all critical classes in a single package .jar file and seal it
    - Add Manifest entry sealed: true
  - Avoid non-final, public static fields
    - They can be accessed from outside classes
  - Make methods of critical classes final

- **Clear sensitive information, such as passwords, immediately after use**

**Notes:**

<i>l,29: Java Pitfalls</i>
Ensure That Objects Are Initialized

- Deserialization and cloning do not go through constructors
  - Prevent cloning by throwing exceptions from clone()
  - Prevent serialization by throwing exceptions from readObject()
  - Don't give Permission to subclass
    OutputStream/OutputStream

- Serialization can be a security loophole
  - Use transient rigorously
  - Never pass internal data to input/output streams in read/writeObject

Notes:
<ipf>R,30: Ensure That Objects Are Initialized</ipf>
Implementing Security Best Practices

What Java Security Entails
Authentication and Authorization
Hands-On Exercise 7.1
Securing Against Untrusted Applications
Securing Against Untrusted Code

Secure Coding

Notes:
Securing Java in the Presence of Untrusted Users

- It is up to the programmer to secure the application
  — Java provides basic capabilities only
- Make sure untrusted users cannot harm system

Notes:
<ipf>L.32: Securing Java in the Presence of Untrusted Users</ipf>
Most Critical Security Flaws

- **Open Web Application Security Project (OWASP) lists as most critical:**
  1. Unvalidated input
  2. Broken access control
  3. *Broken authentication and session management*
  4. *Cross-site scripting*
  5. Buffer overflow
  6. Injection of parameters to external system
  7. Improper error handling
  8. Insecure storage of data and keys
  9. *Denial-of-service attacks*
  10. *Insecure configuration management by application servers*

- **Let's look at those most relevant to general-purpose Java code**
  — Consider best practices to safeguard against these problems

Note: Italics signify items irrelevant to Java.

www.owasp.org

*Notes:*

<ipf>R,33: Most Critical Security Flaws</ipf>

Stuff in italics – they’re not relevant to Java
1. Unvalidated Input

🌟 Validate all user inputs
   — User inputs are tainted
   — Never use them directly
   — Verify and validate before use

• Verify and validate
   — System properties
   — Environment variables
   — Configuration files

Notes:

<ipf>L,34: 1. Unvalidated Input</ipf>

Ian Darwin:
You may want to say that this has been top of the list for years. For example, UNIX inventor Ken Thompson said that this was the cause of most bugs he’d seen in the 1970s.
Handling User Inputs in General-Purpose Packages

- If writing a general-purpose API
  - Users are programmers

- May be worthwhile to program defensively
  
  ```java
  public final class MyCriticalClass{
      private Date date;
      public MyCriticalClass(Date date){
          this.date = new Date(date.getTime());
      }
      public Date getDate(){
          return new Date(date.getTime());
      }
  }
  ```

  - Prevents problem if programmer calls `setTime()` on input parameter later
  - Or passes in a subclass of `Date` that overrides critical behavior

- Don’t overdo this
  - Typically, you want users to be able to pass in subclass (Strategy pattern)

**Notes:**

<R,35: Handling User Inputs in General-Purpose Packages</ipf>

Can NOT call clone() because of the subclass issue

Can do clone() in second instance, but better to be consistent
2. Broken Access Control

.staff


• Grant principal-based permissions
  — Perform permission checks in critical parts of code
  — Create custom permissions for business-logic restrictions

Notes:

<ipf>L.36: 2. Broken Access Control</ipf>
5. Buffer Overflow

• The JVM security prevents common ways of breaking into software:
  — Buffer overflow
  — Reading past the end of an array

• The byte code verifier will not execute code that does unsafe operations
  ★ Run Java from the byte code using the JVM
  — Do not use “pre-compiled to native executable” Java applications

• Programmer needs to be careful when invoking native operations
  ★ Avoid using Java Native Interface
  — C has no such checks

Notes:
<ipf>R,37: 5. Buffer Overflow</ipf>
gcj on Linux lets you create native Linux executables.
6. Injection of Parameters to External System

- SQL injection happens when user input:
  - Contains literal escape characters
  - Is directly embedded into SQL query
  - Is not checked for type safety

```java
String sql = "SELECT * FROM purchases where itemCode='" + input + ";

String input = 43' OR 'hack'='hack;

String input = 43'; DROP table promotions; SELECT * from purchases where itemCode='43";
```

**Notes:**

R,38: 6. Injection of Parameters to External System

The first hack always returns true

The second hack drops another table, but is transparent to application.
Preventing SQL Injection Attacks

Use PreparedStatement

```java
PreparedStatement stmt = conn.prepareStatement("SELECT * FROM promotions
where couponCode = ?");
stmt.setString(1, input);
```

