

Java Programming Language, Java SE 6

Student Guide

SL-275-SE6 G.2

D61748GC11

Edition 1.1

June 2010

D67981

ORACLE®

Copyright © 2009, 2010, Oracle and/or its affiliates. All rights reserved.

Disclaimer

This document contains proprietary information, is provided under a license agreement containing restrictions on use and disclosure, and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except as expressly permitted in your license agreement or allowed by law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

Sun Microsystems, Inc. Disclaimer

This training manual may include references to materials, offerings, or products that were previously offered by Sun Microsystems, Inc. Certain materials, offerings, services, or products may no longer be offered or provided. Oracle and its affiliates cannot be held responsible for any such references should they appear in the text provided.

Restricted Rights Notice

If this documentation is delivered to the U.S. Government or anyone using the documentation on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS

The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. UNIX is a registered trademark licensed through X/Open Company, Ltd.

This page intentionally left blank.

Mayur Patel (mayurp391@gmail.com) has a non-transferable license
to use this Student Guide.

This page intentionally left blank.

Mayur Patel (mayurp391@gmail.com) has a non-transferable license
to use this Student Guide.

Table of Contents

About This Course	Preface-xv
Course Goals.....	Preface-xv
Course Overview	Preface-xvii
Course Map.....	Preface-xviii
Module-by-Module Overview	Preface-xix
Course Objectives.....	Preface-xxi
Topics Not Covered	Preface-xxii
How Prepared Are You?.....	Preface-xxiii
Introductions	Preface-xxiv
How to Use Course Materials	Preface-xxv
Typographical Conventions and Symbols	Preface-xxvi
Icons	Preface-xxvi
Typographical Conventions	Preface-xxvii
Additional Conventions.....	Preface-xxviii
Getting Started.....	1-1
Objectives	1-1
Relevance.....	1-2
Additional Resources	1-3
What Is the Java™ Technology?	1-4
Primary Goals of the Java Technology	1-5
The Java Virtual Machine	1-6
Garbage Collection	1-7
The Java Runtime Environment.....	1-8
JVM™ Tasks.....	1-10
The Class Loader.....	1-10
The Bytecode Verifier	1-11
A Simple Java Application.....	1-12
The TestGreeting Application	1-13
The Greeting Class.....	1-15
Compiling and Running the TestGreeting Program.....	1-16
Troubleshooting the Compilation	1-17

Object-Oriented Programming	2-1
Objectives	2-1
Relevance.....	2-2
Software Engineering	2-3
The Analysis and Design Phase.....	2-5
Analysis and Design Example	2-6
Abstraction.....	2-7
Classes as Blueprints for Objects	2-7
Declaring Java Technology Classes.....	2-8
Declaring Attributes	2-9
Declaring Methods.....	2-10
Example	2-11
Accessing Object Members.....	2-12
Information Hiding.....	2-13
Encapsulation	2-15
Declaring Constructors	2-16
The Default Constructor	2-17
Source File Layout	2-18
Software Packages	2-19
The package Statement.....	2-20
The import Statement.....	2-21
Directory Layout and Packages	2-23
Development	2-24
Compiling Using the -d Option	2-24
Deployment	2-25
Terminology Recap.....	2-26
Using the Java Technology API Documentation.....	2-27
Identifiers, Keywords, and Types	3-1
Objectives	3-1
Relevance.....	3-2
Comments	3-3
Semicolons, Blocks, and White Space	3-4
Identifiers	3-6
Java Programming Language Keywords	3-7
Basic Java Programming Language Types.....	3-8
Primitive Types	3-8
Logical – boolean	3-8
Textual – char	3-9
Textual – String.....	3-9
Integral – byte, short, int, and long.....	3-10
Floating Point – float and double.....	3-11
Variables, Declarations, and Assignments.....	3-12
Java Reference Types.....	3-13

Constructing and Initializing Objects	3-14
Memory Allocation and Layout.....	3-15
Explicit Attribute Initialization	3-15
Executing the Constructor	3-16
Assigning a Variable.....	3-16
This Is Not the Whole Story	3-16
Assigning References.....	3-17
Pass-by-Value	3-18
The this Reference.....	3-20
Java Programming Language Coding Conventions	3-21
Expressions and Flow Control.....	4-1
Objectives	4-1
Relevance.....	4-2
Variables.....	4-3
Variables and Scope.....	4-3
Variable Initialization.....	4-5
Initialization Before Use Principle	4-6
Operators.....	4-7
Operator Precedence	4-7
Logical Operators.....	4-8
Bitwise Logical Operators.....	4-9
Right-Shift Operators >> and >>>.....	4-10
Left-Shift Operator <<	4-10
Shift Operator Examples	4-11
String Concatenation With +	4-12
Casting.....	4-12
Promotion and Casting of Expressions.....	4-13
Branching Statements.....	4-14
Simple if, else Statements.....	4-14
Complex if, else Statements.....	4-14
The switch Statement	4-16
Looping Statements	4-18
The for Loops	4-18
The while Loop.....	4-18
The do/while Loop	4-19
Special Loop Flow Control	4-19
Arrays	5-1
Objectives	5-1
Relevance.....	5-2
Declaring Arrays	5-3
Creating Arrays.....	5-4
Creating Reference Arrays.....	5-5
Initializing Arrays	5-6
Multidimensional Arrays.....	5-7
Array Bounds	5-8
Using the Enhanced for Loop	5-8
Array Resizing.....	5-9
Copying Arrays.....	5-10

Class Design	6-1
Objectives	6-1
Relevance.....	6-2
Subclassing.....	6-3
Single Inheritance.....	6-4
Access Control.....	6-6
Overriding Methods	6-7
Overridden Methods Cannot Be Less Accessible.....	6-8
Invoking Overridden Methods	6-9
Polymorphism	6-10
Virtual Method Invocation	6-10
Heterogeneous Collections.....	6-11
Polymorphic Arguments	6-12
The instanceof Operator.....	6-13
Casting Objects	6-14
Overloading Methods	6-15
Methods Using Variable Arguments	6-16
Overloading Constructors	6-17
Constructors Are Not Inherited.....	6-18
Invoking Parent Class Constructors.....	6-18
Constructing and Initializing Objects: A Slight Reprise.....	6-20
Constructor and Initialization Examples.....	6-20
The Object Class	6-22
The equals Method.....	6-22
The toString Method	6-24
Wrapper Classes.....	6-26
Autoboxing of Primitive Types.....	6-27
Advanced Class Features	7-1
Objectives	7-1
Relevance.....	7-2
The static Keyword	7-3
Class Attributes	7-3
Class Methods	7-4
Static Initializers	7-6
The final Keyword	7-7
Final Classes.....	7-7
Final Methods	7-7
Final Variables	7-7
Enumerated Types	7-9
Old-Style Enumerated Type Idiom	7-10
The New Enumerated Type	7-13
Advanced Enumerated Types.....	7-16
Static Imports.....	7-18
Abstract Classes.....	7-19
The Problem.....	7-20
The Solution	7-21

Interfaces	7-23
The Flyer Example	7-24
Multiple Interface Example	7-28
Uses of Interfaces	7-29
Exceptions and Assertions	8-1
Objectives	8-1
Relevance.....	8-2
Exceptions and Assertions.....	8-3
Exceptions	8-4
Exception Example	8-5
The try-catch Statement	8-6
Call Stack Mechanism	8-8
The finally Clause	8-9
Exception Categories	8-10
Common Exceptions.....	8-12
The Handle or Declare Rule	8-13
Method Overriding and Exceptions.....	8-15
Creating Your Own Exceptions	8-16
Throwing a User-Defined Exception	8-17
Handling a User-Defined Exception	8-18
Assertions.....	8-19
Recommended Uses of Assertions	8-19
Controlling Runtime Evaluation of Assertions	8-22
Collections and Generics Framework	9-1
Objectives	9-1
Additional Resources	9-2
The Collections API	9-3
Collection Implementations	9-5
A Set Example	9-5
A List Example	9-6
The Map Interface.....	9-8
A Map example.....	9-9
Legacy Collection Classes.....	9-10
Ordering Collections	9-11
The Comparable Interface	9-11
The Comparator Interface	9-13
Generics	9-17
Generic Set Example.....	9-18
Generic Map Example	9-19
Generics: Examining Type Parameters	9-20
Wild Card Type Parameters	9-22
Generics: Refactoring Existing Non-Generic Code	9-25
Iterators.....	9-27
The Enhanced for Loop	9-29

I/O Fundamentals.....	10-1
Objectives	10-1
Additional Resources	10-2
Command-Line Arguments	10-3
System Properties.....	10-4
The Properties Class	10-5
I/O Stream Fundamentals.....	10-6
Data Within Streams.....	10-6
Byte Streams	10-7
The InputStream Methods	10-7
The OutputStream Methods.....	10-8
Character Streams.....	10-9
The Reader Methods.....	10-9
The Writer Methods.....	10-10
Node Streams.....	10-11
A Simple Example.....	10-11
Buffered Streams	10-13
I/O Stream Chaining.....	10-14
Processing Streams	10-15
Basic Byte Stream Classes	10-16
The FileInputStream and FileOutputStream Classes	10-17
The BufferedInputStream and BufferedOutputStream Classes	10-17
The PipedInputStream and PipedOutputStream Classes	10-17
The DataInputStream and DataOutputStream Classes	10-17
The ObjectInputStream and ObjectOutputStream Classes	10-18
Serialization	10-21
Serialization and Object Graphs	10-21
Writing and Reading an Object Stream	10-22
Basic Character Stream Classes.....	10-25
The InputStreamReader and OutputStreamWriter Methods	10-26
Byte and Character Conversions	10-26
Using Other Character Encoding	10-26
The FileReader and FileWriter Classes	10-27
The BufferedReader and BufferedWriter Classes	10-27
The StringReader and StringWriter Classes	10-27
The PipedReader and PipedWriter Classes.....	10-27

Console I/O and File I/O	11-1
Objectives	11-1
Additional Resources	11-2
Console I/O	11-3
Writing to Standard Output	11-3
Reading From Standard Input	11-4
Simple Formatted Output.....	11-6
Simple Formatted Input.....	11-7
Files and File I/O	11-8
Creating a New File Object	11-8
The File Tests and Utilities	11-9
File Stream I/O.....	11-10
Building Java GUIs Using the Swing API.....	12-1
Objectives	12-1
Additional Resources	12-2
Pluggable Look-and-Feel	12-4
Swing Architecture	12-5
Swing Packages	12-7
Examining the Composition of a Java Technology GUI	12-9
Swing Containers	12-11
Top-level Containers	12-11
Swing Components.....	12-12
The Swing Component Hierarchy.....	12-13
Properties of Swing Components	12-14
Common Component Properties.....	12-14
Component-Specific Properties	12-15
Layout Managers	12-16
The BorderLayout Layout Manager	12-16
The FlowLayout Layout Manager	12-18
The BoxLayout Layout Manager.....	12-19
The CardLayout Layout Manager	12-20
The GridLayout Layout Manager	12-21
The GridBagLayout Layout Manager.....	12-22
The GroupLayout Layout Manager	12-23
GUI Construction.....	12-24
Programmatic Construction	12-24
Handling GUI-Generated Events	13-1
Objectives	13-1
Additional Resources	13-2
What Is an Event?	13-3
Java SE Event Model.....	13-4
Delegation Model.....	13-4
A Listener Example.....	13-5
GUI Behavior	13-7
Event Categories	13-7
Complex Example.....	13-9

Multiple Listeners	13-12
Developing Event Listeners.....	13-13
Event Adapters.....	13-13
Event Handling Using Inner Classes	13-14
Event Handling Using Anonymous Classes.....	13-15
Concurrency in Swing.....	13-17
GUI-Based Applications	14-1
Objectives	14-1
Relevance.....	14-2
How to Create a Menu	14-3
Creating a JMenuBar.....	14-4
Creating a JMenu	14-5
Creating a JMenuItem	14-5
Creating a JCheckboxMenuItem.....	14-6
Controlling Visual Aspects.....	14-8
Colors.....	14-8
Threads	15-1
Objectives	15-1
Relevance.....	15-2
Threads	15-3
Creating the Thread.....	15-4
Starting the Thread	15-5
Thread Scheduling	15-6
Terminating a Thread.....	15-8
Basic Control of Threads.....	15-10
Testing Threads	15-10
Accessing Thread Priority	15-10
Putting Threads on Hold	15-10
Other Ways to Create Threads.....	15-13
Selecting a Way to Create Threads	15-14
Using the synchronized Keyword	15-15
The Problem.....	15-15
The Object Lock Flag	15-17
Releasing the Lock Flag.....	15-19
Using synchronized – Putting It Together.....	15-20
Thread States	15-21
Deadlock.....	15-21
Thread Interaction – wait and notify.....	15-22
Scenario	15-22
The Problem.....	15-22
The Solution.....	15-22
Thread Interaction.....	15-23
The wait and notify Methods	15-23
Thread States	15-24
Monitor Model for Synchronization	15-25

Putting It Together	15-26
The Producer Thread	15-27
The Consumer Thread	15-28
The SyncStack Class.....	15-29
The SyncTest Example.....	15-32
Networking	16-1
Objectives	16-1
Relevance.....	16-2
Networking	16-3
Sockets	16-3
Setting Up the Connection.....	16-3
Addressing the Connection.....	16-5
Port Numbers	16-5
Java Networking Model.....	16-6
Minimal TCP/IP Server	16-7
Minimal TCP/IP Client.....	16-8
Elements of Advanced Java Programming.....	A-1
Objectives	A-1
Introduction to Two-Tier and Three-Tier Architectures	A-2
Three-Tier Architecture.....	A-3
Three-Tier Client-Server Definition	A-3
A Database Front End	A-4
Introduction to the JDBC™ API.....	A-6
JDBC, An Overview	A-6
JDBC Drivers.....	A-7
The JDBC-ODBC Bridge	A-7
Distributed Computing.....	A-8
RMI.....	A-9
RMI Architecture.....	A-9
Creating an RMI Application	A-10
CORBA	A-11
The Java IDL	A-11
RMI Compared With CORBA.....	A-12
The JavaBeans™ Component Model	A-14
Bean Architecture.....	A-14
Bean Introspection	A-16
A Sample Bean Interaction	A-17
The Beans Development Kit (BDK)	A-17
JAR Files	A-18
Using the javadoc Tool.....	A-19
Documentation Tags.....	A-19
Example	A-20
Quick Reference for UML.....	B-1
Objectives	B-1
Additional Resources	B-2
UML Fundamentals.....	B-3

General Elements	B-6
Packages	B-6
Stereotypes	B-8
Annotation	B-8
Constraints	B-9
Tagged Values	B-9
Use Case Diagrams	B-10
Class Diagrams	B-11
Class Nodes.....	B-11
Inheritance.....	B-14
Interface Implementation.....	B-15
Association, Roles, and Multiplicity	B-16
Aggregation and Composition	B-18
Association Classes	B-19
Other Association Elements	B-21
Object Diagrams	B-22
Collaboration Diagrams	B-24
Sequence Diagrams.....	B-26
Statechart Diagrams.....	B-28
Transitions.....	B-29
Activity Diagrams	B-30
Component Diagrams	B-34
Deployment Diagrams	B-36
Swing Components	C-1
Objectives	C-1
Swing Component Examples	C-2
Top Level Containers	C-3
General-purpose Containers	C-4
Special-Purpose Containers.....	C-6
Buttons.....	C-8
Text Components	C-10
Uneditable Information Display Components	C-12
Menus	C-13
Formatted Display Components.....	C-14
Other Basic Controls.....	C-16

Preface

About This Course

Course Goals

Upon completion of this course, you should be able to:

- Create Java™ technology applications that leverage the object-oriented features of the Java language, such as encapsulation, inheritance, and polymorphism
- Execute a Java technology application from the command line
- Use Java technology data types and expressions
- Use Java technology flow control constructs
- Use arrays and other data collections
- Implement error-handling techniques using exception handling
- Create an event-driven graphical user interface (GUI) by using Java technology GUI components: panels, buttons, labels, text fields, and text areas
- Implement input/output (I/O) functionality to read from and write to data and text files
- Create multithreaded programs
- Create a simple Transmission Control Protocol/Internet Protocol (TCP/IP) client that communicates through sockets

Course Goals

The main goal of the *Java™ Programming Language* course is to provide you with the knowledge and skills necessary for object-oriented programming of advanced Java applications. In this course, you learn Java programming language syntax and object-oriented concepts, as well as more sophisticated features of the Java runtime environment, such as support for GUIs, multithreading, and networking. This course covers prerequisite knowledge to help prepare you for the Sun Certified Programmer for the Java™ Platform (SCJP) examination. For information about the exam, review the Web site:

<http://www.sun.com/training/certification/java/>

However, SL-275 is not sufficient to immediately pass the exam. You should spend several months practicing these techniques by building real programs. You should also review the exam objectives and study areas that were not discussed in this course. The SCJP exam objectives can be found at the web site listed previously.

Course Overview

This course first describes the Java runtime environment and the syntax of the Java programming language. The course then covers object-oriented concepts as they apply to the Java programming language. As the course progresses, advanced features of the Java platform are discussed.

The audience for this course includes people who are familiar with implementing elementary programming concepts using the Java programming language or other languages. This is the follow-up course to SL-110: *Fundamentals of the Java™ Programming Language*.

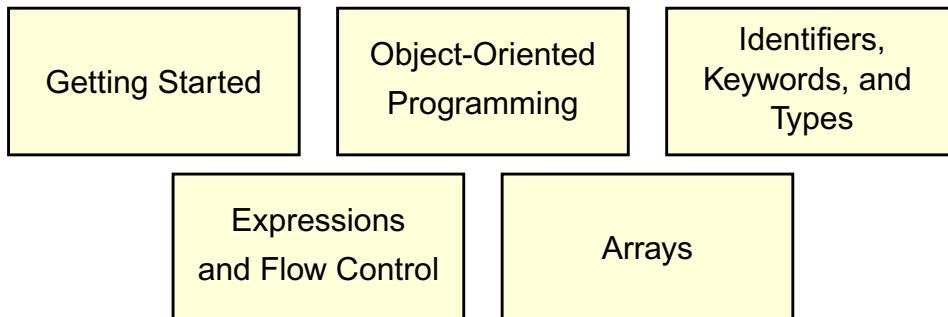
While the Java programming language is operating-system-independent, the GUI that it produces can be dependent on the operating system on which the code is executed. In this course, code examples are run in the Solaris™ Operating System (Solaris OS) and in the Microsoft Windows operating environment; therefore, the graphics in this guide have both a Motif and a Windows GUI. The content of this course is applicable to all Java operating system ports.

Course Map

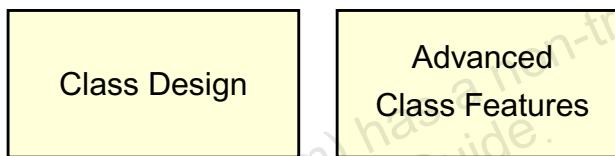
Course Map

The following course map enables you to see what you have accomplished and where you are going in reference to the course goal.

The Java Programming Language Basics



More Object-Oriented Programming



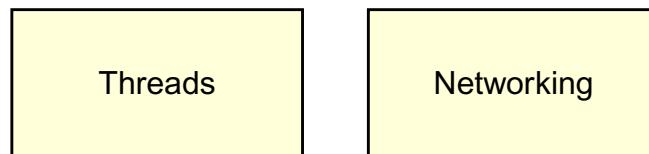
Exceptions, Collections, and I/O



Developing Graphical User Interfaces



Advanced Java Programming



Module-by-Module Overview

- **Module 1 – “Getting Started”**

This module provides a general overview of the Java programming language and its main features and introduces a simple Java application.

- **Module 2 – “Object-Oriented Programming”**

This module introduces basic object-oriented programming concepts and describes their implementation using the Java programming language.

- **Module 3 – “Identifiers, Keywords, and Types”**

The Java programming language contains many programming constructs similar to the C language. This module provides a general overview of the constructs available and the general syntax required for each construct. It also introduces the basic object-oriented approach to data association using aggregate data types.

- **Module 4 – “Expressions and Flow Control”**

This module looks at expressions, including operators, and the syntax of Java program control.

- **Module 5 – “Arrays”**

This module describes how Java arrays are declared, created, initialized, and copied.

- **Module 6 – “Class Design”**

This module builds on the information about object-oriented concepts provided in Module 2 and takes the information to the next level, including a discussion on subclassing, overloading, and overriding.

- **Module 7 – “Advanced Class Features”**

This module completes the Java object-oriented programming model by discussing the concepts of static members, final variables, abstract classes, and interfaces.

- **Module 8 – “Exceptions and Assertions”**

Exceptions provide you with a mechanism for trapping errors at runtime. This module explores both predefined and user-defined exceptions.

Module-by-Module Overview

- **Module 9 – “Collections and Generics Framework”**

This module examines the collections framework and the use of generics in the Java programming language.

- **Module 10 – “I/O Fundamentals”**

This module describes the classes available for reading and writing both data and text. It also contains a discussion on object serialization.

- **Module 11 – “Console I/O and File I/O”**

This module introduces topics that are useful in implementing large, text-based applications, such as console and file I/O.

- **Module 12 – “Building Java GUIs”**

All graphical user interfaces in the Java programming language are built on the concept of frames and panels. This module introduces layout management and containers using Swing.

- **Module 13 – “GUI Event Handling”**

Creating a layout of GUI components in a frame is not enough. You must write code to handle the events that occur, such as clicking a button or typing a character. This module demonstrates how to write GUI event handlers.

- **Module 14 – “GUI-Based Applications”**

This module discusses a variety of GUI elements.

- **Module 15 – “Threads”**

Threads are a complex topic; this module explains threading as it relates to the Java programming language and introduces a straightforward example of thread communication and synchronization.

- **Module 16 – “Networking”**

This module introduces the Java network programming package and demonstrates a TCP/IP client-server model.

Course Objectives

Upon completion of this course, you should be able to:

- Describe key language features
- Compile and run a Java technology application
- Use the online hypertext Java technology documentation
- Describe language syntactic elements and constructs
- Describe the object-oriented paradigm
- Use the object-oriented features of the Java programming language
- Use exceptions
- Use the Collections application programming interface (API)
- Read and write to files
- Develop a GUI
- Describe the Java technology Abstract Window Toolkit (AWT)
- Develop a program to take input from a GUI
- Describe event handling
- Use the `java.io` package
- Describe the basics of multithreading
- Develop multithreaded Java technology applications
- Develop Java client and server programs by using TCP/IP

Topics Not Covered

Topics Not Covered

This course does not cover the following topics. Many of these topics are covered in other courses offered by Sun Educational Services:

- Object-oriented analysis and design – Covered in OO-226: *Object-Oriented Application Analysis and Design Using UML*
- General programming concepts – Covered in SL-110: *Fundamentals of the Java™ Programming Language*

Refer to the Sun Educational Services catalog for specific information and registration.

How Prepared Are You?

Before attending this course, you should have completed SL-110: *Fundamentals of the Java Programming Language*, or have:

- Created and compiled programs with C or C++
- Created and edited text files using a text editor
- Used a World Wide Web (WWW) browser, such as Netscape Navigator™

Introductions

Now that you have been introduced to the course, introduce yourself to the other students and the instructor, addressing the following items:

- Name
- Company affiliation
- Title, function, and job responsibility
- Experience related to topics presented in this course
- Reasons for enrolling in this course
- Expectations for this course

How to Use Course Materials

To enable you to succeed in this course, these course materials contain a learning module that is composed of the following components:

- Goals – You should be able to accomplish the goals after finishing this course and meeting all of its objectives.
- Objectives – You should be able to accomplish the objectives after completing a portion of instructional content. Objectives support goals and can support other higher-level objectives.
- Lecture – The instructor presents information specific to the objective of the module. This information helps you learn the knowledge and skills necessary to succeed with the activities.
- Activities – The activities take various forms, such as an exercise, self-check, discussion, and demonstration. Activities help you facilitate the mastery of an objective.
- Visual aids – The instructor might use several visual aids to convey a concept, such as a process, in a visual form. Visual aids commonly contain graphics, animation, and video.

Typographical Conventions and Symbols

The following conventions are used in this course to represent various training elements and alternative learning resources.

Icons



Additional resources – Indicates other references that provide additional information on the topics described in the module.



Discussion – Indicates a small-group or class discussion on the current topic is recommended at this time.



Note – Indicates additional information that can help students but is not crucial to their understanding of the concept being described. Students should be able to understand the concept or complete the task without this information. Examples of notational information include keyword shortcuts and minor system adjustments.



Caution – Indicates that there is a risk of personal injury from a nonelectrical hazard, or risk of irreversible damage to data, software, or the operating system. A caution indicates that the possibility of a hazard (as opposed to certainty) might happen, depending on the action of the user.



This indicates that a slide is related to the material in the student guide at the location of the icon. The title of slide appears above the visual aid icon.

Typographical Conventions

Courier is used for the names of commands, files, directories, programming code, and on-screen computer output; for example:

Use `ls -al` to list all files.
system% You have mail.

Courier is also used to indicate programming constructs, such as class names, methods, and keywords; for example:

The `getServletInfo` method is used to get author information.
The `java.awt.Dialog` class contains `Dialog` constructor.

Courier bold is used for characters and numbers that you type; for example:

To list the files in this directory, type:
`# ls`

Courier bold is also used for each line of programming code that is referenced in a textual description; for example:

```
1 import java.io.*;  
2 import javax.servlet.*;  
3 import javax.servlet.http.*;
```

Notice the `javax.servlet` interface is imported to allow access to its life cycle methods (Line 2).

Courier italics is used for variables and command-line placeholders that are replaced with a real name or value; for example:

To delete a file, use the `rm filename` command.

Courier italic bold is used to represent variables whose values are to be entered by the student as part of an activity; for example:

Type `chmod a+rwx filename` to grant read, write, and execute rights for `filename` to world, group, and users.

Palatino italics is used for book titles, new words or terms, or words that you want to emphasize; for example:

Read Chapter 6 in the *User's Guide*.
These are called *class* options.

Additional Conventions

Java programming language examples use the following additional conventions:

- Method names are not followed with parentheses unless a formal or actual parameter list is shown; for example:
“The `doIt` method...” refers to any method called `doIt`.
“The `doIt()` method...” refers to a method called `doIt` that takes no arguments.
- Line breaks occur only where there are separations (commas), conjunctions (operators), or white space in the code. Broken code is indented four spaces under the starting code.
- If a command used in the Solaris OS is different from a command used in the Microsoft Windows platform, both commands are shown; for example:

If working in the Solaris OS

```
$ cd $SERVER_ROOT/bin
```

If working in Microsoft Windows

```
C:\> cd %SERVER%\bin
```

Module 1

Getting Started

Objectives

Upon completion of this module, you should be able to:

- Describe the key features of Java technology
- Write, compile, and run a simple Java technology application
- Describe the function of the Java Virtual Machine (JVM™)¹
- Define garbage collection
- List the three tasks performed by the Java platform that handle code security

This module provides a general overview of Java technology, including the JVM, garbage collection, security features, and JVM Debug Interface tool.

1. The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java™ platform.

Relevance



Relevance

Discussion – The following questions are relevant to the material presented in this module:

- Is the Java programming language a complete language or is it useful only for writing programs for the web?

- Why do you need another programming language?

- How does the Java technology platform improve on other language platforms?

Additional Resources



Additional resources – The following references provide additional details on the topics discussed in this module:

- Gosling, Joy, Bracha, and Steele. *The Java Language Specification, Second Edition*. Addison-Wesley. 2000. [Also online at: <http://java.sun.com/docs/books/jls/>].
- Lindholm and Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley. 1999. [Also online at: <http://java.sun.com/docs/books/vmspec/>].
- Yellin, Frank. Low-Level Security in Java, white paper. [Online]. Available: <http://java.sun.com/sfaq/verifier.html>.

What Is the Java™ Technology?

The Java technology is:

- A programming language
- A development environment
- An application environment
- A deployment environment

The syntax of the Java programming language is similar to C++ syntax. You can use the Java programming language to create all kinds of applications that you could create by using any conventional programming language.

As a development environment, Java technology provides you with a large suite of tools: a compiler, an interpreter, a documentation generator, a class file packaging tool, and so on.

The Java programming language is usually mentioned in the context of the World Wide Web (web) and browsers that are capable of running programs called *applets*. Applets are programs written in the Java programming language that reside on web servers, are downloaded by a browser to a client's system, and are run by that browser. Applets are usually small in size to minimize download time and are invoked by a Hypertext Markup Language (HTML) web page.

Java technology *applications* are standalone programs that do not require a web browser to execute. Typically, they are general-purpose programs that run on any machine where the Java runtime environment (JRE) is installed.

There are two main *deployment environments*. First, the JRE supplied by the Java 2 Software Development Kit (Java 2 SDK) contains the complete set of class files for all of the Java technology packages, which includes basic language classes, GUI component classes, an advanced Collections API, and so on. The other main deployment environment is on your web browser. Most commercial browsers supply a Java technology interpreter and runtime environment.

Primary Goals of the Java Technology

Java technology provides the following:

- A language that is easy to program because it:
 - Eliminates many pitfalls of other languages, such as pointer arithmetic and memory management that affect the robustness of the code
 - Is object-oriented to help you visualize the program in real-life terms
 - Enables you to streamline the code
- An interpreted environment resulting in the following benefits:
 - Speed of development – Reduces the compile-link-load-test cycle
 - Code portability – Enables you to write code that can be run on multiple operating systems on any certified JVM
- A way for programs to run more than one thread of activity
- A means to change programs dynamically during their runtime life by enabling them to download code modules
- A means of ensuring security by checking loaded code modules

The Java technology architecture uses the following features to fulfill the previously listed goals:

- The JVM
- Garbage collection
- The JRE
- JVM tool interface

The Java Virtual Machine

The Java Virtual Machine Specification defines the JVM as:

An imaginary machine that is implemented by emulating it in software on a real machine. Code for the JVM is stored in .class files, each of which contains code for at most one public class.

The Java Virtual Machine Specification provides the hardware platform specifications to which you compile all Java technology code. This specification enables the Java software to be platform-independent because the compilation is done for a generic machine, known as the JVM. You can emulate this *generic machine* in software to run on various existing computer systems or implement it in hardware.

The compiler takes the Java application source code and generates *bytecodes*. Bytecodes are machine code instructions for the JVM. Every Java technology interpreter, regardless of whether it is a Java technology development tool or a web browser that can run applets, has an implementation of the JVM.

The JVM specification provides concrete definitions for the implementation of the following: an instruction set (equivalent to that of a central processing unit [CPU]), a register set, the class file format, a runtime stack, a garbage-collected heap, a memory area, fatal error reporting mechanism, and high-precision timing support.

The code format of the JVM machine consists of compact and efficient bytecodes. Programs represented by JVM bytecodes must maintain proper type discipline. The majority of type checking is done at compile time.

Any compliant Java technology interpreter must be able to run any program with class files that conform to the class file format specified in *The Java Virtual Machine Specification*.

The JVM design enables the creation of implementations for multiple operating environments. For example, Sun Microsystems provides implementations of the JVM for the Solaris OS and the Linux and Microsoft Windows operating environments.

Garbage Collection

Many programming languages permit the memory to be allocated dynamically at runtime. The process of allocating memory varies based on the syntax of the language, but always involves returning a pointer to the starting address of a memory block.

After the allocated memory is no longer required (the pointer that references the memory has gone *out of extent*), the program or runtime environment should de-allocate the memory.

In C, C++, and other languages, you are responsible for de-allocating the memory. This can be a difficult exercise at times, because you do not always know in advance when memory should be released. Programs that do not de-allocate memory can crash eventually when there is no memory left on the system to allocate. These programs are said to have *memory leaks*.

The Java programming language removes you from the responsibility of de-allocating memory. It provides a system-level thread that tracks each memory allocation. During idle cycles in the JVM, the garbage collection thread checks for and frees any memory that can be freed.

Garbage collection happens automatically during the lifetime of a Java technology program, eliminating the need to deallocate memory and avoiding memory leaks. However, garbage collection schemes can vary dramatically across JVM implementations.

The Java Runtime Environment

Figure 1-1 illustrates the JRE and how it enforces code security.

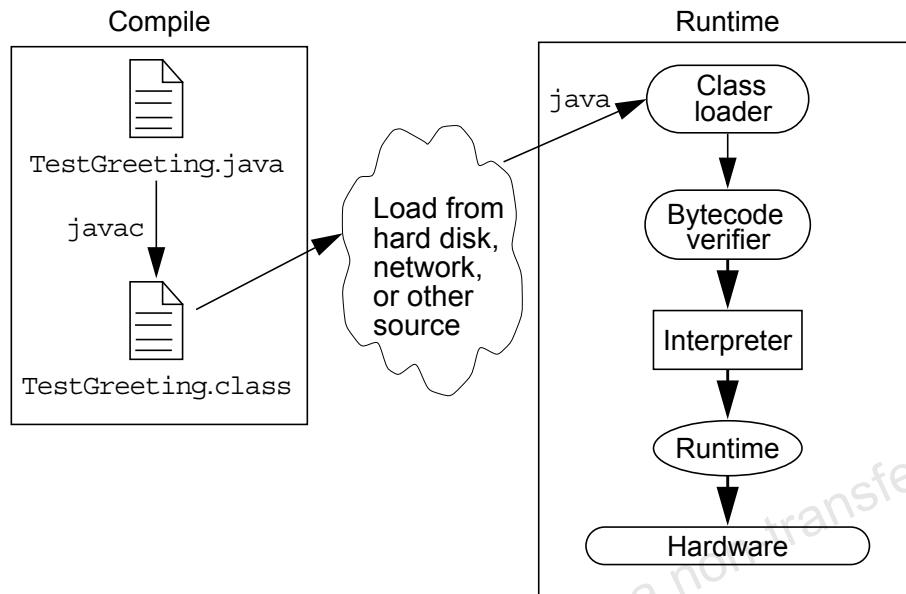


Figure 1-1 Operation of the JRE

Java software source files are *compiled* in the sense that they are converted into a set of bytecodes from the text format in which you write them. The bytecodes are stored in .class files.

At runtime, the bytecodes that make up a Java software program are loaded, checked, and run in an interpreter. In the case of applets, you can download the bytecodes, and then they are interpreted by the JVM built into the browser. The interpreter has two functions: it executes bytecodes and it makes the appropriate calls to the underlying hardware.

In some Java technology runtime environments, a portion of the verified bytecode is compiled to native machine code and executed directly on the hardware platform. This enables the Java software code to run close to the speed of C or C++ with a small delay at load time to enable the code to be compiled to the native machine code (see Figure 1-2).

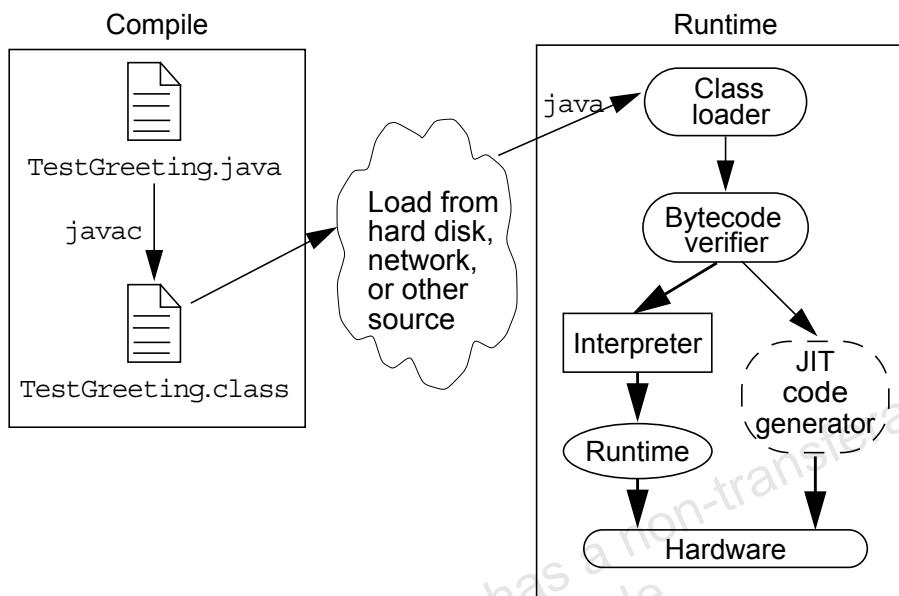


Figure 1-2 Operation of the JRE With a Just-In-Time (JIT) Compiler



Note – Sun Microsystems has enhanced the JVM machine by adding new performance-enabling technologies. One of these technologies is called the Java HotSpot™ virtual machine and has the potential to enable the Java programming language to run as fast as compiled C++ code.