Tie down database privileges: don’t allow applications to drop tables

Notes:

&lt;ipf&gt;R,39: Preventing SQL Injection Attacks&lt;/ipf&gt;
Injection Into Operating System

- Injection can also happen via `System.exec()`
  - Don’t invoke native scripts directly from Java application
    - And definitely don’t pass user parameters to such scripts
    - If you do, validate the command line

Notes:

<ipf>L.40: Injection Into Operating System</ipf>
7. Improper Error Handling

- Improper error handling can put software in indeterminate state
  - Explicitly test bad parameters
  - Including parameters that would throw runtime exceptions
  - Ensure that objects clean up in the case of exceptions
  - Place finally blocks to handle this case

```java
public CreditCheck checkCredit(Customer c, Purchase p){
    CreditReport report = null;
    try{
        // any of these methods may throw run time exception
        report = CreditAgency.getReport(c.getUserName());
        report.verifyCreditLimit(p.getSalePrice());
        return new CreditCheck(…);
    }
    finally{
        // clean up credit report
        if (report != null){
            report.close();
        }
    }
}
```

Notes:

<ipf>R,41: 7. Improper Error Handling</ipf>
8. Insecure Storage of Data and Keys

- Use a keystore to store encryption keys
- Use certificates to transfer keys
- Encrypt with private key to sign message
  - Recipients can trust that message came from signing party
- Encrypt with public key of recipient to preserve confidentiality
  - No one but recipient can decode message

- The Java tool `keytool` helps in managing keystores
- The Java cryptography APIs provide encryption and decryption capability

Notes:

<ipf>L.42: 8. Insecure Storage of Data and Keys</ipf>
Review

1. What is granted in the authorization policy file?

2. How can a user’s authorization be verified in code?

3. Name one way to safeguard against SQL injection attacks.

Notes:
<ipf>L.43: Review</ipf>
(b) permissions to users and applications
© Subject.doAs()
Prepared statement
Implementing Security Best Practices

- **Always run distrusted code with a SecurityManager**
  - Grant permissions only to signed .jar files and authenticated users
  - Install the JRE so as to prevent users from overriding security settings

- **Always distribute clearly circumscribed Java applications**
  - Explicitly list permissions required by code
  - Sign .jar files

- **Securing against untrusted code changes everything**
  - Seal packages, avoid inner classes, consider making classes final, design with immutability in mind, ensure that objects are initialized

- **Secure Java code**
  - Mistrust user inputs
  - Validate tainted variables
  - Explicitly test error handling
  - Use PreparedStatement

Notes:

<ipf>R.44: Implementing Security Best Practices</ipf>
Chapter Summary

In this chapter, you have learned how to

• Restrict access to protected resources
• Provide clearly circumscribed Java applications to end users
• Avoid security pitfalls

Notes:

<ipf>R,45: Chapter Summary</ipf>
A quick chapter presenting concepts that we will build on throughout the course.
Chapter 8

Course Summary

Notes:

<ipf>B,1: Course Summary</ipf>
Java Best Practices

- **Aim to write high-quality software**
- **Optimize your software development environment**
  - Use Ant for building so as to avoid dependence on IDEs
  - Unit tests should capture all the ways a class is currently used
  - Use a logging API that lets you selectively enable class logging
- **Improve code quality**
  - Consider both extensibility and maintainability
  - Apply design patterns to solve common design problems
  - Design for type safety and encapsulation
  - Throw exceptions only when caller needs to stop routine processing
  - Be aware of implicit contracts in the Java Core API
  - When designing frameworks, use interfaces to provide extension points
- **Tuning for performance should be performed only on bottlenecks**
  - Write better, faster code at the outset
  - Optimize loops and collections

Notes:

<ipf>L,2: Java Best Practices</ipf>
Summary of the summaries
Do this before you hand out the evals :)
Java Best Practices (continued)

- Threads improve performance through parallelization and responsiveness
  - Design for minimal data sharing between threads
  - Identify potential threading hazards
  - Prefer executors for short-lived tasks that need to run in separate thread

- To extend application functionality
  - Design to interfaces
  - Inject behavior by applying aspects using a dynamic proxy
  - Permit remote control of objects by making them managed beans
  - Provide end-user customization through scripting

- Implement security correctly
  - Run distrusted code with a security manager
  - Distribute clearly circumscribed applications
  - Secure Java code against untrusted users

- Bottom line: Aim to write high-quality software

Notes:

<i>ipf>R,3: Java Best Practices (continued)</i>

Summary of the summaries
Do this before you hand out the evals :)

© Copyright: All rights reserved. Not to be reproduced without prior written consent.
## Course Summary