Primary Goals of the Java Technology

JVM™ Tasks

The following section provides a more comprehensive discussion of the three main tasks performed by the JVM:

- Loads code – Performed by the class loader
- Verifies code – Performed by the bytecode verifier
- Executes code – Performed by the runtime interpreter

The Class Loader

The class loader loads all classes needed for the execution of a program. The class loader adds security by separating the namespaces for the classes of the local file system from those imported from network sources. This limits any Trojan Horse applications, because local classes are always loaded first.

After all of the classes have been loaded, the memory layout of the executable file is determined. At this point, specific memory addresses are assigned to symbolic references and the lookup table is created. Because memory layout occurs at runtime, the Java technology interpreter adds protection against unauthorized access into the restricted areas of code.



The Bytecode Verifier

Java software code passes several tests before running on your machine. The JVM puts the code through a bytecode verifier that tests the format of code fragments and checks code fragments for illegal code, which is code that forges pointers, violates access rights on objects, or attempts to change object type.

Note – All class files imported across the network pass through the bytecode verifier.

Verification Process

The bytecode verifier makes four passes on the code in a program. It ensures that the code adheres to the JVM specifications and does not violate system integrity. If the verifier completes all four passes without returning an error message, then the following is ensured:

- The classes adhere to the class file format of the JVM specification.
- There are no access restriction violations.
- The code causes no operand stack overflows or underflows.
- The types of parameters for all operational codes are correct.
- No illegal data conversions, such as converting integers to object references, have occurred.

A Simple Java Application

Like any other programming language, you use the Java programming language to create applications. Code 1-1 and Code 1-2 show a simple Java application that prints a greeting to the world.

Code 1-1 The TestGreeting.java Application

```
1  //
2  // Sample "Hello World" application
3  //
4  public class TestGreeting {
5      public static void main (String[] args) {
6          Greeting hello = new Greeting();
7          hello.greet();
8      }
9  }
```

Code 1-2 The Greeting.java Class

```
1  public class Greeting {
2      public void greet() {
3          System.out.println("hi");
4      }
5  }
```

The TestGreeting Application

Code 1-3 Lines 1–3

```
1  //
2  // Sample "Hello World" application
3  //
```

Lines 1–3 in the program are comment lines.

Code 1-4 Line 4

```
4  public class TestGreeting {
```

Line 4 declares the class name as `TestGreeting`. A class name specified in a source file creates a `classname.class` file when the source file is being compiled. If you do not specify a target directory for the compiler to use, this class file is in the same directory as the source code. In this case, the compiler creates a file called `TestGreeting.class`. It contains the compiled code for the public class `TestGreeting`.

Code 1-5 Line 5

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

Line 5 is where the program starts to execute. The Java technology interpreter must find this defined exactly as given or it refuses to run the program.

Other programming languages, notably C and C++, also use the `main()` declaration as the starting point for execution. The various parts of this declaration are briefly described here. The details are covered later in this course.

If the program is given any arguments on its command line, these are passed into the `main()` method in an array of `String` called `args`. In this example, no arguments are used.

The following describes each element of Line 5:

- `public` – The method `main()` can be accessed by anything, including the Java technology interpreter.
- `static` – This keyword tells the compiler that the `main()` method is usable in the context of the class `TestGreeting`. No instance of the class is needed to execute static methods.

A Simple Java Application

- `void` – This keyword indicates that the method `main()` does not return any value. This is important because the Java programming language performs careful type-checking to confirm that the methods called return the types with which they were declared.
- `String args[]` – This method declares the single parameter to the `main` method, `args`, and has the type of a `String` array. When this method is called, the `args` parameter contains the arguments typed on the command line following the class name; for example:

```
java TestGreeting args[0] args[1] . . .
```

Code 1-6 Line 6

```
6     Greeting hello = new Greeting();
```

Line 6 illustrates how to create an object, referred to by the `hello` variable. The `new Greeting` syntax tells the Java technology interpreter to construct a new object of the class `Greeting`.

Code 1-7 Line 7

```
7     hello.greet();
```

Lines 7 demonstrates an object method call. This call tells the `hello` object to greet the world. The implementation of this method is shown on Lines 3–5 of the `Greeting.java` file.

Code 1-8 Lines 8–9

```
8     }
9 }
```

Lines 8–9 of the program, the two braces, close the method `main()` and the class `TestGreeting`, respectively.

The Greeting Class

Code 1-9 Line 1

```
1 public class Greeting {
```

Line 1 declares the Greeting class.

Code 1-10 Lines 2–4

```
2 public void greet() {
3     System.out.println("hi");
4 }
```

Lines 2–4 demonstrate the declaration of a method. This method is declared public, making it accessible to the TestGreeting program. It does not return a value, so void is used as the return type.

The greet method sends a string message to the standard output stream. The println() method is used to write this message to the standard output stream.

Code 1-11 Line 5

```
5 }
```

Line 5 closes the class declaration for Greeting.

Compiling and Running the TestGreeting Program

After you have created the `TestGreeting.java` source file, compile it with the following line:

```
javac TestGreeting.java
```

If the compiler does not return any messages, the new file `TestGreeting.class` is stored in the same directory as the source file, unless specified otherwise. The `Greeting.java` file has been compiled into `Greeting.class`. This is done automatically by the compiler because the `TestGreeting` class uses the `Greeting` class.

To run your `TestGreeting` application, use the Java technology interpreter. The executables for the Java technology tools (`javac`, `java`, `javadoc`, and so on) are located in the `bin` directory.

```
java TestGreeting
```

Note – You must set the `PATH` environment variable to find `java` and `javac`; ensure that it includes `java_root/bin` (where `java_root` represents the directory root where the Java technology software is installed).



Troubleshooting the Compilation

The following sections describe errors that you might encounter when compiling code.

Compile-Time Errors

The following are common errors seen at compile time, with examples of compiler or runtime messages. Your messages can vary depending on which version of the Java 2 SDK you are using.

- `javac: Command not found`

The PATH variable is not set properly to include the javac compiler. The javac compiler is located in the bin directory below the installed Java Development Kit (JDK™) software directory.

- ```
Greeting.java:4:cannot resolve symbol
symbol : method println (java.lang.String)
location: class java.io.PrintStream
System.out.println("hi");
^
```

The method name `println` is typed incorrectly.

- **Class and file naming**

If the .java file contains a public class, then it must have the same file name as that class. For example, the definition of the class in the previous example is:

```
public class TestGreeting
```

The name of the source file must be `TestGreeting.java`. If you named the file `TestGreet.java`, then you would get the error message:

```
TestGreet.java:4: Public class TestGreeting must be
defined in a file called "TestGreeting.java".
```

- **Class count**

You should declare only one top-level, non-static class to be public in each source file, and it must have the same name as the source file. If you have more than one public class, then you will get the same message as in the previous bullet for every public class in the file that does not have the same name as the file.

## A Simple Java Application

---

### Runtime Errors

Some of the errors generated when typing `java TestGreeting` are:

- Can't find class `TestGreeting`

Generally, this means that the class name specified on the command line was spelled differently than the `filename.class` file. The Java programming language is *case-sensitive*.

For example,

```
public class TestGreet {
```

creates a `TestGreet.class`, which is not the class name (`TestGreeting.class`) that the compiler expected.

- Exception in thread "main" `java.lang.NoSuchMethodError: main`

This means that the class you told the interpreter to execute does not have a static `main` method. There might be a `main` method, but it might not be declared with the `static` keyword or it might have the wrong parameters declared, such as:

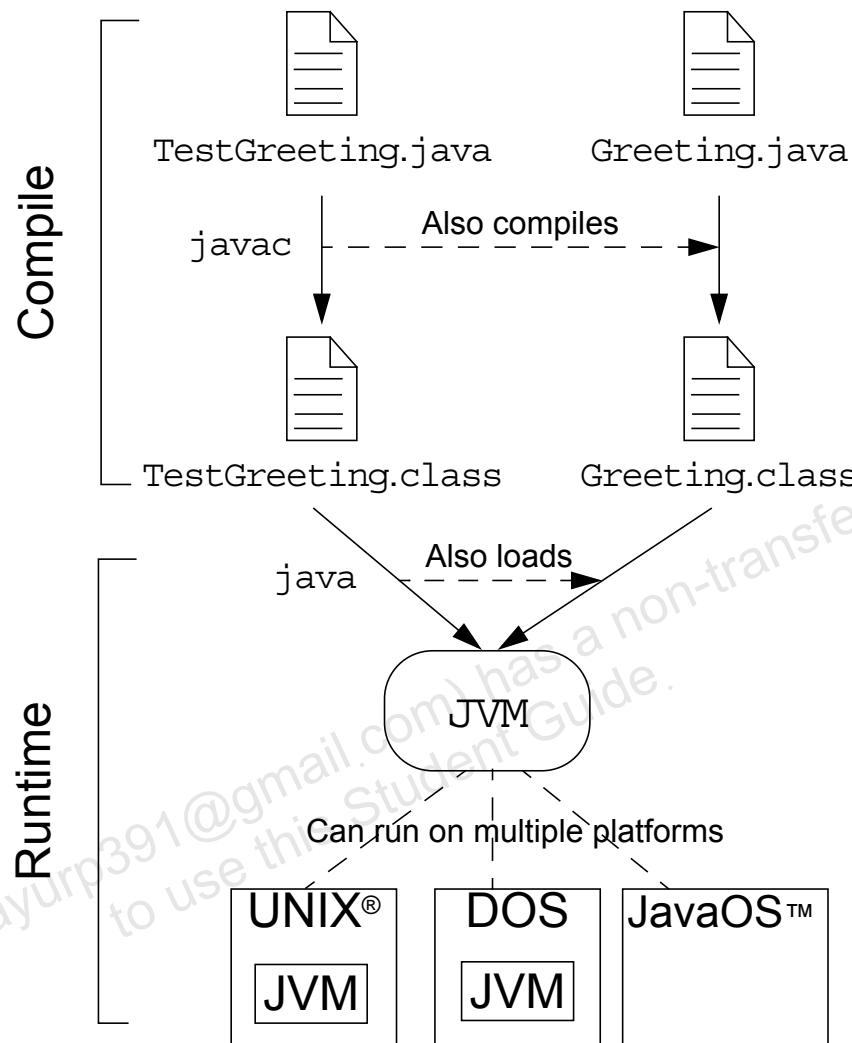
```
public static void main(String args) {
```

In this example, `args` is a single string, not an array of strings.

```
public static void main() {
```

In this example, the coder forgot to include any parameter list.

Figure 1-3 illustrates how Java technology programs can be compiled and then run on the JVM. There are many implementations of the JVM on different hardware and operating system platforms.



**Figure 1-3** Java Technology Runtime Environment



## Module 2

---

# Object-Oriented Programming

---

## Objectives

Upon completion of this module, you should be able to:

- Define modeling concepts: *abstraction*, *encapsulation*, and *packages*
- Discuss why you can reuse Java technology application code
- Define *class*, *member*, *attribute*, *method*, *constructor*, and *package*
- Use the access modifiers `private` and `public` as appropriate for the guidelines of encapsulation
- Invoke a method on a particular object
- Use the Java technology API online documentation

This module is the first of three modules that describe the object-oriented (OO) paradigm and the object-oriented features of the Java programming language.

---

## Relevance



**Discussion** – The following questions are relevant to the material presented in this module:

- What is your understanding of software analysis and design?

---

---

---

- What is your understanding of design and code reuse?

---

---

---

- What features does the Java programming language possess that make it an object-oriented language?

---

---

---

- Define the term *object-oriented*.

---

---

---

# Software Engineering

Software engineering is a difficult and often unruly discipline. For the past half-century, computer scientists, software engineers, and system architects have sought to make creating software systems easier by providing reusable code. Figure 2-1 shows a brief history of software engineering.

| Toolkits / Frameworks / Object APIs (1990s–Up)  |                    |                           |            |                              |                   |
|-------------------------------------------------|--------------------|---------------------------|------------|------------------------------|-------------------|
| Java 2 SDK                                      | AWT / J.F.C./Swing | Jini™                     | JavaBeans™ | JDBC™                        |                   |
| Object-Oriented Languages (1980s–Up)            |                    |                           |            |                              |                   |
| SELF                                            | Smalltalk          | Common Lisp Object System | Eiffel     | C++                          | Java              |
| Libraries / Functional APIs (1960s–Early 1980s) |                    |                           |            |                              |                   |
| NASTRAN                                         | TCP/IP             | ISAM                      | X-Windows  | OpenLook                     |                   |
| High-Level Languages (1950s–Up)                 |                    |                           |            | Operating Systems (1960s–Up) |                   |
| Fortran                                         | LISP               | C                         | COBOL      | OS/360                       | UNIX              |
|                                                 |                    |                           |            | MacOS                        | Microsoft Windows |
| Machine Code (Late 1940s–Up)                    |                    |                           |            |                              |                   |

**Figure 2-1** Brief History of Software Engineering

At first, they created computer languages to conceal the complexity of the machine language and added callable operating system procedures to handle common operations, such as opening, reading, and writing to files.

Other developers grouped collections of common functions and procedures into libraries for anything from calculating structural loads for engineering (NASTRAN), writing character and byte streams between computers on a network (TCP/IP), accessing data through an indexed sequential access method (ISAM), and creating windows, graphics, and other GUI widgets on a bit-mapped monitor (X-Windows and Open Look).

Many of these libraries manipulated data in the form of *open record* data-structures, such as the C language `struct`. The main problem with record structures is that the library designer cannot hide the implementation of the data used in the procedures. This makes it difficult to modify the implementation of the library without affecting the client code because that code is often tied to the particular details of the data structures.

By the late 1980s, object-oriented programming (OOP) became popular with C++. One of the greatest advantages of OOP was the ability to hide certain aspects of a library's implementation so that updates do not affect client code (assuming the interfaces do not change). The other important advantage is that procedures were associated with the data structure. The combination of data attributes and procedures (called methods) are known as a *class*.

Today's equivalent of function libraries are class libraries or toolkits. These libraries provide classes to perform many of the same operations as functional libraries but, with the use of subclassing, client programmers can easily extend these tools for their own applications. Frameworks provide APIs that different vendors can implement to allow you to choose the amount of flexibility and performance suitable to your applications.

Java technology is a platform that is continuously extended by new APIs and frameworks, such as Java Foundation Classes/Swing (J.F.C./Swing) and other J.F.C. technology, JavaBeans architecture (Java technology's component architecture), and the Java DataBase Connectivity API (JDBC API). The list of Java APIs is long and growing.

## The Analysis and Design Phase

There are five primary workflows in a software development project: Requirement Capture, Analysis, Design, Implementation, and Test. They are all important, however, you must ensure that you schedule enough time for the analysis and design phases.

During the Analysis phase, you define *what* the system is supposed to accomplish. You do this by defining the set of actors (users, devices, and other systems that interact with the proposed system) and activities that the proposed system must accommodate. Also, the Analysis phase must identify the *domain objects* (both physical and conceptual) that the proposed system will manipulate and the behaviors and interactions among these objects. These behaviors implement the activities that the proposed system must support. The description of the activities should be detailed enough to create baseline criteria for the Test phase.

During the Design phase, you define *how* the system will achieve its goals. In this phase, you create a model of the actors, activities, objects, and behaviors for the proposed system. For this class, you use the Unified Modeling Language (UML) as your modeling tool.



---

**Note** – UML is a large and complex language. You use only a small portion of it. Appendix B is a reference to the UML elements that are used in this course. It also shows you how to implement Java technology code from a UML class diagram.

---

## Analysis and Design Example

This module uses the example of a shipping company. You assume a simple set of requirements:

- The software must support a single shipping company.
- The shipping company maintains a fleet of vehicles that transport boxes.
- The weight of the boxes is the only important factor in loading a vehicle.
- The shipping company owns two types of vehicles: trucks and river barges.
- Boxes are weighed on scales that measure in kilograms; however, the algorithms for calculating vehicle engine power require the total vehicle load to be measured in newtons.

---

**Note** – A *newton* is a measure of force (or weight) that is equivalent to 9.8 times the mass of the object in kilograms.

---

- You use a GUI to keep track of adding boxes to vehicles.
- You must generate several reports from the fleet records.

From these requirements, you can create a high-level design:

- The following objects must be represented in the system: a company and two types of vehicles.
- A company is an aggregate of multiple vehicle objects.
- Other functional objects exist: several reports and GUI screens.



## Abstraction

Software design has moved from low-level constructs, such as writing in machine code, toward much higher levels. There are two interrelated forces that guided this process: simplification and abstraction.

Simplification was at work when early language designers built high-level language constructs, such as the IF statements and FOR loops, out of raw machine codes. Abstraction is the force that hides private implementation details behind public interfaces.

The concept of abstraction led to the use of subroutines (functions) in high-level languages and to the pairing of functions and data into objects. At higher levels, abstraction led to the development of frameworks and APIs.

## Classes as Blueprints for Objects

Just as a draftsman can create a blueprint for a device that can be used many times to construct actual devices, a *class* is a software blueprint that you can use to *instantiate* (that is, create) many individual objects. A class defines the set of data elements (*attributes*) that define the objects, as well as the set of behaviors or functions (called *methods*) that manipulate the object or perform interactions between related objects. Together, attributes and methods are called *members*. For example, a vehicle object in a shipping application must keep track of its maximum and current load along with methods for adding a box (with a certain weight) to the vehicle.

The Java technology programming language supports three key features of OOP: encapsulation, inheritance, and polymorphism.

---

**Note** – Encapsulation is covered in “Encapsulation” on page 2-15. Inheritance and polymorphism are described in Module 6.

---



# Declaring Java Technology Classes

The Java technology class declaration takes the following form:

```
<modifier>* class <class_name> {
 <attribute_declarator>*
 <constructor_declarator>*
 <method_declarator>*
}
```

The `<class_name>` can be any legal identifier, and it is the name of the class being declared. There are several possible `<modifier>` keywords, but for now, use only `public`. This declares that the class is accessible to the universe. The body of the class declares the set of data attributes, constructors, and methods associated with the class. Code 2-1 shows an example class declaration.