In this course, you have learned how to

- Apply Java best practices to increase productivity and build fast, secure, and reliable applications
- Optimize the compilation, deployment, and testing of software applications
- Solve architectural problems with proven design patterns and advanced language features
- Maximize software performance
- Improve the reliability of threaded applications
- Customize application behavior with scripting
- Secure sensitive data and authenticate users with JAAS

---

**Notes:**

<ipf>L,4: Course Summary</ipf>
Appendix A

Self-Study: Servlets

Notes:
<ipf>B,1: Self-Study: Servlets </ipf>
Objectives

- You will learn how to
  - Implement and deploy a Java Web application using servlets
  - Test servlets

Notes:

<i>Chapter Objectives</i>
Self-Study: Servlets

 Servlets

 Testing Servlets

Notes:
<ipf>R,3: Servlets </ipf>
What Is a Servlet?

- A Servlet is a server-side Java component

![Servlet Diagram]

- Servlets run inside a servlet engine (such as Tomcat) embedded inside a Web server (such as Apache)
  - They are Java classes
  - But you have to tell the servlet engine which URL maps to which servlet
  - Servlets print out to the response object
    - In formats that Web browsers understand

Notes:

<ipf>L,4: What Is a Servlet? </ipf>

Ask students whether they are familiar with servlets. Spend less/more time on this as needed.
A Simple Servlet

- Extend `javax.servlet.http.HttpServlet` (imports not shown)

```java
public class SimpleServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        resp.setContentType("text/xml");
        PrintWriter out = resp.getWriter();
        out.println("<promotions>");
        out.println(" <promotion name='JazzInJune'/> ");
        out.println(" <promotion name='ChristmasInJuly'/> ");
        out.println("</promotions>");
        out.flush();
    }
}
```

1. Override the `doGet()` or `doPost()` methods
2. Set content type based on what you are going to be sending back
3. Get writer from response object
4. Print to it

**Notes:**

R.5: A Simple Servlet

May want to mention up front here that sample code typically shows only snippets; imports, etc. are not shown.
web.xml

- This servlet needs to be reachable ultimately through a Web server
  - Need to specify what the URL is
    - http://crs516.learningtree.com/MyApp/simple
  - The machine name comes from the Web server/DNS mappings
  - The context root (MyApp) comes from the .war file (next slide)

- The servlet mapping is specified in a file called web.xml

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<web-app>
  <servlet>
    <servlet-name>SimpleServlet</servlet-name>
    <servlet-class>com.ltree.crs516.SimpleServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>SimpleServlet</servlet-name>
    <url-pattern>/simple</url-pattern>
  </servlet-mapping>
</web-app>
```

Notes:

Even folks who have programmed servlets may not be aware of how to deploy servlets because someone in their project wrote the ant build.xml

Emphasize that there is a lot of stuff to do between programming the servlet and testing it out.
Deploying the Servlet

- Place a Web archive in the servlet engine’s autodeploy directory
  - To reach as: http://crs516.learningtree.com/MyApp/simple
  - Name the Web archive file MyApp.war

- Create this directory structure:
  - MyApp
    - META-INF
      - MANIFEST.MF
    - WEB-INF
      - classes
        - All the .class files that SimpleServlet needs
        - Any resource files needed by your project classes
    - lib
      - All the .jar files that SimpleServlet needs
    - web.xml
      - Static html pages and jsp files

- Then, create the .war file using: jar cvf MyApp.war MyApp

Notes:
<ipf>R,7: Deploying the Servlet</ipf>
Self-Study: Servlets

Servlets

Testing Servlets

Notes:
<ipf>L,8: Servlets </ipf>
• Many tools exist for testing Web applications
  — HttpUnit
    – Functional testing using Java API
    – Used to test servlets *when developing*
  — PageUnit
    – Functional testing using scripting language
    – Used to test Web pages in *deployed* environment
  — Cactus
    – Unit testing of servlets
    – No need to use Cactus if most application logic is *called* from servlet
      – And not embedded in the servlet itself
Functionality Testing

- **Used to test use cases**
  - Test behavior invoked by user interaction
    - From a servlet/JSP
    - From a Swing application
  - Not isolated methods of a class
    - Job of a unit test

- **A functionality test ensures that**
  - The individual classes in a use case work together correctly
  - A user will obtain the expected results

---

**Notes:**

<ipf>L,10: Functionality Testing</ipf>

We’ll resurrect Swing in the threading section.
Testing a Servlet

Do not test servlets by simply visiting a Web page
  — Visually examining results is very subjective
  — Need automated way to ensure that pages conform to requirements

• HttpUnit simplifies functional testing of Web applications
  — Write servlets and web.xml as usual
  — Use JUnit framework to test application behavior

• To use HttpUnit, create a JUnit test case:
  1. Create a mock Web conversation using HttpUnit
  2. Specify request parameters
  3. Set session variables if required
  4. Invoke servlet with specified request and obtain response
  5. Perform assertions on the response

Notes:
<ipf>R,11: Testing a Servlet</ipf>
Testing With HttpUnit

• HttpUnit allows you to parse the returned response
  — As text
  — As XML—response.getDOM()