## Code 2-1 Example Class Declaration

```
1 public class Vehicle {
2 private double maxLoad;
3 public void setMaxLoad(double value) {
4 maxLoad = value;
5 }
6 }
```

## Declaring Attributes

The declaration of an object attribute takes the following form:

```
<modifier>* <type> <name> [= <initial_value>];
```

**Example:**

```
1 public class Foo {
2 private int x;
3 private float y = 10000.0F;
4 private String name = "Bates Motel";
5 }
```

The *<name>* can be any legal identifier, and it is the name of the attribute being declared. There are several possible values for *<modifier>*, but for now, use either `public` or `private`. The `private` keyword declares that the attribute is accessible only to the methods within this class. The *<type>* of the attribute can be any primitive type (`int`, `float`, and so on) or any class.

---

## Declaring Methods

# Declaring Methods

To define methods, the Java programming language uses an approach that is similar to other languages, particularly C and C++. The declaration takes the following basic form:

```
<modifier>* <return_type> <name> (<argument>*) {
 <statement>*
}
```

The *<name>* can be any legal identifier, with some restrictions based on the names that are already in use.

The *<modifier>* segment is optional and can carry a number of different modifiers, including (but not limited to) public, protected, and private. The public access modifier indicates that the method can be called from other code. The private method indicates that a method can be called only by the other methods in the class. The protected method is described later in this course.

The *<return\_type>* indicates the type of value returned by the method. If the method does not return a value, it should be declared void. Java technology is rigorous about returned values, and if the declaration states that the method returns an int, for example, then the method must return an int from all possible return paths (and can be invoked only in contexts that expect an int to be returned). Use the return statement within a method to pass back a value.

The *<argument>* list allows argument values to be passed into a method. Elements of the list are separated by commas, while each element consists of a type and an identifier.

## Example

Code 2-2 shows two methods for the Dog class. The method `getWeight` returns the `weight` data attribute; it uses no parameters. A value is returned from a method using the `return` statement (Line 4). The method `setWeight` modifies the `weight` value with the parameter `newWeight`; it does not return any value. This method uses a conditional statement to restrict the client code from setting the dog's weight to a negative number or zero.

### Code 2-2 Example Methods

```
1 public class Dog {
2 private int weight;
3 public int getWeight() {
4 return weight;
5 }
6 public void setWeight(int newWeight) {
7 if (newWeight > 0) {
8 weight = newWeight;
9 }
10 }
11 }
```

## Accessing Object Members

---

# Accessing Object Members

In the previous example, you saw the following line of code in the `TestDog.main` method:

```
d.setWeight(42);
```

This line of code tells the `d` object (actually a variable, `d`, holding a reference to an object of type `Dog`) to execute its `setWeight` method. This is called *dot notation*. The dot operator enables you to access non-private attribute and method members of a class.

Within the definition of a method, you do not need to use the dot notation for accessing local members. For example, the `setWeight` method of the `Dog` class does not use the dot notation to access the `weight` attribute.

Code 2-3 demonstrates the behavior of the `Dog` methods. When the `Dog` object is created the `weight` instance variable is initialized to 0. Therefore, the `getWeight` method returns 0. Line 6 of this code sets the `weight` to 42; this is a valid argument and `setWeight` method sets the `weight` variable. However, setting the `weight` to -42 (Line 9) is illegal and `setWeight` method does not alter the `weight` variable.

### Code 2-3 Example Invocation of Methods

```

1 public class TestDog {
2 public static void main(String[] args) {
3 Dog d = new Dog();
4 System.out.println("Dog d's weight is "
5 + d.getWeight());
6 d.setWeight(42);
7 System.out.println("Dog d's weight is "
8 + d.getWeight());
9 d.setWeight(-42);
10 System.out.println("Dog d's weight is "
11 + d.getWeight());
12 }
13 }
```

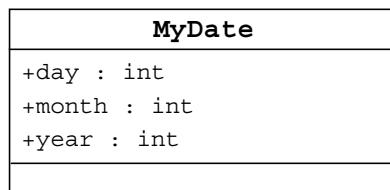
The output is:

```

Dog d's weight is 0
Dog d's weight is 42
Dog d's weight is 42
```

## Information Hiding

Suppose that you have a `MyDate` class that includes the attributes: `day`, `month`, and `year`. Figure 2-2 shows a class diagram of a possible implementation of the `MyDate` class.



**Figure 2-2** UML Class Diagram of the `MyDate` Class

A simple implementation allows direct access to these data attributes; for example:

```
public class MyDate {
 public int day;
 public int month;
 public int year;
}
```

Client code then accesses the attributes directly and makes mistakes; for example (`d` refers to a `MyDate` object):

```
d.day = 32;
// invalid day

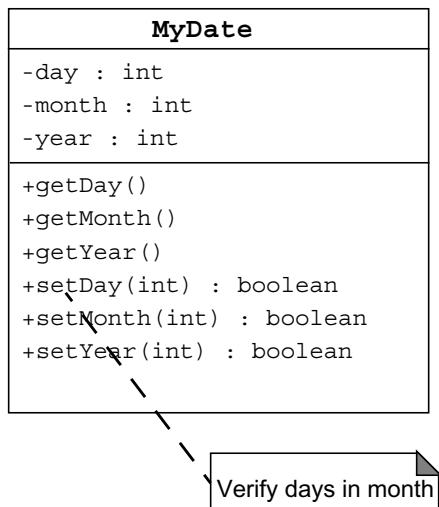
d.month = 2; d.day = 30;
// plausible but wrong

d.day = d.day + 1;
// no check for wrap around
```

## Information Hiding

To solve the problem, hide the data attributes by making them private and supply retrieval access methods, `getXYZ()`, which are often called *getters*, and storage access methods, `setXYZ()`, which are often called *setters*.

Figure 2-3 shows another UML diagram of the `MyDate` class that hides the instance variables behind getter and setter methods.



**Figure 2-3** Hiding the Instance Variables of the `MyDate` Class

These methods allow the class to modify the internal data, but more importantly, to verify that the requested changes are valid. For example:

```

MyDate d = new MyDate();

d.setDay(32);
// invalid day, returns false

d.setMonth(2); d.setDay(30);
// plausible but wrong, setDay returns false

d.setDay(d.getDay() + 1);
// this will return false if wrap around needs to occur

```

## Encapsulation

Encapsulation is the methodology of hiding certain elements of the implementation of a class but providing a public interface for the client software. This is an extension of information hiding because the information in the data attributes is a significant element of a class's implementation.

For example, the programmer for the `MyDate` class might decide to reimplement the internal representation of a date as the number of days since the beginning of some epoch. This could make date comparisons and calculating date intervals easier. Because the programmer encapsulated the attributes behind a public interface, the programmer can make this change without affecting the client code. Figure 2-4 shows this variation on the `MyDate` class.

| <b>MyDate</b>    |           |
|------------------|-----------|
| -date            | : long    |
| +getDay()        | : int     |
| +getMonth()      | : int     |
| +getYear()       | : int     |
| +setDay(int)     | : boolean |
| +setMonth(int)   | : boolean |
| +setYear(int)    | : boolean |
| -isDayValid(int) | : boolean |

**Figure 2-4** Encapsulation Provides Data Representation Flexibility

# Declaring Constructors

A constructor is a set of instructions designed to initialize an instance. Parameters can be passed to the constructor in the same way as for a method. The basic declaration takes the following form:

```
[<modifier>] <class_name> (<argument>*) {
 <statement>*
}
```

The name of the constructor must always be the same as the class name. If present, the only valid modifiers (<modifier>) for constructors are public, protected, and private.

The <argument> list is the same as for method declarations.

---

**Note – Constructors are not methods. They do not have return values and are not inherited.**

---

For example:

```
1 public class Dog {
2 private int weight;
3
4 public Dog() {
5 weight = 42;
6 }
```

The Dog class has a single instance variable weight. The constructor (with no parameters) initializes weight to 42.

Constructors can also be declared with parameters. This is discussed later in this course.





## The Default Constructor

Every class has at least one constructor. If you do not write a constructor, the Java programming language provides one for you. This constructor takes no arguments and has an empty body.

The default constructor enables you to create object instances with `new Xyz();`; otherwise, you must provide a constructor for every class.

---

**Note** – If you add any constructor declaration to a class that previously had no explicit constructors, you lose the default constructor. From that point, unless the constructor you wrote takes no arguments, calls to `new Xyz()` cause compiler errors.

---

## Source File Layout

# Source File Layout

A Java technology source file takes the following form:

```
[<package_declaration>]
<import_declaration>*
<class_declaration>+
```

---

**Note** – The plus (+) indicates one or more. To be meaningful, a source file must contain at least one class definition.

---



The order of these items is important. That is, any import statements must precede all class declarations and, if you use a package declaration, it must precede both the classes and imports.

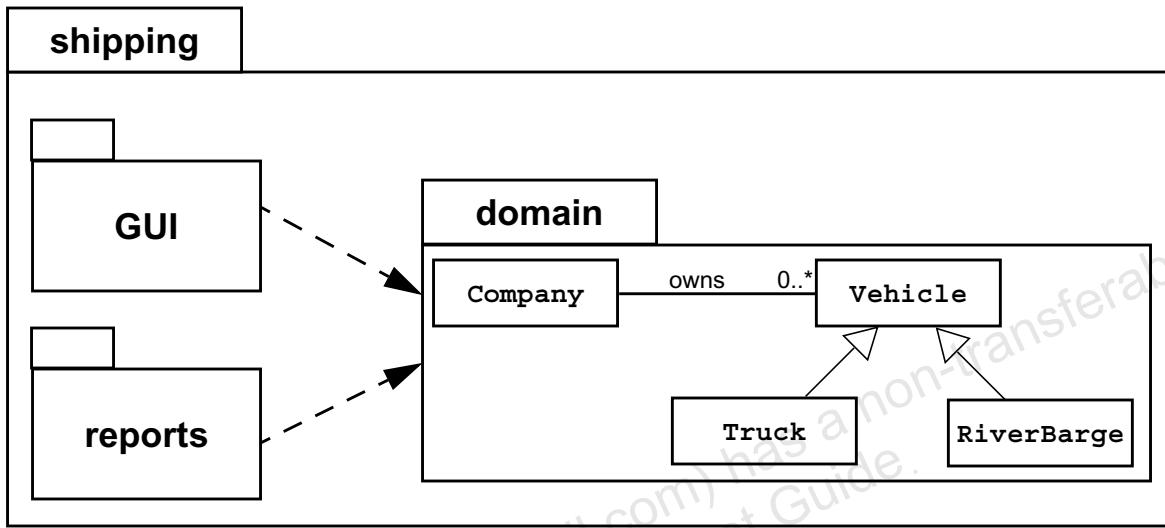
The name of the source file must be the same as the name of the public class declaration in that file. A source file can include more than one class declaration, but *only one* class can be declared public. If a source file contains no public class declarations, then the name of the source file is not restricted. However, it is good practice to have one source file for every class declaration, and the name of the file is identical to the name of the class.

For example, the file `VehicleCapacityReport.java` should look like the following:

```
1 package shipping.reports;
2
3 import shipping.domain.*;
4 import java.util.List;
5 import java.io.*;
6
7 public class VehicleCapacityReport {
8 private List vehicles;
9 public void generateReport(Writer output) {
10 // code to generate the report
11 }
12 }
```

## Software Packages

Most software systems are large. It is common to group classes into packages to ease the management of the system. UML includes the concept of packages in its modeling language. Packages can contain classes as well as other packages that form a hierarchy of packages. An example package structure is illustrated in Figure 2-5.



**Figure 2-5** An Example UML Diagram of Java Packages

There are many ways to group classes into meaningful packages. There is no right or wrong way; but a common technique is to group classes into a package by semantic similarity.

For example, a shipping software system could contain a set of domain objects (such as the company and vehicles, boxes, destinations, and so on), a set of reports, and a set of GUI panels that are used to create the main data entry application. The **GUI** and **reports** subsystems depend on the **domain** package. UML packages can be useful for modeling subsystems or other groupings according to your need. All of these packages are contained in the top-level package called **shipping**.

---

## The package Statement

The Java technology programming language provides the package statement as a way to group related classes. The package statement takes the following form:

```
package <top_pkg_name>[.<sub_pkg_name>]*;
```

You can indicate that classes in a source file belong to a particular package by using the package statement; for example:

```
1 package shipping.domain;
2
3 // Class Vehicle of the 'domain' sub-package within
4 // the 'shipping' application package.
5 public class Vehicle {
6 ...
7 }
```

The package declaration, if any, must be at the beginning of the source file. You can precede it with white space and comments, but nothing else. Only one package declaration is permitted, and it governs the entire source file. If a Java technology source file does not contain a package declaration, then the classes declared in that file belongs to the unnamed (default) package.

Package names are hierarchical and are separated by dots. It is usual for the elements of the package name to be entirely lower case. However, the class name usually starts with a capital letter, and you can capitalize the first letter of each additional word to distinguish words in the class name. These naming conventions and others are described in “Java Programming Language Coding Conventions” on page 3-21.

---

**Note** – If a package statement is not included in the file, then all classes declared in that file *belong* to the default package (that is, a package with no name).

---



# The import Statement

The `import` statement takes the following form:

```
import <pkg_name>[.<sub_pkg_name>].<class_name>;
```

or

```
import <pkg_name>[.<sub_pkg_name>].*;
```

When you want to use packages, use the `import` statement to tell the compiler where to find the classes. In fact, the package name (for example, `shipping.domain`) forms part of the name of the classes within the package. You could refer to the `Company` class as `shipping.domain.Company` throughout, or you could use the `import` statement and just the class name `Company`.

---

**Note** – The `import` statements must precede all class declarations.

---



The following is a file fragment that uses the `import` statement.

```
1 package shipping.reports;
2
3 import shipping.domain.*;
4 import java.util.List;
5 import java.io.*;
6
7 public class VehicleCapacityReport {
8 private Company companyForReport;
9 ...
10 }
```

When you use a package declaration, you do not need to import the same package or any element of that package. Remember that the `import` statement is used to make classes in other packages accessible to the current class.

The `import` statement specifies the class to which you want access. For example, if you want only the `Writer` class (from the `java.io` package) included in the current name space, then you would use:

```
import java.io.Writer;
```

## The import Statement

---

If you want access to all classes within a package, use “\*.” For example, to access all classes in the `java.io` package, use:

```
import java.io.*;
```

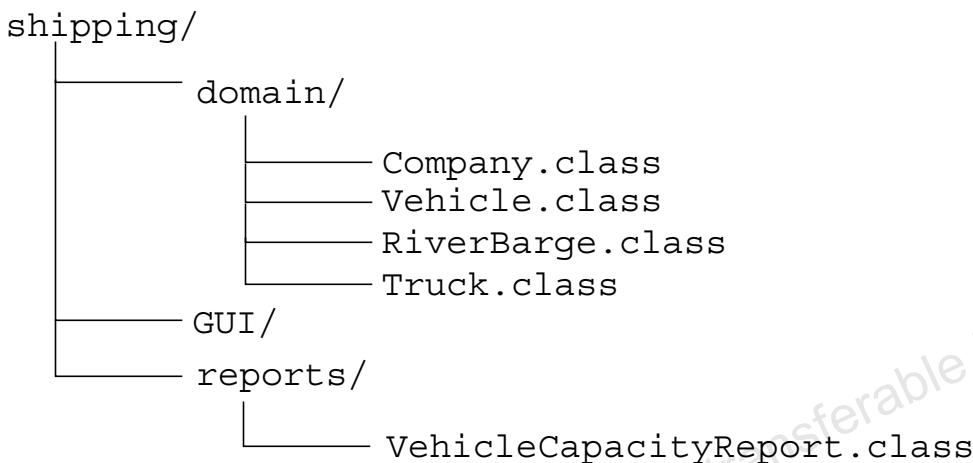
**Note** – The `import` statement allows you to use the short class names in your source program, nothing more. An `import` statement does not cause the compiler to load anything additional into working memory. In this respect, the `import` statement is quite different from the `#include` statement in C or C++. The `import` statement, whether or not it uses the wild card (\*), does not have any effect on the output class file nor does it have any effect on runtime performance. It is also very unlikely that any form of `import` statement will cause any difference in compilation performance.

---



## Directory Layout and Packages

Packages are stored in a directory tree containing a branch that is the package name. For example, the `Company.class` file should exist in the directory structure shown in Figure 2-6.

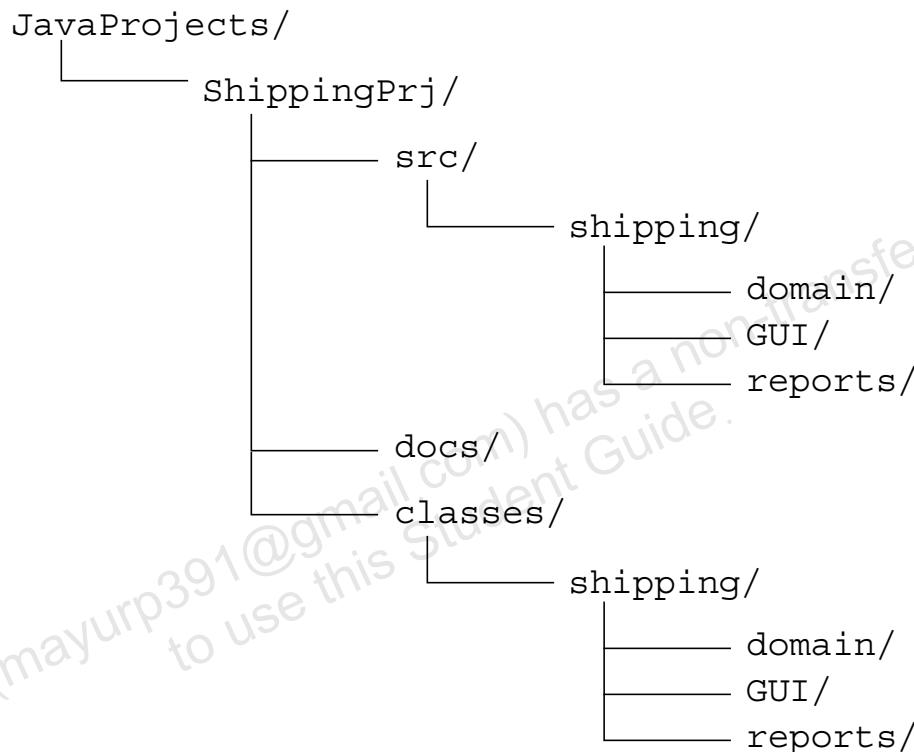


**Figure 2-6** Example Class Package Directory Structure

## Development

It is common to be working on several development projects at once. There are many ways to organize your development files. This section describes one such technique.