```java
public void testGet() throws Exception {
    // Step 1
    ServletRunner sr = new ServletRunner(new File("web.xml");
    ServletUnitClient client = sr.newClient();
    WebRequest req =
        new GetMethodWebRequest("http://localhost/PromotionAnalyzer");
    // Step 2
    req.setParameter("retailer_id", "2");
    // Step 4
    WebResponse response = client.getResponse(req);
    String txt = response.getText();
    // Step 5
    assertTrue(txt.contains("PromotionAnalyzer results");
    assertTrue(txt.contains("Final score");
}
```

Notes:

<L,12: Testing With HttpUnit </L>

There is a corresponding PostMethodWebRequest also, but no general-purpose ServiceWebRequest
HttpUnit

- **Open the Eclipse project** dm21_httpunit
  - Examine the servlet code (**CurrentPromotionsServlet**)
  - Examine the HttpUnit code that tests the servlet
    - **CurrentPromotionsOutsideContainerTest**
  - Run the Ant test task

- **Does the task launch Tomcat?**
  - How is the servlet tested?

---

**Notes:**

`<ipf>R,13: HttpUnit</ipf>`

XXX fix the code to match the slide
Out-of-Container Test

- **HttpUnit does not perform the test within Tomcat**
  - It tests only the functionality of the servlet
    - Outside of the servlet container
  - For better repeatability
    - Can do test with different parameters
      - Ensure that results are what is required
      - For example, that a request without a `retailer_id` parameter assumes `retailer_id=0`

Notes:

<ipf>L,14: Out-of-Container Test </ipf>
In-Container Functionality Testing With PageUnit

- HttpUnit uses mock objects to test functionality
  - Tests are out-of-container
  - What if you need to test functionality, but within container?
    - To make sure deployed Web application is correct?

- Use PageUnit to test functionality of deployed application

```plaintext
P http://www.example.com/index.html
F login
R username Fred
R passwd ${PASSWD}
S
M Welcome.*Acme
```

- Get page, find a form named "login", set request parameters, submit, match regular expression
- No Java coding, just a script

Notes:

<ipf>R,15: In-Container Functionality Testing With PageUnit </ipf>
Cactus

- Cactus provides a framework to perform in-container tests
  - An open-source project from Apache
  - Extends JUnit, uses HttpUnit
  - Provides redirector servlets that run in the container
    - And execute test code within the servlet engine

http://cactus.apache.org/

Notes:

<i>Small</i> L,16: Cactus

Put alongside next slide

Cactus provides its own WebResponse object but it doesn’t have the power of HttpUnit’s one, so that’s preferred.
Writing a Cactus Unit Test

- To write a Cactus unit test for servlets:
  1. Extend ServletTestCase
     - Also JspTestCase and FilterTestCase
  2. Create a beginXYZ(WebRequest)
     - Add request parameters
  3. Create a testXYZ() method
     - Create servlet instance
     - Call the xyz() method of servlet
  4. Create a endXYZ(WebResponse)
     - Can use HttpUnit’s WebResponse object
     - Test response

Notes:

*R,17: Writing a Cactus Unit Test*

Put alongside next slide

Cactus provides its own WebResponse object but it doesn’t have the power of HttpUnit’s one, so that’s preferred.
Cactus Example Unit Test

// Step 1: extend ServletTestCase
public class CurrentPromotionsInContainerTest extends ServletTestCase {
   // Step 2
   public void beginGet(WebRequest request) {
      request.addParameter("retailer_id", "5");
   }
   // Step 3
   public void testGet() throws Exception {
      CurrentPromotionsServlet servlet = new CurrentPromotionsServlet();
      servlet.doGet(request, response);
   }
   // Step 4
   public void endGet(WebResponse resp) throws Exception {
      Document doc = XMLUtils.parse(resp.getInputStream());
      assertEquals(doc.getDocumentElement().getTagName(), "promotions");
   }
}

Notes:

<ipf>L,18: Cactus Example Unit Test</ipf>
In-Container vs. Out-of-Container

- Cactus does unit testing of servlets in container
  - HttpUnit does everything that Cactus does, but out of container
    - Unless a container resource is required, use HttpUnit, which is simpler
  - Cactus allows you to check that a servlet forward is working as intended
    - HttpUnit allows testing of the final response

- Cactus is not a great way to do a performance test
  - Even though it works in-container
  - It launches its own custom Tomcat tasks
  - Do not do performance testing on developer workstations

- Keep unit/functionality tests separate from performance testing
  - Unit/functionality tests are performed on the developer’s machine
  - Performance testing is done in the deployed environment
  - Don’t mix the two tests

Notes:

<ipf>L,19: In-Container vs. Out-of-Container </ipf>

Unit testing should be outside the container
Performance testing should be in-container, but should be of a usecase, not of single components.
How does it matter which JSP provides the response? What matters is the response … this is what HttpUnit tests.
Point out functionality testing outside the complexities of a deployed environment can be useful for other reasons too.
Summary

- You have learned how to
  — Implement and deploy a Java Web application using servlets
  — Test servlets

Notes:
R,20: Summary
Appendix B

Self-Study: Leveraging the Operating System With NIO and JNI

Notes:
B,1: Self-Study: Leveraging the Operating System With NIO and JNI
Objectives

You will learn how to

- Leverage the operating system’s capabilities using
  - New Input/Output (NIO)
  - Java Native Interface (JNI)

Notes:

<ipf>L,2: Chapter Objectives</ipf>
Self-Study: Leveraging the Operating System With NIO and JNI

Notes:

L,3: Self-Study: Leveraging the Operating System With NIO and JNI
Java NIO

• The Java New Input Output APIs (Java NIO)
  — Since Java 1.4
  — Bring some scalable I/O operations to Java

• NIO brings performance improvements in specific situations
  — Operating on data from large (gigabyte-size) files
  — Read from multiple sockets simultaneously
  — Operate at the binary data level
    – To transfer files, but not to parse them
    – Or to read into an array of primitives

• NIO is meant to supplement regular I/O
  Most Java programs should still use java.io classes
  — The java.io classes have been modified to support NIO operations

Notes:
<ipf>R,4: Java NIO</ipf>
Reading Large Files

- Memory mapping improves performance when operating on very large files
  - Operating systems have support for memory mapping the files
  - Random access in memory is faster than doing seeks on files
  - NIO has support for memory mapping a file