Figure 2-7 demonstrates an example development directory hierarchy for a development project. The important element of this hierarchy is that the source files of each project are separated from the compiled (.class) files.



**Figure 2-7** Example Project Development Directory Structure

## Compiling Using the `-d` Option

Normally, the Java compiler places the class files in the same directory as the source files. You can reroute the class files to another directory using the `-d` option of the `javac` command. The simplest way to compile files within packages is to be working in the directory one level above the beginning of the package. (In this example, the `src` directory.)

To compile all the files within the `shipping.domain` package and have the compiled classes end up in their correct package directory under `ShippingPrj/classes/`, type the following:

```
cd JavaProjects/ShippingPrj/src
javac -d ../classes shipping/domain/*.java
```

## Deployment

You can deploy an application on a client machine without manipulating the user's `CLASSPATH` environment variable. Usually, this is best done by creating an **executable Java archive (JAR) file**. To create an executable JAR file, you must create a temporary file that indicates the class name that contains your `main` method, like this:

```
Main-Class: mypackage.MyClass
```

Next, build the JAR file as normal except that you add an additional option so that the contents of this temporary file are copied into the `META-INF/MANIFEST.MF` file. Do this using the "`m`" option, like this:

```
jar cmf tempfile MyProgram.jar
```

Finally, the program can be run simply by executing a command like this:

```
java -jar /path/to/file/MyProgram.jar
```

---

**Note** – On some platforms, simply double clicking the icon for an executable JAR file is sufficient to launch the program.



## Library Deployment

Sometimes you need to deploy library code in a JAR file. In such situations it is possible to copy the JAR file into the `ext` subdirectory of the `lib` directory in the main directory of the JRE. Be careful when you do this, because if you deploy classes in this way they are usually granted full security privileges, and might cause runtime problems if any classes have naming conflicts with other classes in the core JDK or that have been installed in this way.

---

## Terminology Recap

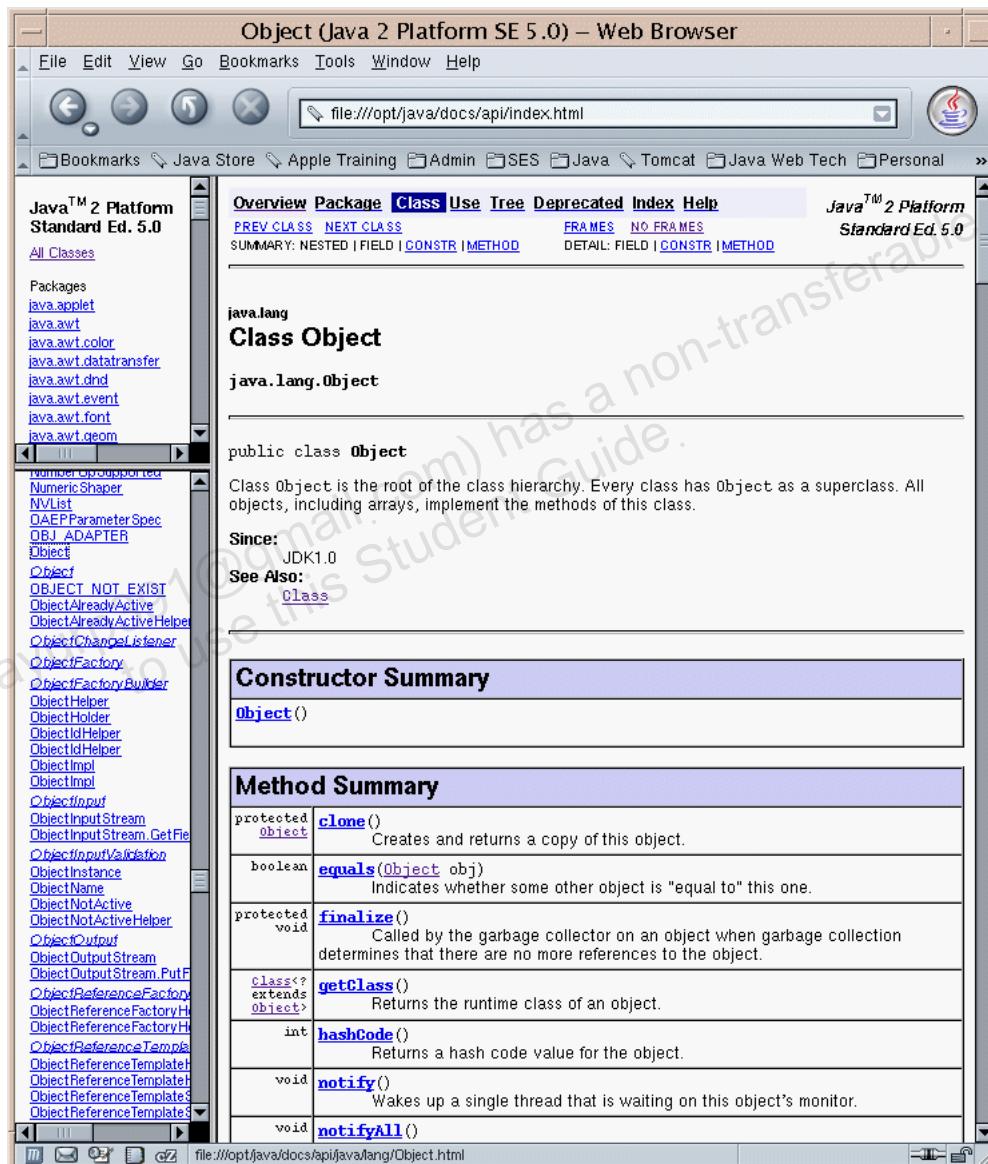
# Terminology Recap

The following describes some of the terms introduced in this module:

- Class – A way to define new types of objects in the Java programming language. The class can be considered as a blueprint, which is a model of the object that you are describing.
- Object – An actual instance of a class. An object is what you get each time you instantiate a class using `new`. An object is also known as an *instance*.
- Attribute – A data element of an object. An attribute stores information for an object. An attribute is also known as a *data member*, an *instance variable*, or a *data field*.
- Method – A functional element of an object. A method is also known as a *function* or a *procedure*.
- Constructor – A *method-like* construct used to initialize (or build) a new object. Constructors have the same name as the class.
- Package – A grouping of classes, subpackages, or both.

# Using the Java Technology API Documentation

A set of HTML files document the supplied API. The layout of this documentation is hierarchical, so that the home page lists all the packages as hyperlinks. When you select a particular package hyperlink, the classes that are members of that package are listed. Selecting a class hyperlink from a package page presents a page of information about that class. Figure 2-8 shows one such class.



**Figure 2-8** Java Technology API Documentation

## Using the Java Technology API Documentation

---

The main sections of a class document include the following:

- The class hierarchy
- A description of the class and its general purpose
- A list of attributes
- A list of constructors
- A list of methods
- A detailed list of attributes with descriptions
- A detailed list of constructors with descriptions and formal parameter lists
- A detailed list of methods with descriptions and formal parameter lists

## Module 3

---

# Identifiers, Keywords, and Types

---

## Objectives

Upon completion of this module, you should be able to:

- Use comments in a source program
- Distinguish between valid and invalid identifiers
- Recognize Java technology keywords
- List the eight primitive types
- Define literal values for numeric and textual types
- Define the terms *primitive variable* and *reference variable*
- Declare variables of class type
- Construct an object using `new`
- Describe default initialization
- Describe the significance of a reference variable
- State the consequence of assigning variables of class type

This module describes some of the basic components used in Java technology programs including variables, keywords, primitive types, and class types.

## Relevance

---

# Relevance

**Discussion** – The following questions are relevant to the material presented in this module:

- Do you know the primitive Java technology types?

---

---

---

- Can you describe the difference between variables holding primitive values as compared with object references?

---

---

---

## Comments

The three permissible styles for inserting comments are:

```
// comment on one line

/* comment on one
 * or more lines
 */

/** documentation comment
 * can also span one or more lines
 */
```

Documentation comments placed immediately before a declaration (of a variable, method, or class) indicate that the comments should be included in any documentation that is generated automatically (for example, the HTML files generated by the `javadoc` command) to serve as a description of the declared item.



**Note** – The format of these comments and the use of the Javadoc™ tool is described in the documentation for Java 2 SDK. Refer to the following Universal Resource Locator (URL):

<http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/>

---

# Semicolons, Blocks, and White Space

In the Java programming language, a statement is one or more lines of code terminated with a semicolon (:).

For example,

```
totals = a + b + c + d + e + f;
```

is the same as

```
totals = a + b + c
 + d + e + f;
```

A *block*, sometimes called a compound statement, is a group of statements bound by opening and closing braces ({ }). A block has useful features that are described in Module 4, “Expressions and Flow Control.” For now, just think of a block as a group of statements that are collected together.

The following are other examples of block statements or groupings:

```
// a block statement
{
 x = y + 1;
 y = x + 1;
}

// a class definition is contained in a block
public class MyDate {
 private int day;
 private int month;
 private int year;
}

// a block statement can be nested within
// another block statement
while (i < large) {
 a = a + i;
 // nested block
 if (a == max) {
 b = b + a;
 a = 0;
 }
 i = i + 1;
}
```

You can have *white space* between elements of the source code. Any amount of white space is allowed. You can use white space, including spaces, tabs, and new lines, to enhance the clarity and visual appearance of your source code. Compare:

```
{int x;x=23*54;}
```

with:

```
{
 int x;
 x = 23 * 54;
}
```

---

## Identifiers

# Identifiers

In the Java programming language, an *identifier* is a name given to a variable, class, or method. Identifiers start with a letter, underscore (\_), or dollar sign (\$). Subsequent characters can be digits. Identifiers are case-sensitive and have no maximum length.

The following are valid identifiers:

- identifier
- userName
- user\_name
- \_sys\_var1
- \$change

Java technology sources are in 16-bit Unicode rather than 8-bit American Standard Code for Information Interchange (ASCII) text, so a letter is a considerably wider definition than just a-z and A-Z.

While identifiers can use non-ASCII characters, be aware of the following caveats:

- Unicode can support *different* characters that look the same.
- Class names should *only* be in ASCII characters because most file systems do not support Unicode characters.

An identifier cannot be a keyword, but it can contain a keyword as part of its name. For example, thisOne is a valid identifier, but this is not because this is a Java technology keyword.

---

**Note** – Identifiers containing a dollar sign (\$) are generally unusual, although languages such as BASIC, along with VAX/VMS systems, make extensive use of them. Because they are unfamiliar, it is probably best to avoid them unless there is a local convention or other pressing reason for including this symbol in the identifier.

---



# Java Programming Language Keywords

Keywords have special meaning to the Java technology compiler. They identify a data type name or program construct name.

Table 3-1 lists keywords that are used in the Java programming language.

**Table 3-1** Java Programming Language Keywords

|          |          |            |           |              |
|----------|----------|------------|-----------|--------------|
| abstract | continue | for        | new       | switch       |
| assert   | default  | goto       | package   | synchronized |
| boolean  | do       | if         | private   | this         |
| break    | double   | implements | protected | throw        |
| byte     | else     | import     | public    | throws       |
| case     | enum     | instanceof | return    | transient    |
| catch    | extends  | int        | short     | try          |
| char     | final    | interface  | static    | void         |
| class    | finally  | long       | strictfp  | volatile     |
| const    | float    | native     | super     | while        |

**Note** – While you might think `true` and `false` are keywords, they are in fact Boolean literals, according to *The Java Language Specification* (Section 3.10.3). Similarly, `null` is in fact the null literal (Section 3.10.7).



The following are important notes about the keywords:

- The literals `true`, `false`, and `null` are lowercase, not uppercase as in the C++ language. Strictly speaking, these are not keywords but literals; however, the distinction is academic.
- There is no `sizeof` operator; direct memory access is not possible so the information would be valueless.
- The `goto` and `const` keywords are not used in the Java programming language.

# Basic Java Programming Language Types

The Java programming language has many built in data types. These fall into two broad categories: class types and primitive types. Primitive types are simple values, are not objects. Class types are used for more complex types, including all of the types that you declare yourself. Class types are used to create objects.

## Primitive Types

The Java programming language defines eight *primitive* data types, which can be considered in four categories:

- Logical – boolean
- Textual – char
- Integral – byte, short, int, and long
- Floating point – double and float

### Logical – boolean

Logical values are represented using the boolean type, which takes one of two values: true or false. These values can be used to represent any two states, such as on and off, or yes and no. The boolean type has two literal values: true and false. The following code is an example of the declaration and initialization of a boolean type variable:

```
// declares the variable truth as boolean and
// assigns it the value true
boolean truth = true;
```

---

**Note** – There are no casts between integer types and the boolean type. Some languages, most notably C and C++, allow numeric values to be interpreted as logical values. This is not permitted in the Java programming language; when a boolean type is required, you can use only boolean values.

---



## Textual – char

Single characters are represented by using the `char` type. A `char` represents a 16-bit, unsigned Unicode character. You must enclose a `char` literal in single quotes (''). For example:

|          |                                                                                                                                             |
|----------|---------------------------------------------------------------------------------------------------------------------------------------------|
| 'a'      | The letter a                                                                                                                                |
| '\t'     | A tab                                                                                                                                       |
| '\u????' | A specific Unicode character, ????, is replaced with exactly four hexadecimal digits.<br>For example, '\u03A6' is the Greek letter phi [Φ]. |

## Textual – String

You use the `String` type, which is not a primitive but a class, to represent sequences of characters. The characters themselves are Unicode, and the backslash notation shown previously for the `char` type also works in a `String`. Unlike C and C++, strings do not end with \0.

A `String` literal is enclosed in double quote marks:

"The quick brown fox jumps over the lazy dog."

Some examples of the declarations and initialization of `char` and `String` type variables are:

```
// declares and initializes a char variable
char ch = 'A';

// declares two char variables
char ch1, ch2;

// declares two String variables and initializes them
String greeting = "Good Morning !! \n";
String errorMessage = "Record Not Found !";

// declares two String variables
String str1, str2;
```

## Basic Java Programming Language Types

---

### Integral – byte, short, int, and long

There are four integral types in the Java programming language. Each type is declared using one of the keywords `byte`, `short`, `int`, or `long`. You can represent literals of integral type using decimal, octal, or hexadecimal forms as follows:

|        |                                               |
|--------|-----------------------------------------------|
| 2      | This is the decimal form for the integer 2.   |
| 077    | The leading 0 indicates an octal value.       |
| 0xBAAC | The leading 0x indicates a hexadecimal value. |

All numeric types in the Java programming language represent signed numbers.

Integral literals are of type `int` unless followed explicitly by the letter `L` that indicates a `long` value. In the Java programming language, you can use either an uppercase or lowercase `L`. A lowercase `l` is not recommended because it is hard to distinguish from the digit `1`.

Long versions of the literals shown previously are:

|         |                                                                          |
|---------|--------------------------------------------------------------------------|
| 2L      | The L indicates that the decimal value 2 is represented as a long value. |
| 077L    | The leading 0 indicates an octal value.                                  |
| 0xBAACL | The 0x prefix indicates a hexadecimal value.                             |

The size and range for the four integral types are shown in Table 3-2. The range representation is defined by the Java programming language specification as a 2's complement and is platform-independent.

**Table 3-2** Integral Data Types – Size and Range

| Integer Length | Name or Type       | Range                                       |
|----------------|--------------------|---------------------------------------------|
| 8 bits         | <code>byte</code>  | From -2 <sup>7</sup> to 2 <sup>7</sup> -1   |
| 16 bits        | <code>short</code> | From -2 <sup>15</sup> to 2 <sup>15</sup> -1 |
| 32 bits        | <code>int</code>   | From -2 <sup>31</sup> to 2 <sup>31</sup> -1 |
| 64 bits        | <code>long</code>  | From -2 <sup>63</sup> to 2 <sup>63</sup> -1 |

## Floating Point – float and double

You can declare a floating-point variable using the keywords `float` or `double`. The following list contains examples of floating-point numbers. A numeric literal is a floating point if it includes either a decimal point or an exponent part (the letter E or e), or is followed by the letter F or f (float) or the letter D or d (double). Some examples of floating-point literals include:

|             |                                                    |
|-------------|----------------------------------------------------|
| 3.14        | A simple floating-point value (a double)           |
| 6.02E23     | A large floating-point value                       |
| 2.718F      | A simple <code>float</code> size value             |
| 123.4E+306D | A large <code>double</code> value with redundant D |

**Note** – The 23 after the E in the second example is implicitly positive. That example is equivalent to 6.02E+23.



Floating point literals are `double` by default. You can declare a literal of type `float` by appending F or f to the value.

The format of a floating point number is defined by *The Java Language Specification* to be Institute of Electrical and Electronics Engineers (IEEE) 754, using the sizes shown in Table 3-3. This format is platform independent.

**Table 3-3** Floating Point Data Type Size

| Float Length | Name or Type        |
|--------------|---------------------|
| 32 bits      | <code>float</code>  |
| 64 bits      | <code>double</code> |

**Note** – Floating point literals are `double` unless declared explicitly as `float`.



# Variables, Declarations, and Assignments

The following program illustrates how to declare and assign values to int, float, boolean, char, and String type variables:

```

1 public class Assign {
2 public static void main (String args []) {
3 // declare integer variables
4 int x, y;
5 // declare and assign floating point
6 float z = 3.414f;
7 // declare and assign double
8 double w = 3.1415;
9 // declare and assign boolean
10 boolean truth = true;
11 // declare character variable
12 char c;
13 // declare String variable
14 String str;
15 // declare and assign String variable
16 String str1 = "bye";
17 // assign value to char variable
18 c = 'A';
19 // assign value to String variable
20 str = "Hi out there!";
21 // assign values to int variables
22 x = 6;
23 y = 1000;
24 }
25 }
```

The following are examples of illegal assignments:

```

y = 3.1415926;
// 3.1415926 is not an int
// It requires casting and decimal will be truncated.

w = 175,000;
// The comma symbol (,) cannot appear;

truth = 1;
// this is a common mistake made by ex-C / C++ programmers

z = 3.14156;
// Can't fit double into a float. This requires casting.
```

## Java Reference Types

As you have seen, there are eight primitive Java types: boolean, char, byte, short, int, long, float, and double. All other types refer to objects rather than primitives. Variables that refer to objects are *reference variables*. For example, you can define a class MyDate:

```
1 public class MyDate {
2 private int day = 1;
3 private int month = 1;
4 private int year = 2000;
5 public MyDate(int day, int month, int year) { ... }
6 public String toString() { ... }
7 }
```

The following is an example of using MyDate:

```
1 public class TestMyDate {
2 public static void main(String[] args) {
3 MyDate today = new MyDate(22, 7, 1964);
4 }
5 }
```

The variable today is a reference variable holding one MyDate object.