```java
FileChannel fc = new FileInputStream(file).getChannel();
MappedByteBuffer bytes =
    fc.map(FileChannel.MapMode.READ_ONLY, 0, fc.size());
```

Notes:

<ipf>L.5: Reading Large Files </ipf>
Reading Data from Byte Buffer

- Wrap up the byte buffer and treat it as an array of primitives

```java
DoubleBuffer buf = DoubleBuffer.asDoubleBuffer(bytes);
```

- Or in a character array if the encoding is known

```java
CharsetDecoder decoder =
    Charset.forName("ISO-8859-15").newDecoder();
CharSequence chars = decoder.decode(bytes);
```

Notes:

<i>Reading Data from Byte Buffer</i>
Regular Expressions

- Regular expressions are a powerful way to parse and operate on text
  - ^[0-1][0-9]/[0-3][0-9]/200[5|6]\w+20:\d\d:\d\d UTC
  - Matches lines of a time-stamped log file that were written between 8 p.m. and 9 p.m. UTC
  - In combination with NIO, you can avoid creating a lot of temporary Strings when combing through large log files like this

Notes:

<ipf>R,7: Regular Expressions </ipf>
Regular Expression Parsing

- The regular expressions API can be used independently of NIO
  - The Pattern classes will operate on any CharSequence
  - CharBuffer is a CharSequence, but so is String

```java
public CharSequence replace(CharSequence chars, String regex, String bythis)
    throws IOException
{
    Pattern pattern = Pattern.compile(regex);
    Matcher match = pattern.matcher(chars);
    return match.replaceAll(bythis);
}
```

Notes:

<ipf>R,8: Regular Expression Parsing </ipf>
Reading in Pieces

- **NIO allows reading a file in pieces**
  - Useful for doing binary transfers

```java
ByteBuffer bytes = ByteBuffer.allocateDirect(1024);
CharBuffer chars = CharBuffer.allocate(1024);
while (fc.read(bytes) != -1) {// from FileChannel to buffer
    bytes.flip(); // changes it to read-mode
    decoder.decode(bytes, chars, false); // decoder from two slides back
    chars.flip(); // changes it to read-mode
    // do something with the chars buffer, perhaps transfer it somewhere
    bytes.clear(); // set the read/write pointer back to beginning
    chars.clear(); // set the read/write pointer back to beginning
}
```

---

**Notes:**

<i>Reading in Pieces</i>
Reading From Sockets

• To read from sockets, use a socket channel

```java
SocketChannel sc = SocketChannel.open();
sc.connect( new InetSocketAddress(host, port) );
```

• With NIO, socket reads can be set not to block
  — The program should poll the socket to find out if there is something to read
  — File reads are still only the blocking kind
  — In java.io, every read from a socket blocks until there is data to read
    – If you have several sockets to read, you need a thread for each socket
    – Makes for many threads when writing a high-performance server

Notes:
<ipf>L,10: Reading From Sockets </ipf>
Non-Blocking Reads

- To register a `Selector` with a bunch of socket channels
  - Use a `Selector` to poll the sockets
  - A `FileChannel` is not selectable

```java
Selector selector = Selector.open();
// sc is an SocketChannel[]
for (int i=0; i < sc.length; ++i){
    sc[i].configureBlocking(false); // sc[i] is a SocketChannel
    sc[i].register( selector, SelectionKey.OP_READ );
}
```

**Notes:**

<ipf>R,11: Non-Blocking Reads </ipf>

The fact that `FileChannel` is not selectable is referred to in the JNI section, so make sure to mention it here.
Performing the Select

- The selector’s select method will return all of the channels that have activity
  - Loop through them
  - Read the data
  - All in one thread!

```java
while (selector.select(timeout) > 0){
    Set<SelectionKey> keys = selector.selectedKeys();
    Iterator<SelectionKey> iter = keys.iterator();
    while (iter.hasNext()){
        // remove the channel you are processing
        SelectionKey key = iter.next();
        iter.remove();
        SocketChannel sc = key.channel();
        // read from socket
    }
}
```

- Need to remove the channel being processing inside the select loop
  - The list of channels can change while you are reading it
  - Why is the NIO API not very intuitive?

Notes:

<ipf>L,12: Performing the Select </ipf>

APIs designed for optimality are rarely easy to use … refer back to the Ex4.2 API for the Iterator …
NIO

JNI

Notes:

<ipf>L,13: Self-Study: Leveraging the Operating System With NIO and JNI <ipf>
What Is JNI?

- The Java Native Interface (JNI) allows you to invoke C code from Java
  - JNI rarely improves performance
  - Lots of high-performance, visualization software now written in Java

Notes:

<ipf>L,14: What Is JNI? </ipf>
XXX: break into two slides
Where Is JNI Useful?

- **Use JNI to**
  - Invoke legacy code in C, C++, or Fortran
  - Call to native Operating System (OS) functionality
  - Not portable, but useful for functionality that only OS can provide

- **For example, on Linux, there is a mechanism called inotify**
  - Can select and wait on a directory or file
    - Just as NIO provides ability to select and wait on a socket
    - Notified whenever the file or directory is written to or read from
      - No need to poll the directory, list its files, or look for changes!
  - Can drastically improve efficiency
  - inotify is a UNIX device; it has no counterpart on Windows
    - Useful when you know your server environment is Linux
  - You need to access it via a C interface

**Notes:**

<ipf>L,15: Where Is JNI Useful? </ipf>
XXX: break into two slides
How to Write a Class With a Native Method

```java
public class JNIFileChangeMonitor {
    static {
        System.loadLibrary("fcmonitor"); // Step 5
    }
    private native int hasChanged(String filename); // Step 1
}
```

```xml
<javac destdir="build">
    <src path="src"/>
    <classpath refid="project.classpath"/>
</javac>