# Constructing and Initializing Objects

You have seen how you must execute a call to `new XYZ()` to allocate space for a new object. You will see that sometimes you can place arguments in the parentheses, for example  
`new MyDate(22, 7, 1964)`.

Using the keyword `new` causes the following:

1. First, the space for the new object is allocated and initialized to the form of 0 or null. In the Java programming language, this phase is indivisible to ensure that you cannot have an object with random values in it.
2. Second, any explicit initialization is performed.
3. Third, a *constructor*, which is a special method, is executed. Arguments passed in the parentheses to `new` are passed to the constructor (22, 7, 1964).
4. Finally, the return value from the `new` operation is a reference to the new object in heap memory. This reference is stored in the reference variable.

## Memory Allocation and Layout

In a method body, the following declaration allocates storage only for the reference shown in Figure 3-1:

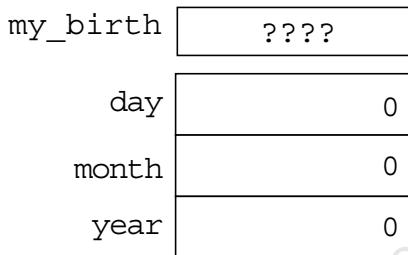
```
MyDate my_birth = new MyDate(22, 7, 1964);
```

|          |      |
|----------|------|
| my_birth | ???? |
|----------|------|

**Figure 3-1** Declaration of the Object Reference

The keyword new in the following example implies allocation and initialization of storage, as shown in Figure 3-2.

```
MyDate my_birth = new MyDate(22, 7, 1964);
```



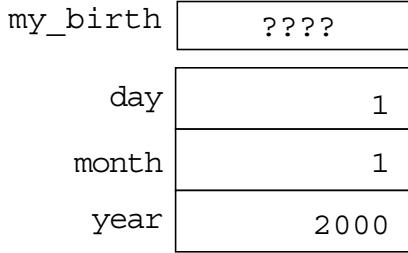
**Figure 3-2** Using the new Keyword

## Explicit Attribute Initialization

If you place simple assignment expressions in your member declarations, you can initialize members explicitly during construction of your object.

In the MyDate class in this example, initializing all three attributes explicitly is declared, as shown in Figure 3-3:

```
MyDate my_birth = new MyDate(22, 7, 1964);
```



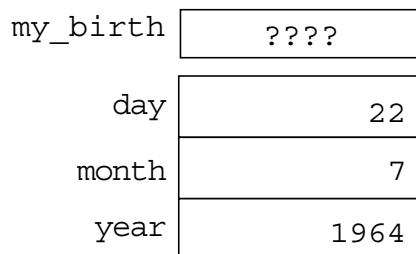
**Figure 3-3** Explicit Initialization

## Executing the Constructor

The final stage of initializing a new object is to call the constructor. The constructor enables you to override the default initialization. You can perform computations. You can also pass arguments into the construction process so that the code that requests the construction of the new object can control the object it creates.

The following example calls the constructor, as shown in Figure 3-4:

```
MyDate my_birth = new MyDate(22, 7, 1964);
```

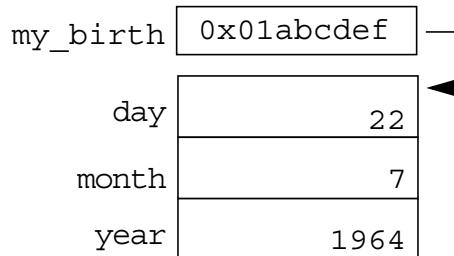


**Figure 3-4** Constructor Call

## Assigning a Variable

The variable assignment then sets up the reference variable `my_birth` in this example, so that it refers properly to the newly created object as shown in Figure 3-5.

```
MyDate my_birth = new MyDate(22, 7, 1964);
```



**Figure 3-5** Reference Variable Assignment

## This Is Not the Whole Story

It turns out that constructing and initializing objects is more complex than is described here. This topic is revisited in Module 6.

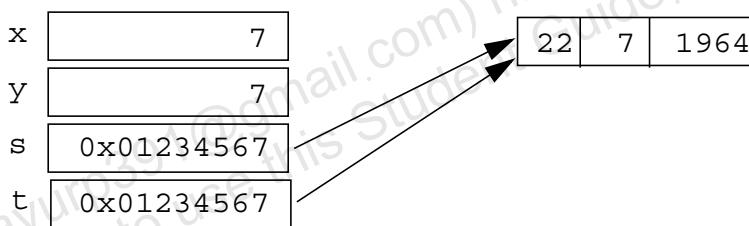
# Assigning References

In the Java programming language, a variable declared with a type of class is referred to as a reference type because it refers to a non-primitive type. This has consequences for the meaning of the assignment. Consider this code fragment:

```
int x = 7;
int y = x;
MyDate s = new MyDate(22, 7, 1964);
MyDate t = s;
```

Four variables are created: two primitives of type `int` and two references of type `MyDate`. The value of `x` is 7, and this value is copied into `y`. Both `x` and `y` are independent variables, and further changes to either do not affect the other.

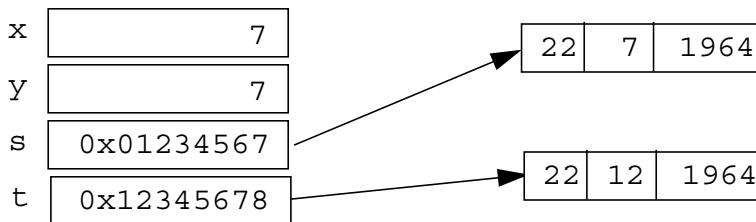
With the variables `s` and `t`, only one `MyDate` object exists, and it contains the date 22 July 1964. Both `s` and `t` refer to that single object, as shown in Figure 3-6. This scenario is depicted as:



**Figure 3-6** Two Variables Refer to the Same Reference Object

With a reassignment of the variable `t`, the new `MyDate` object (for 22 December 1964) is created and `t` refers to this object, as shown in Figure 3-7. This scenario is depicted as:

```
t = new MyDate(22, 12, 1964); // reassign the variable
```



**Figure 3-7** Variable `t` Refers to a New Object

## Pass-by-Value

The Java programming language passes arguments only *by value*, that is, you cannot change the argument value in the calling method from within the called method.

However, when an object instance is passed as an argument to a method, the value of the argument is not the object itself but a reference to the object. You can change the *contents* of the object in the called method but not the object reference.

To many people, this looks like pass-by-reference, and behaviorally, it has much in common with pass-by-reference. However, there are two reasons this is inaccurate. First, the ability to change the *thing* passed into a method only applies to objects, not primitive values. Second, the actual value associated with a variable of object type is the reference to the object, and not the object itself. This is an important distinction in other ways, and if clearly understood, is entirely supporting of the point that the Java programming language passes arguments by value.

The following code example illustrates this point:

```
1 public class PassTest {
2
3 // Methods to change the current values
4 public static void changeInt(int value) {
5 value = 55;
6 }
7 public static void changeObjectRef(MyDate ref) {
8 ref = new MyDate(1, 1, 2000);
9 }
10 public static void changeObjectAttr(MyDate ref) {
11 ref.setDay(4);
12 }
13
14 public static void main(String args[]) {
15 MyDate date;
16 int val;
17
18 // Assign the int
19 val = 11;
20 // Try to change it
```

```
21 changeInt(val);
22 // What is the current value?
23 System.out.println("Int value is: " + val);
24
25 // Assign the date
26 date = new MyDate(22, 7, 1964);
27 // Try to change it
28 changeObjectRef(date);
29 // What is the current value?
30 System.out.println("MyDate: " + date);
31
32 // Now change the day attribute
33 // through the object reference
34 changeObjectAttr(date);
35 // What is the current value?
36 System.out.println("MyDate: " + date);
37 }
38 }
```

This code outputs the following:

```
java PassTest
```

```
Int value is: 11
MyDate: 22-7-1964
MyDate: 4-7-1964
```

The **MyDate** object is not changed by the **changeObjectRef** method; however, the **changeObjectAttr** method changes the day attribute of the **MyDate** object.

# The this Reference

Two uses of the `this` keyword are:

- To resolve ambiguity between instance variables and parameters
- To pass the current object as a parameter to another method

Code 3-1 provides a class definition that demonstrates these uses. The `MyDate` class declares instance variables, Lines 2–4. One of the parameters to one of the constructors (Lines 6–10) is also called `day`, so in that context, the keyword `this` resolves the ambiguity (Line 7). The `addDays` method creates a new date object (Line 18). In this constructor call, the method uses the `this` keyword as an argument to refer to the current object.

## Code 3-1 The Use of the `this` Keyword

```

1 public class MyDate {
2 private int day = 1;
3 private int month = 1;
4 private int year = 2000;
5
6 public MyDate(int day, int month, int year) {
7 this.day = day;
8 this.month = month;
9 this.year = year;
10 }
11 public MyDate(MyDate date) {
12 this.day = date.day;
13 this.month = date.month;
14 this.year = date.year;
15 }
16
17 public MyDate addDays(int moreDays) {
18 MyDate newDate = new MyDate(this);
19 newDate.day = newDate.day + moreDays;
20 // Not Yet Implemented: wrap around code...
21 return newDate;
22 }
23 public String toString() {
24 return "" + day + "-" + month + "-" + year;
25 }
26 }
```

# Java Programming Language Coding Conventions

The following are the coding conventions of the Java programming language:

- **Packages** – Package names should be nouns in lowercase.  
`package shipping.objects`
- **Classes** – Class names should be nouns, in mixed case, with the first letter of each word capitalized.  
`class AccountBook`
- **Interfaces** – Interface names should be capitalized like class names.  
`interface Account`
- **Methods** – Method names should be verbs, in mixed case, with the first letter in lowercase. Within each method name, capital letters separate words. Limit the use of underscores.  
`balanceAccount()`
- **Variables** – All variables should be in mixed case with a lowercase first letter. Words are separated by capital letters. Limit the use of underscores, and avoid using the dollar sign (\$) because this character has special meaning to inner classes.  
`currentCustomer`  
Variables should be meaningful and indicate to the casual reader the intent of their use. Avoid single character names except for temporary *throwaway* variables (for example, i, j, and k, used as loop control variables).
- **Constants** – Primitive constants should be all uppercase with the words separated by underscores. Object constants can use mixed-case letters.  
`HEAD_COUNT`  
`MAXIMUM_SIZE`
- **Control structures** – Use braces ({ }) around all statements, even single statements, when they are part of a control structure, such as an if-else or for statement.  
`if ( condition ) {  
 statement1;  
} else {  
 statement2;  
}`

## Java Programming Language Coding Conventions

- **Spacing** – Place only a single statement on any line, and use two-space or four-space indentations to make your code readable. The number of spaces can vary depending on what code standards you use.
- **Comments** – Use comments to explain code segments that are not obvious. Use the // comment delimiter for normal commenting; you can comment large sections of code using the /\* . . . \*/ delimiters. Use the /\*\* . . . \*/ documenting comment to provide input to javadoc for generating HTML documentation for the code.

```
// A comment that takes up only one line.
```

```
/* Comments that continue past one line and take up
space on multiple lines. */
```

```
/** A comment for documentation purposes.
 * @see Another class for more information
 */
```

---

**Note** – The @see tag is a special javadoc tag giving the effect of a *see also* link that references a class or method. For more information about javadoc, refer to the complete definition of the documentation system in *The Design of Distributed Hyperlinked Programming Documentation*, a paper by Lisa Friendly. It is available at the following URL:  
<http://java.sun.com/docs/javadoc-paper.html>

Also, for more information on Sun's Java technology coding conventions, refer to the following Web page:

<http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>

---



## Module 4

---

# Expressions and Flow Control

---

## Objectives

Upon completion of this module, you should be able to:

- Distinguish between instance and local variables
- Describe how to initialize instance variables
- Identify and correct a Possible reference before assignment compiler error
- Recognize, describe, and use Java software operators
- Distinguish between legal and illegal assignments of primitive types
- Identify boolean expressions and their requirements in control constructs
- Recognize assignment compatibility and required casts in fundamental types
- Use if, switch, for, while, and do constructions and the labeled forms of break and continue as flow control structures in a program

This module describes variables, operators, and arithmetic expressions, and lays out the different control structures governing the path of execution.

---

## Relevance



# Relevance

**Discussion** – The following questions are relevant to the material presented in this module:

- What types of variables are useful to programmers?

---

---

---

- Can multiple classes have variables with the same name and, if so, what is their scope?

---

---

---

- What types of control structures are used in other languages? What methods do these languages use to control flow (such as in a loop or switch)?

---

---

---

# Variables

This section describes variables are declared, scoped, and initialized in the Java programming language.

## Variables and Scope

You have seen two ways to describe variables: variables of primitive type or variables of reference type. You have also seen two places to declare variables: inside a method or outside a method but within a *class* definition.

Variables defined inside a method are called *local* variables, but are sometimes referred to as *automatic*, *temporary*, or *stack* variables. You must initialize local variables explicitly before the first use. Method parameters and constructor parameters are also local variables but they are initialized by the calling code.

Variables defined outside a method are created when the object is constructed using the `new Xxx()` call. There are two possible types of these variables. The first kind is a class variable that is declared using the `static` keyword. Variables labeled `static` are created when the class is loaded and continue to exist for as long as the class is loaded. The second type is an instance variable that is declared without the `static` keyword. Instance variables continue to exist for as long as the object exists. Instance variables are sometimes referred to as member variables, because they are members of objects created from the class. The `static` variable is described later in this course in more detail. Both member variables and class variables are initialized automatically when they are created.

Method parameter variables define arguments passed in a method call. Each time the method is called, a new variable is created and it lasts only until the method is exited.

Local variables are created when execution enters the method and are destroyed when the method is exited. This is why local variables are sometimes referred to as *temporary* or *automatic*. Variables that are defined within a member function are local to that member function, so you can use the same variable name in several member functions to refer to different variables. This is illustrated in the example on Code 4-1 on page 4-3.

### Code 4-1 Variable Scope Example

## Variables

```

1 public class ScopeExample {
2 private int i=1;
3
4 public void firstMethod() {
5 int i=4, j=5;
6 this.i = i + j;
7 secondMethod(7);
8 }
9 public void secondMethod(int i) {
10 int j=8;
11 this.i = i + j;
12 }
13 }
```

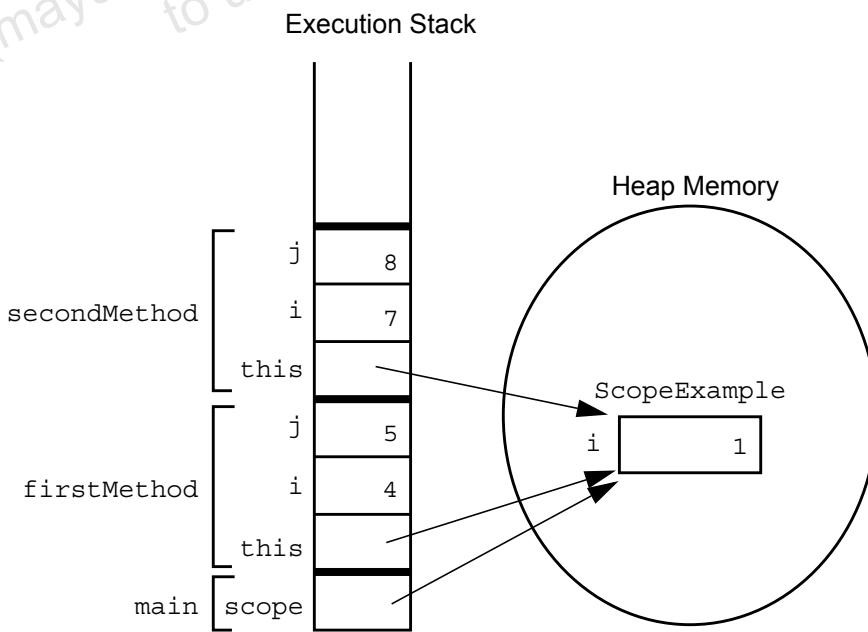
  

```

1 public class TestScoping {
2 public static void main(String[] args) {
3 ScopeExample scope = new ScopeExample();
4 scope.firstMethod();
5 }
6 }
```

**Note** – This code serves to exemplify scoping rules *only*. Reusing names in this manner is not a good practice.

Figure 4-1 shows a visualization of the variables in Code 4-1.



**Figure 4-1** Diagram of Variable Scoping for Code 4-1

## Variable Initialization

You must initialize a variable in a Java program before you can use it. For stack variables it is easy for the compiler to check if the code assigns a value to such a variable before it attempts to use that value: The `this` reference and the method parameters have assigned values when the method execution starts. Local variables that are defined within the method code can be checked by looking at the method code only. So the compiler enforces that the method code cannot read a variable value before we have assigned one.

For object attributes on the heap, such a check is not easily possible: How can the compiler know whether a client of our class calls a setter method on an object before maybe some other client wants to call a getter method on it? So the only reasonable way to handle this problem is to enforce assigning an initial value already in the constructor. If the programmer does not provide a default value, the compiler will assign the default value. Table 4-1 lists the default values for primitive and reference instance variables.

**Table 4-1** Default Values of Primitive Types

| Variable            | Value    |
|---------------------|----------|
| byte                | 0        |
| short               | 0        |
| int                 | 0        |
| long                | 0L       |
| float               | 0.0F     |
| double              | 0.0D     |
| char                | '\u0000' |
| boolean             | false    |
| All reference types | null     |



**Note** – A reference that has the `null` value does not refer to an object. An attempt to use the object it refers to causes an exception. Exceptions are errors that occur at runtime and are described in a later module.

## Variables

---

### Initialization Before Use Principle

While variables defined outside of a method are initialized automatically, local variables *must* be initialized manually before use. The compiler flags an error if it can determine a condition where a variable can be used before being initialized.

```
public void doComputation() {
 int x = (int) (Math.random() * 100);
 int y;
 int z;
 if (x > 50) {
 y = 9;
 }
 z = y + x; // Possible use before initialization
}
```

# Operators

This section describes the operators in the Java programming language.