<javah class="com.ltree.JNIFileChangeMonitor"
    outputfile="src_c/jni_filechange_monitor.h">
    <classpath refid="project.classpath"/>
</javah>
```

Notes:

R,16: How to Write a Class With a Native Method

3,4 are C and OS-specific
How to Write a Class With a Native Method (continued)

1. Mark any methods that will be implemented in C as native
2. Compile the class using javac and javah
3. javah creates a C header file: Write its implementation in C (or C++)
4. Create a shared library (e.g, fcmonitor.dll or libfcmonitor.so)
5. Load shared library: Needs to be in path OS searches for libraries
6. Use like a regular Java object
   — Call native method like any instance method of object

Notes:

<ipf>R,17: How to Write a Class With a Native Method (continued) </ipf>

3,4 are C and OS-specific
Example JNI and NIO Application

• Open the Eclipse project dm41_niojni
  — This illustrates several operations using NIO
    – Counting the number of words in a file
    – Replacing words in a file and writing out a new file
  — The word count operation is also illustrated using JNI

• The Ant “test” task will run a set of timing tests
  — On Shakespeare’s *The Merchant of Venice*
    – The file is only 130 KB
    Does NIO seem to help?
    How about JNI?
    What does this tell you about the speed of the JDK core classes?

Notes:

<i>IP</i>L,18: Example JNI and NIO Application</i>

NIO is good for pure, binary transfers of data. As soon as you need to “peek” inside the data to do something,

Java I/O is just as fast.
Summary

You have learned how to

- Leverage the operating system’s capabilities using
  - New Input/Output (NIO)
  - Java Native Interface (JNI)

Notes:

<ipf>R,19: Summary</ipf>
Appendix C

Self-Study: Java Cryptography APIs and Tools

Notes:

<ipf>B,1: Self-Study: Java Cryptography APIs and Tools </ipf>
Objectives

You will learn how to

- Secure sensitive data with Java cryptography APIs and tools

Notes:

<ipf>L.2: Chapter Objectives</ipf>

A quick chapter presenting concepts that we will build on throughout the course.
Cryptography

- **Science of “hiding” messages**
  - Plaintext can be recovered from **encrypted text** only with the right **key**

- **Cryptography can be used for one or both of the following:**
  - Keeping data confidential
  - Ensuring that data has not been modified in transit

**Incorporate encryption into applications if data must be safeguarded**
- The data may need to travel through public channels
  - Most common scenario for encryption
- Sensitive data should be encrypted even if it always resides on server
  - Authentication and authorization only restrict access to server
  - If interloper gains access, such data may need to be protected, too

---

**Notes:**

<ipf>R.3: Cryptography</ipf>
Symmetric Encryption

- Symmetric encryption algorithms have only one SecretKey
  - Encryption and decryption are achieved with the same key
  - Example: Advanced Encryption Standard (AES)
  - Fast, but key exchange is a problem

Notes:

<ipf>L.4: Symmetric Encryption</ipf>
Symmetric Encryption: Java Code

```java
SecretKey secretKey = ...;
javax.crypto.Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, secretKey);
byte[] codedText = cipher.doFinal(plainText.getBytes());
```

Notes:

<ipf>L.5: Symmetric Encryption: Java Code </ipf>
Generating a Good Secret Key

• A good SecretKey can be generated using:

```
SecretKey secretKey = ...;
javax.crypto.Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, secretKey);
byte[] codedText = cipher.doFinal(plainText.getBytes());
```

— Or, using the genkey option to keytool, a Java 5 SDK tool

```
SecretKey secretKey = KeyGenerator.getInstance("AES").generateKey();
```

Notes:

<ipf>L,6: Generating a Good Secret Key </ipf>
Asymmetric Encryption

- Asymmetric encryption algorithms have a **KeyPair**
  - Encryption is done with one key (PublicKey or PrivateKey)
  - Decryption is done with the other key
  - Also called public key/private key methods
  - Example: RSA

Notes:

<i>IPF>L,7: Asymmetric Encryption</i>
**Uses of Asymmetric Encryption**

- **Anyone can send you confidential information using your public key**
  - Encrypt with public key → Send encrypted text → Decrypt with private key

- **You can “sign” a message with your private key**
  - Encrypt with private key → Send signed message → Decrypt with public key

**Notes:**

<ipf>R,8: Uses of Asymmetric Encryption</ipf>
Asymmetric Encryption in Java

- Use a Cipher to perform the actual encryption and decryption

```java
cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.DECRYPT_MODE, privateKey);
byte[] plainText = cipher.doFinal(codedText.getBytes());
```

- A good KeyPair can be generated using KeyPair generator
  — However, the public key must be published or transmitted
  — Use keystores and certificates to store and exchange keys
  — Using the keytool that is part of the Java 5 SDK

Notes:

<ipf>L,9: Asymmetric Encryption in Java </ipf>
Key Management Steps

- **Steps to generate, exchange, and extract keys**
  1. Generate public/private key pair and store in keystore
  2. Get public key signed by certificate authority, or self-sign
  3. Export public key as certificate
  4. Recipient imports public key into his/her keystore
  5. Get your private key or recipient’s public key from keystore
     — Encrypt data using Cipher and key (above code)

---

Notes:

<ipf>L,10: Key Management Steps</ipf>
Steps 1 and 2: Generating Authenticated Keys

1. Create a public/private key pair, creating the keystore if needed

   keytool -genkey -dname datamining.rainforest.com -keypass rfpassword \
   -storepass rfpassword -keyalg RSA -keystore ${HOME}/.keystore \
   -alias rainforest

   — dname is the “distinguished name”
   — It is typical to provide domain name
   — Alias is the name by which the private key may later be retrieved

2. Get certificate signed

   — Typically by a trusted third party called a Certificate Authority (CA)
   — Such as Thawte or VeriSign
   — Follow instructions from CA on creating certificate
   — Can also simply self-sign certificate if recipient will acknowledge

   keytool -selfcert -keypass rfpassword \
   -storepass rfpassword -keystore ${HOME}/.keystore \
   -alias rainforest

Notes:
<ipf>R,11: Steps 1 and 2: Generating Authenticated Keys </ipf>
Steps 3 and 4: Exchanging Keys

3. Export public key as a certificate