## Operator Precedence

The Java programming language operators are similar in style and function to those of C and C++. Table 4-2 lists the operators in order of precedence (L to R means left-to-right associative; R to L means right-to-left associative).

**Table 4-2 Operators and Precedence**

| Associative | Operators                                                                                              |
|-------------|--------------------------------------------------------------------------------------------------------|
| R to L      | <code>++ -- + - ~ ! (&lt;data_type&gt;)</code>                                                         |
| L to R      | <code>*</code> <code>/</code> <code>%</code>                                                           |
| L to R      | <code>+</code> <code>-</code>                                                                          |
| L to R      | <code>&lt;&lt;</code> <code>&gt;&gt;</code> <code>&gt;&gt;&gt;</code>                                  |
| L to R      | <code>&lt;</code> <code>&gt;</code> <code>&lt;=</code> <code>&gt;= instanceof</code>                   |
| L to R      | <code>==</code> <code>!=</code>                                                                        |
| L to R      | <code>&amp;</code>                                                                                     |
| L to R      | <code>^</code>                                                                                         |
| L to R      | <code> </code>                                                                                         |
| L to R      | <code>&amp;&amp;</code>                                                                                |
| L to R      | <code>  </code>                                                                                        |
| R to L      | <code>&lt;boolean_expr&gt; ? &lt;expr1&gt; : &lt;expr2&gt;</code>                                      |
| R to L      | <code>= *=% /=% %=% += -= &lt;&lt;=%</code><br><code>&gt;&gt;=% &gt;&gt;&gt;=% &amp;=% ^=%  ==%</code> |

**Note** – The `instanceof` operator is unique to the Java programming language and is described in Module 6.



## Logical Operators

Most Java technology operators are taken from other languages and behave as expected.

Relational and logical operators return a boolean result. The value 0 is not interpreted automatically as false and non-zero values are not interpreted automatically as true.

```
int i = 1;
if (i) // generates a compile error
if (i != 0) // Correct
```

The boolean operators supported are !, &, ^, and | for the algebraic Boolean operations NOT, AND, XOR, and OR, respectively. Each of these returns a boolean result. The operators && and || are the short-circuit equivalents of the operators & and |.

### Short-Circuit Logical Operators

The operators && (defined as AND) and || (defined as OR) perform *short-circuit* logical expressions. Consider this example:

```
MyDate d = reservation.getDepartureDate();
if ((d != null) && (d.day > 31)) {
 // do something with d
}
```

The boolean expression that forms the argument to the if () statement is legal and entirely safe. This is because the second subexpression is skipped when the first subexpression is false, because the entire expression is always false when the first subexpression is false, regardless of how the second subexpression evaluates. Similarly, if the || operator is used and the first expression returns true, the second expression is not evaluated because the whole expression is already known to be true.

## Bitwise Logical Operators

Bit manipulation operations, including logical and shift operations, perform low-level operations directly on the binary representations used in integers. These operations are not often used in enterprise-type systems but might be critical in graphical, scientific, or control systems. The ability to operate directly on binary might save large amounts of memory, might enable certain computations to be performed very efficiently, and can greatly simplify operations on collections of bits, such as data read from or written to parallel I/O ports.

The Java programming language supports bitwise operations on integral data types. These are represented as the operators `~`, `&`, `^`, and `|` for the bitwise operations of NOT (bitwise complement), bitwise AND, bitwise XOR, and bitwise OR, respectively.

Figure 4-2 shows examples of the bitwise operators on byte-sized binary numbers.

|                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                          |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\begin{array}{r} \sim \\ \hline \boxed{0} \boxed{1} \boxed{0} \boxed{0} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \\ \hline \boxed{1} \boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{0} \boxed{0} \boxed{0} \end{array}$                                                                                 | $\begin{array}{r} \& \\ \hline \boxed{0} \boxed{0} \boxed{1} \boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{1} \\ \boxed{0} \boxed{1} \boxed{0} \boxed{0} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \\ \hline \boxed{0} \boxed{0} \boxed{0} \boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{1} \end{array}$ |
| $\begin{array}{r} ^ \\ \hline \boxed{0} \boxed{0} \boxed{1} \boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{1} \\ \boxed{0} \boxed{1} \boxed{0} \boxed{0} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \\ \hline \boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{0} \boxed{0} \boxed{1} \boxed{0} \end{array}$ | $\begin{array}{r}   \\ \hline \boxed{0} \boxed{0} \boxed{1} \boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{1} \\ \boxed{0} \boxed{1} \boxed{0} \boxed{0} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \\ \hline \boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \end{array}$  |

**Figure 4-2** Examples of the Bitwise Operators

## Right-Shift Operators `>>` and `>>>`

The Java programming language provides two right-shift operators. The operator `>>` performs an *arithmetic* or *signed* right shift. The result of this shift is that the first operand is divided by 2 raised to the number of times specified by the second operand. For example:

```
128 >> 1 returns 128/21 = 64
256 >> 4 returns 256/24 = 16
-256 >> 4 returns -256/24 = -16
```

The `>>` operator results in the sign bit being copied during the shift.

The *logical* or *unsigned* right shift operator `>>>` works on the bit pattern rather than the arithmetic meaning of a value and always places 0s in the most significant bits; for example:

```
1010 ... >> 2 gives 111010 ...
1010 ... >>> 2 gives 001010 ...
```

## Left-Shift Operator `<<`

The operator `<<` performs a left shift. The result of this shift is that the first operand is multiplied by two raised to the number specified by the second operand; for example:

```
128 << 1 returns 128*21 = 256
16 << 2 returns 16*22 = 64
```

All three shift operators reduce their right-hand operand modulo 32 for an `int` type left-hand operand and modulo 64 for a `long` type left-hand operand. Therefore, for any `int x, x >>> 32` results in `x` being unchanged, not 0 as you might expect.

The shift operators are permitted only on integral types. The unsigned right shift `>>>` is effective only on `int` or `long` values. If you use it on a `short` or `byte` value, the value is promoted, with sign extension, to an `int` before `>>>` is applied. By this point, the unsigned shift usually has become a signed shift.

## Shift Operator Examples

Figure 4-3 show the bit patterns of a positive and a negative number and the bit patterns resulting from the three shift operators: `>>`, `>>>`, and `<<`.

`1357 = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 0 1 1 0 1]`

`-1357 = [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 1 0 0 1 1]`

`1357 >> 5 = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0]`

`-1357 >> 5 = [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 0 1]`

`1357 >>> 5 = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0]`

`-1357 >>> 5 = [0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 0]`

`1357 << 5 = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 1 1 0 1 0 0 0 0 0 0]`

`-1357 << 5 = [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0]`

**Figure 4-3 Examples of the Shift Operators**

**Note** – The code that generated these examples (including printing out the complete bit pattern) can be found in the file `examples/mod04_stmts/TestShift.java`.



## String Concatenation With +

The + operator performs a concatenation of String objects, producing a new String.

```
String salutation = "Dr. ";
String name = "Pete" + " " + "Seymour";
String title = salutation + " " + name;
```

The result of the last line is:

Dr. Pete Seymour

If either argument of the + operator is a String object, then the other argument is converted to a String object. All objects can be converted to a String object automatically, although the result might be rather cryptic. The object that is not a String object is converted to a string equivalent using the `toString()` member function.

## Casting

*Casting* means assigning a value of one type to a variable of another type. If the two types are compatible, the Java software performs the conversion automatically. For example, an `int` value can always be assigned to a `long` variable.

Where information might be lost in an assignment, the compiler requires that you confirm the assignment with a cast. For example, *squeeze* a `long` value into an `int` variable like this:

```
long bigValue = 99L;
int squashed = bigValue; // Wrong, needs a cast
int squashed = (int) bigValue; // OK

int squashed = 99L; // Wrong, needs a cast
int squashed = (int) 99L; // OK, but...
int squashed = 99; // default integer literal
```

The desired target type is placed in parentheses and used as a prefix to the expression that must be modified. Although it might not be necessary, it is advisable to enclose the entire expression to be cast in parentheses. Otherwise, the precedence of the cast operation can cause problems.

---

**Note – Reference type variables can also be cast; see Module 6.**

---



## Promotion and Casting of Expressions

Variables can be promoted automatically to a longer form (such as `int` to `long`) when there would be no loss of information.

```
long bigval = 6; // 6 is an int type, OK
int smallval = 99L; // 99L is a long, illegal
```

```
double z = 12.414F; // 12.414F is float, OK
float z1 = 12.414; // 12.414 is double, illegal
```

In general, you can think of an expression as being *assignment-compatible* if the variable type is at least as large (maximum value) as the expression type.

For binary operators, such as the `+` operator, when the two operands are of primitive numeric types, the result type is determined as the largest type of either operand, or `int`. Therefore, all binary operations on numeric types result in at least an `int` result, and possibly a larger one if `long`, `float`, or `double` operands are in the expression. This might result in overflow or loss of precision when the result is assigned to a variable.

For example, the following code fragment:

```
short a, b, c;
a = 1;
b = 2;
c = a + b;
```

causes an error because it raises each `short` to an `int` before operating on it. However, if `c` is declared as an `int`, or a cast is done as:

```
c = (short)(a + b);
```

then the code works.

# Branching Statements

Conditional statements enable the selective execution of portions of the program according to the value of some expressions. The Java programming language supports the `if` and `switch` statements for two-way and multiple-way branching, respectively.

## Simple if, else Statements

The basic syntax for an `if` statement is:

```
if (<boolean_expression>)
 <statement_or_block>
```

For example:

```
if (x < 10)
 System.out.println("Are you finished yet?");
```

However, it is recommended that you place all then statements into a block. For example:

```
if (x < 10) {
 System.out.println("Are you finished yet?");
}
```

## Complex if, else Statements

If you require an `else` clause, then you must use the `if-else` statement:

```
if (<boolean_expression>)
 <statement_or_block>
else
 <statement_or_block>
```

For example:

```
if (x < 10) {
 System.out.println("Are you finished yet?");
} else {
 System.out.println("Keep working... ");
}
```

If you require a series of conditional checks, then you can chain a sequence of if-else-if statements:

```
if (<boolean_expression>)
 <statement_or_block>
else if (<boolean_expression>)
 <statement_or_block>
```

For example:

```
int count = getCount(); // a method defined in the class
if (count < 0) {
 System.out.println("Error: count value is negative.");
} else if (count > getMaxCount()) {
 System.out.println("Error: count value is too big.");
} else {
 System.out.println("There will be " + count +
 " people for lunch today.");
}
```

The Java programming language differs from C and C++ because an if() statement takes a boolean expression, not a numeric value. You cannot convert or cast boolean types and numeric types. If you have:

```
if (x) // x is int
```

use:

```
if (x != 0)
```

The entire else part of the statement is optional, and you can omit it if no action is to be taken when the tested condition is false.

## The switch Statement

The switch statement syntax is:

```
switch (<expression>) {
 case <constant1>:
 <statement_or_block>*
 [break;]
 case <constant2>:
 <statement_or_block>*
 [break;]
 default:
 <statement_or_block>*
 [break;]
}
```

In the `switch (<expression>)` statement, `<expression>` must be expression-compatible with an `int` type. Promotion occurs with `byte`, `short`, or `char` types. Floating point, `long` expressions, or object references (including `String`s) are not permitted.

---

**Note** – An enumerated type value may also be used in the `<expression>` and `<constantN>` elements. See Enumerated Types in Module 7.

---

The optional `default` label specifies the code segment to be executed when the value of the variable or expression cannot match any of the `case` values. If there is no `break` statement as the last statement in the code segment for a certain `case`, the execution continues into the code segment for the next `case` without checking the `case` expression's value.

A `return` statement can be used instead of a `break` statement. Moreover, if the `switch` is in a loop, then a `continue` statement would also cause execution to exit out of the `switch` construct.

---

**Note** – Nine out of ten `switch` statements required `breaks` in each `case` block. Forgetting the `break` statement causes the most programming errors when using `switch` statements.

---

An example `switch` statement is shown in Code 4-2 on page 4-17.



**Code 4-2** The switch Statement Example 1

```

switch (carModel) {
 case DELUXE:
 addAirConditioning();
 addRadio();
 addWheels();
 addEngine();
 break;
 case STANDARD:
 addRadio();
 addWheels();
 addEngine();
 break;
 default:
 addWheels();
 addEngine();
}

```

Code 4-2 configures a car object based on the `carModel`. If `carModel` is the integer constant `DELUXE`, then air conditioning is added to the car, as is a radio, and, of course, wheels and an engine. However, if the `carModel` is only a `STANDARD`, then only a radio, wheels, and an engine are added. Finally, by default, any car model will have wheels and an engine added.

A second example switch statement is shown in Code 4-3.

**Code 4-3** The switch Statement Example 2

```

switch (carModel) {
 case DELUXE:
 addAirConditioning();
 case STANDARD:
 addRadio();
 default:
 addWheels();
 addEngine();
}

```

Code 4-3 solves the redundant method calls in the previous example by permitting the flow of control to descend through multiple case blocks. For example, if the `carModel` is `DELUXE`, then the `addAirConditioning` method is called, and then the flow of control falls through the next case statement and calls the `addRadio` method, and finally the flow of control falls through the `default` statement and calls the `addWheels` and `addEngine` methods.

# Looping Statements

Looping statements enable you to execute blocks of statements repeatedly. The Java programming language supports three types of loop constructs: `for`, `while`, and `do` loops. The `for` and `while` loops test the loop condition before executing the loop body; the `do` loops check the loop condition after executing the loop body. This implies that the `for` and `while` loops might not execute the loop body even once, whereas `do` loops execute the loop body at least once.

## The `for` Loops

The `for` loop syntax is:

```
for (<init_expr>; <test_expr>; <alter_expr>)
 <statement_or_block>
```

For example:

```
for (int i = 0; i < 10; i++)
 System.out.println(i + " squared is " + (i*i));
```

However, it is recommended that you place all loop-clause statements into a block. For example:

```
for (int i = 0; i < 10; i++) {
 System.out.println(i + " squared is " + (i*i));
}
```

In the previous example, `int i` is declared and defined within the `for` block. The variable `i` is accessible only within the scope of this particular `for` block.

---

**Note** – The Java programming language permits the comma separator in a `for()` loop structure. For example,

`for (i = 0, j = 0; j < 10; i++, j++) { }` is legal, and it initializes both `i` and `j` to 0, and increments both `i` and `j` after executing the loop body.

---



## The `while` Loop

The `while` loop syntax is:

---

```
while (<test_expr>)
 <statement_or_block>
```

For example:

```
int i = 0;
while (i < 10) {
 System.out.println(i + " squared is " + (i*i));
 i++;
}
```

Ensure that the loop-control variable is initialized appropriately before the loop body begins execution. You must update the control variable appropriately to prevent an infinite loop.

## The do/while Loop

The syntax for the do/while loop is:

```
do
 <statement_or_block>
while (<test_expr>);
```

For example:

```
int i = 0;
do {
 System.out.println(i + " squared is " + (i*i));
 i++;
} while (i < 10);
```

As with the while loops, ensure that the loop-control variable is initialized appropriately, updated in the body of the loop, and tested properly.

Use the for loop in cases where the loop is to be executed a predetermined number of times. Use the while and do loops in cases where this is not determined beforehand.

## Special Loop Flow Control

You can use the following statements to further control loop statements:

- break [<label>];

## Looping Statements

---

Use the **break** statement to prematurely exit from **switch** statements, loop statements, and labeled blocks.

- `continue [<label>];`

Use the **continue** statement to skip over and jump to the end of the loop body, and then return control to the loop-control statement.

- `<label> : <statement>`

The **label** statement identifies any valid statement to which control must be transferred. With regard to a labeled **break** statement, the label can identify any statement. With regard to a labeled **continue** statement, the label must identify a loop construct.

### The break Statement

Here is an example loop with an unlabeled **break** statement:

```
1 do {
2 statement;
3 if (condition) {
4 break;
5 }
6 statement;
7 } while (test_expr);
```

### The continue Statement

Here is an example loop with an unlabeled **continue** statement:

```
1 do {
2 statement;
3 if (condition) {
4 continue;
5 }
6 statement;
7 } while (test_expr);
```

## Using break Statements with Labels

Here is an example loop with a labelled break statement:

```
1 outer:
2 do {
3 statement1;
4 do {
5 statement2;
6 if (condition) {
7 break outer;
8 }
9 statement3;
10 } while (test_expr);
11 statement4;
12 } while (test_expr);
```

## Using continue Statements with Labels

Here is an example loop with a labelled continue statement:

```
1 test:
2 do {
3 statement1;
4 do {
5 statement2;
6 if (condition) {
7 continue test;
8 }
9 statement3;
10 } while (test_expr);
11 statement4;
12 } while (test_expr);
```



## Module 5

---

# Arrays

---

## Objectives

Upon completion of this module, you should be able to:

- Declare and create arrays of primitive, class, or array types
- Explain why elements of an array are initialized
- Explain how to initialize the elements of an array
- Determine the number of elements in an array
- Create a multidimensional array
- Write code to copy array values from one array to another

This module describes how to define, initialize, and use arrays in the Java programming language.

---

## Relevance

# Relevance

**Discussion** – The following question is relevant to the material presented in this module:

What is the purpose of an array?



---

---

---

## Declaring Arrays

Arrays are used typically to group objects of the same type. Arrays enable you to refer to the group of objects by a common name.

You can declare arrays of any type, either primitive or class:

```
char[] s;
Point[] p; // where Point is a class
```

When declaring arrays with the brackets to the left, the brackets apply to all variables to the right of the brackets.

In the Java programming language, an array is an object even when the array is made up of primitive types, and as with other class types, the declaration does not create the object itself. Instead, the declaration of an array creates a reference that you can use to refer to an array. The actual memory used by the array elements is allocated dynamically either by a new statement or by an array initializer.

You can declare arrays using the square brackets after the variable name:

```
char s[];
Point p[];
```

This is standard for C, C++, and the Java programming language. This format leads to complex forms of declaration that can be difficult to read.

The result is that you can consider a declaration as having the type part on the left, and the variable name on the right. You will see both formats used, but you should decide on one or the other for your own use. The declarations do not specify the actual size of the array.