```
keytool -export \
  -storepass rfpassword -keystore ${HOME}/.keystore \
  -alias rainforest -file rainforest.cert
```

— Certificate is written to rainforest.cert

4. Recipient imports public key certificate into his or her KeyStore

```
keytool -import \
  -storepass mampassword -keystore C:\keystore \ 
  -alias rainforest -file rainforest.cert
```

— Typically returns favor by sending his or her public key certificate
— To incorporate into your KeyStore

**Notes:**

<ipf>L,12: Steps 3 and 4: Exchanging Keys</ipf>
Step 5: Keys Out of a KeyStore

5. a. To obtain a PublicKey from a KeyStore

```java
KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());
InputStream is = new FileInputStream("C:/keystore");
keyStore.load(is, mampasswordArray);
is.close();
Key publicKey = keyStore.getCertificate("rainforest").getPublicKey();
```

5. b. To obtain a PrivateKey from a KeyStore

```java
KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());
InputStream is = new FileInputStream("C:/keystore");
keyStore.load(is, rfpasswordArray);
is.close();
Key privateKey = keyStore.getKey("rainforest", rfpasswordArray);
```

For safety, do not store passwords in your program as strings—strings are interned; use character arrays instead.

Notes:

<R,13: Step 5: Keys Out of a KeyStore</R>
**Message Digest**

- **A message digest is a one-way hash function**
  - Can be used to detect that messages are the same
  - Example: Secure Hash Algorithm (SHA1)

- **Extremely hard to get original message from the hashed value**
  - Can be used to avoid saving passwords in clear text
  - But still possible to do verification

- **To create a MessageDigest**

  ```java
  java.security.MessageDigest md = MessageDigest.getInstance("SHA1");
  md.update( dataBytes ); // call as often as needed
  byte[] digest = md.digest();
  ```

- **MessageDigest has a convenient method to check if two digests are equal**

  ```java
  boolean isSame = MessageDigest.isEqual( digest1, digest2 );
  ```

*Notes:*

<ipf>L,14: Message Digest</ipf>
## Signing Code

- **The `security.policy` file permits granting permissions to code**
  - In addition to or instead of principals

```java
keystore file:/c:/mam.keystore;
grant codebase "file:rainforest.jar" signedBy "rainforest" {
  permission java.net.SocketPermission "promos.rf.com:8080", "connect, read";
};
```

- The keystore should contain a certificate (with public key) of RainForest

- **RainForest should have signed the `.jar` file with its private key**

```bash
jarsigner -keystore rainforest.keystore rainforest.jar rainforest
```

- jarsigner is a free tool that is part of the J2SDK version 5
- The keystore should contain the private key with the alias `rainforest`

### Notes:

<ipf>R,15: Signing Code</ipf>
An Encryption Scenario

- Consider the case of RainForest running store promotions
  1. Sends coupons to customers of that store
  2. Customers visit store and present coupon
     - Retailer verifies coupon
     - Retailer gives customer a discounted price
     - Retailer bills RainForest for the discount
  3. RainForest validates purchase and refunds retailer

Notes:

<ipf>L,16: An Encryption Scenario</ipf>
Step 1: Non-Repudiability

- When RainForest sends a store coupon to customers
  - It needs to sign coupon so that retailers can trust that they will get paid
  - Coupon is non-repudiable

Notes:

<ipf>L,17: Step 1: Non-Repudiability </ipf>

Make sure that you have a floppy or thumb-drive ready. Try not to use a network file transfer as the point of the exercise might get lost if it’s just a bunch of mouse clicks.
Steps 2: Confidentiality

- **When customer presents coupon**
  - Retailer verifies that the coupon is indeed from RainForest
  - Checks signature

- **When Retailer sends the discount to RainForest**
  - Needs to ensure that its sales information is not snooped on by competitors
  - Sales information is *confidential*, and can be read only by RainForest

**Notes:**

<i>ipf>L,18: Steps 2: Confidentiality </i>/ipf>

Make sure that you have a floppy or thumb-drive ready. Try not to use a network file transfer as the point of
the exercise might get lost if it’s just a bunch of mouse clicks.
Step 4: Validation

- When RainForest receives purchase
  — It validates the purchase and refunds retailer

Notes:

<i>®</i>R,19: Step 4: Validation</i>
An Encryption Scenario

- **Examine the code in dm71_keystore**
  - Demonstrates the steps of key exchange with keytool
  - Java code for coupon/purchase scenario just discussed

---

**Notes:**

[ipf]L,20: An Encryption Scenario[/ipf]
Summary

You have learned how to

- Secure sensitive data with Java cryptography APIs and tools