## Creating Arrays

# Creating Arrays

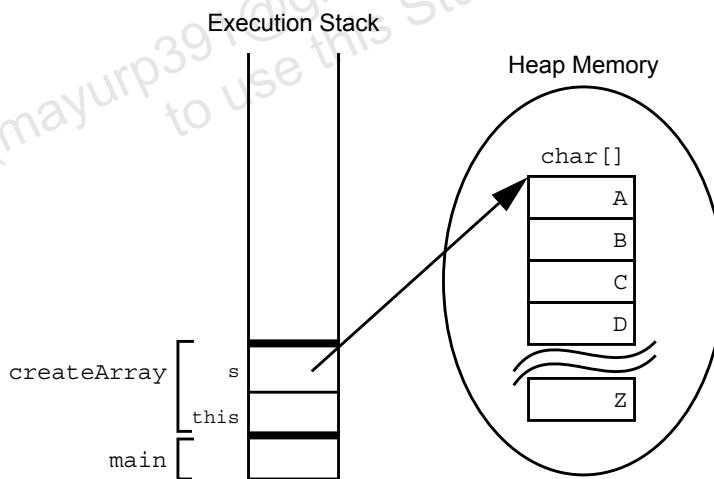
You can create arrays, like all objects, using the new keyword. For example, to create an array of a primitive (char) type:

```
s = new char[26];
```

The first line creates an array of 26 char values. After creation, the array elements are initialized to the default value ('\u0000' for characters). You must fill in the array for it to be useful; for example:

```
1 public char[] createArray() {
2 char[] s;
3
4 s = new char[26];
5 for (int i=0; i<26; i++) {
6 s[i] = (char) ('A' + i);
7 }
8
9 return s;
10 }
```

This code generates an array in the heap memory with the upper case letters of the English alphabet. The array in heap is shown in Figure 5-1.



**Figure 5-1** Creating an Array of Character Primitives

The subscript that indexes the individual array elements always begins from 0 and must be maintained in the legal range: greater than or equal to 0 and less than the array length. Any attempt to access an array element outside these bounds causes a runtime exception.

## Creating Reference Arrays

You can create arrays of objects. You use the same syntax:

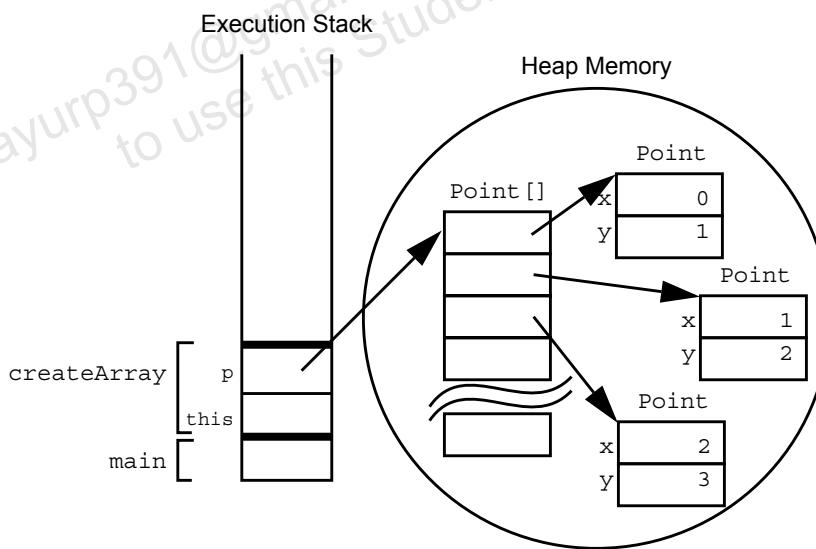
```
p = new Point[10];
```

This line creates an array of 10 references of type Point. However, it does not create 10 Point objects. Create these separately as follows:

```

1 public Point[] createArray() {
2 Point[] p;
3
4 p = new Point[10];
5 for (int i=0; i<10; i++) {
6 p[i] = new Point(i, i+1);
7 }
8
9 return p;
10 }
```

This code generates an array in the heap memory with each array element filled with a reference to a Point object. The array in heap is shown in Figure 5-2.



**Figure 5-2** Creating an Array of Character Primitives With Point Objects

## Initializing Arrays

When you create an array, every element is initialized. In the case of the `char` array `s` in the previous section, each value is initialized to the null ('`\u0000`') character. In the case of the array `p`, each value is initialized to `null`, indicating that it does not (yet) refer to a `Point` object. After the assignment `p[0] = new Point()`, the first element of the array refers to a real `Point` object.

---

**Note** – Initializing all variables, including elements of arrays, is essential to the security of the system. You must not use variables in an uninitialized state.

---

The Java programming language allows a shorthand that creates arrays with initial values:

```
String[] names = {
 "Georgianna",
 "Jen",
 "Simon"
};
```

This code is equivalent to:

```
String[] names;
names = new String[3];
names[0] = "Georgianna";
names[1] = "Jen";
names[2] = "Simon";
```

You can use this shorthand for any element type; for example:

```
MyDate[] dates = {
 new MyDate(22, 7, 1964),
 new MyDate(1, 1, 2000),
 new MyDate(22, 12, 1964)
};
```



## Multidimensional Arrays

The Java programming language does not provide multidimensional arrays in the same way that other languages do. Because you can declare an array to have any base type, you can create arrays of arrays (and arrays of arrays of arrays, and so on). The following example shows a two-dimensional array:

```
int [] [] twoDim = new int [4] [] ;
twoDim[0] = new int [5] ;
twoDim[1] = new int [5] ;
```

The object that is created by the first call to new is an array that contains four elements. Each element is a null reference to an element of type array of int and you must initialize each element separately so that each element points to its array.



**Note** – Although the declaration format allows the square brackets to be at the left or right of the variable name, this flexibility does not carry over to other aspects of the array syntax. For example, new int [] [4] is not legal.

Because of this separation, you can create non-rectangular arrays of arrays. That is, you can initialize the elements of twoDim as follows:

```
twoDim[0] = new int [2] ;
twoDim[1] = new int [4] ;
twoDim[2] = new int [6] ;
twoDim[3] = new int [8] ;
```

Because this type of initialization is tedious, and the rectangular array of arrays is the most common form, there is a shorthand to create two-dimensional arrays. For example, you can use the following to create an array of four arrays of five integers each:

```
int [] [] twoDim = new int [4] [5] ;
```

# Array Bounds

In the Java programming language, all array indices begin at 0. The number of elements in an array is stored as part of the array object in the `length` attribute. If an out-of-bounds runtime access occurs, then a runtime exception is thrown.

Use the `length` attribute to iterate on an array as shown in Code 5-1.

### Code 5-1 Using the Array Bounds to Iterate Over the Array

```
public void printElements(int[] list) {
 for (int i = 0; i < list.length; i++) {
 System.out.println(list[i]);
 }
}
```

Using the `length` attribute of the array makes program maintenance easier because you do not need to know the number of elements in the array at compile time.

## Using the Enhanced for Loop

Iterating over an array is a very common task. The Java 2 Platform, Standard Edition (J2SE™) version 5.0 has added an enhanced `for` loop to make array iteration easier. This is shown in Code 5-2.

### Code 5-2 Using the Enhanced for Loop to Iterate Over the Array

```
public void printElements(int[] list) {
 for (int element : list) {
 System.out.println(element);
 }
}
```

This version of the `for` loop can be read as `for each element in list do`. The compiler handles the iteration code. Code 5-2 is equivalent to Code 5-1.

## Array Resizing

After it is created, you cannot resize an array. However, you can use the same reference variable to refer to an entirely new array:

```
int [] myArray = new int [6] ;
myArray = new int [10] ;
```

In this case, the first array effectively is lost unless another reference to it is retained elsewhere.

---

## Copying Arrays

The Java programming language provides a special method in the `System` class, `arraycopy()`, to copy arrays. For example, you can use the `arraycopy()` method as follows:

```
1 // original array
2 int [] myArray = { 1, 2, 3, 4, 5, 6 };
3
4 // new larger array
5 int [] hold = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
6
7 // copy all of the myArray array to the hold
8 // array, starting with the 0th index
9 System.arraycopy(myArray, 0, hold, 0,
10 myArray.length);
```

At this point, the array `hold` has the following contents: 1, 2, 3, 4, 5, 6, 4, 3, 2, 1.

---

**Note** – The `System.arraycopy` method copies references, not objects, when dealing with arrays of objects. The objects themselves do not change.

---



## Module 6

---

# Class Design

---

## Objectives

Upon completion of this module, you should be able to:

- Define *inheritance, polymorphism, overloading, overriding, and virtual method invocation*
- Use the access modifiers *protected* and the default (*package-friendly*)
- Describe the concepts of constructor and method overloading
- Describe the complete object construction and initialization operation

This module is the second of three modules that describe the object-oriented paradigm and the object-oriented features of the Java programming language.

---

## Relevance

# Relevance

**Discussion** – The following question is relevant to the material presented in this module:

How does the Java programming language support object inheritance?

---

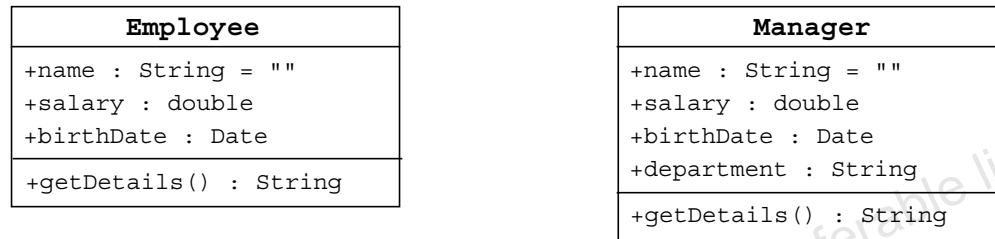
---

---

## Subclassing

In programming, you often create a model of something (for example, an employee), and then need a more specialized version of that original model. For example, you might want a model for a manager. A manager *is an* employee, but an employee with additional features.

Figure 6-1 shows the UML class diagrams that model the Employee and Manager classes.



**Figure 6-1** Class Diagrams for Employee and Manager

Code 6-1 and Code 6-2 show possible implementations of the Employee and Manager classes as they are modeled in Figure 6-1.

### Code 6-1 A Possible Implementation of the Employee Class

```

public class Employee {
 public String name = "";
 public double salary;
 public Date birthDate;

 public String getDetails() { ... }
}

```

### Code 6-2 A Possible Implementation of the Manager Class

```

public class Manager {
 public String name = "";
 public double salary;
 public Date birthDate;
 public String department;

 public String getDetails() { ... }
}

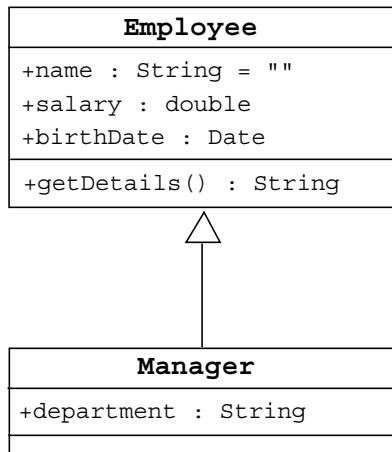
```

## Subclassing

---

This example illustrates the duplication of data between the Manager class and the Employee class. Additionally, there could be a number of methods applicable to both Employee and Manager. Therefore, you need a way to create a new class from an existing class; this is called *subclassing*.

In object-oriented languages, special mechanisms are provided that enable you to define a class in terms of a previously defined class. Figure 6-2 shows the UML class diagram in which the Manager class is a subclass of the Employee class.



**Figure 6-2** Class Diagrams for Employee and Manager Using Inheritance

Code 6-3 shows an implementation of the Manager class that inherits from the Employee class as modeled in Figure 6-2.

### Code 6-3 Another Implementation of the Manager Class

```

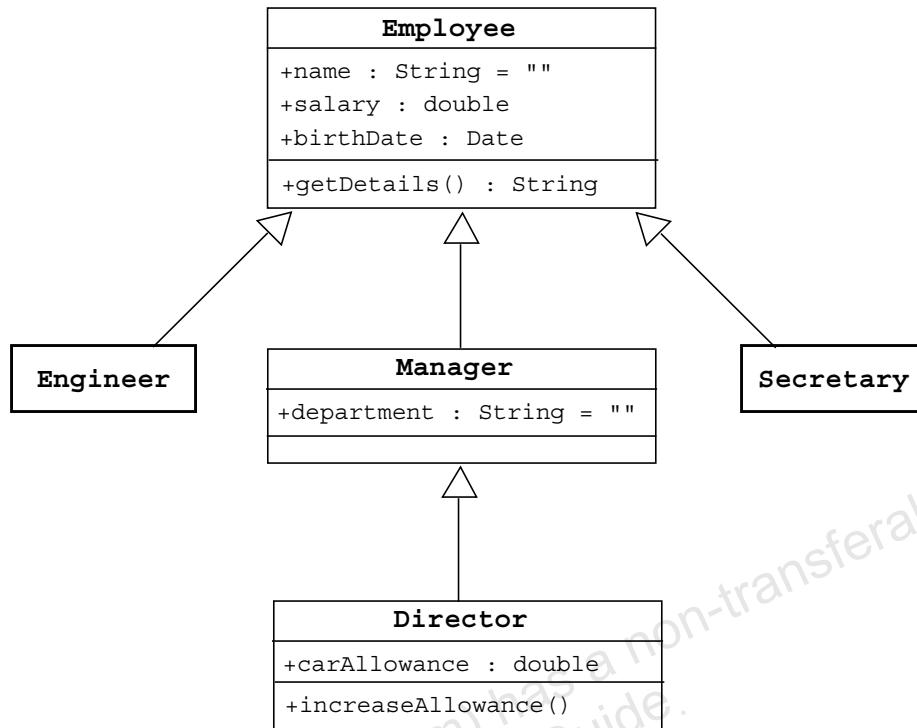
public class Manager extends Employee {
 public String department;
}

```

## Single Inheritance

The Java programming language permits a class to extend one other class only. This restriction is called *single inheritance*. The relative merits of single and multiple inheritance are the subject of extensive discussions among object-oriented programmers. Module 7, “Advanced Class Features,” examines a language feature called *interfaces* that provides most of the benefits of multiple inheritance without suffering from any of its drawbacks.

Figure 6-3 shows the base class `Employee` and three subclasses: `Engineer`, `Manager`, and `Secretary`. The `Manager` is also subclassed by `Director`.



**Figure 6-3** An Example Inheritance Tree

The `Employee` class contains three attributes (`name`, `salary`, and `birthDate`), as well as one method (`getDetails`). The `Manager` class inherits all of these members and specifies an additional attribute, `department`, as well as the `getDetails` method. The `Director` class inherits all of the members of `Employee` and `Manager` and specifies a `carAllowance` attribute and a new method, `increaseAllowance`.

Similarly, the `Engineer` and `Secretary` classes inherit the members of the `Employee` class and might specify additional members (not shown).

# Access Control

Variables and methods can be at one of four access levels: `public`, `protected`, `default`, or `private`. Classes can be at the `public` or `default` level. Table 6-1 shows the access levels.

**Table 6-1** Accessibility Criteria

| Modifier               | Same Class | Same Package | Subclass | Universe |
|------------------------|------------|--------------|----------|----------|
| <code>private</code>   | Yes        |              |          |          |
| <code>default</code>   | Yes        | Yes          |          |          |
| <code>protected</code> | Yes        | Yes          | Yes      |          |
| <code>public</code>    | Yes        | Yes          | Yes      | Yes      |

A variable or method marked `private` is accessible only by methods that are members of the same class as the `private` variable or method.

A variable, method, or class has default accessibility if it does not have an explicit access modifier as part of its declaration. Such accessibility means that access is permitted from any method in classes that are members of the *same package* as the target. This is often called *package-friendly* or *package-private*.

A variable or method marked with the modifier `protected` is actually more accessible than one with default access control. A `protected` method or variable is accessible from methods in classes that are members of the same package and from any method in any *subclass*. You should use the `protected` access when it is appropriate for a class's subclass, but not unrelated classes.

A variable or method marked with the modifier `public` is accessible universally.

**Note –** Protected access is provided to subclasses that reside in a different package from the class that owns the protected feature.



## Overriding Methods

In addition to producing a new class based on an old one by adding additional features, you can modify existing behavior of the parent class.

If a method is defined in a subclass so that the name, return type, and argument list match exactly those of a method in the parent class, then the new method is said to *override* the old one.



**Note** – In J2SE version 5.0, these matching rules have changed slightly. The return type of the overriding method can now be a subclass of the inherited method. A discussion of covariant returns is beyond the scope of this course.

Consider these sample methods in the `Employee` and `Manager` classes:

```
public class Employee {
 protected String name;
 protected double salary;
 protected Date birthDate;

 public String getDetails() {
 return "Name: " + name + "\n"
 + "Salary: " + salary;
 }
}

public class Manager extends Employee {
 protected String department;

 public String getDetails() {
 return "Name: " + name + "\n"
 + "Salary: " + salary + "\n"
 + "Manager of: " + department;
 }
}
```

The `Manager` class has a `getDetails` method by definition because it inherits one from the `Employee` class. However, the original method has been replaced, or overridden, by the child class's version.

## Overriding Methods

# Overridden Methods Cannot Be Less Accessible

Remember that the method name and the order of arguments of a child method must be identical to those of the method in the parent class for that method to override the parent's version. Furthermore, an overriding method cannot be less accessible than the method it overrides.

Consider this invalid scenario:

```
public class Parent {
 public void doSomething() {}
}

public class Child extends Parent {
 private void doSomething() {} // illegal
}

public class UseBoth {
 public void doOtherThing() {
 Parent p1 = new Parent();
 Parent p2 = new Child();
 p1.doSomething();
 p2.doSomething();
 }
}
```

The Java programming language semantics dictate that `p2.method()` results in the Child version of method being executed, but, because the method is declared `private`, `p2` (declared as `Parent`) cannot access it. Thus, the language semantics are violated.

## Invoking Overridden Methods

A subclass method can invoke a superclass method using the `super` keyword.

The `super` keyword refers to the superclass of the class in which the keyword is used. It is used to refer to the member variables or the methods of the superclass.

This can be achieved using the keyword `super` as follows :

```
public class Employee {
 private String name;
 private double salary;
 private Date birthDate;

 public String getDetails() {
 return "Name: " + name + "\nSalary: " + salary;
 }
}

public class Manager extends Employee {
 private String department;

 public String getDetails() {
 // call parent method
 return super.getDetails()
 + "\nDepartment: " + department;
 }
}
```

A call of the form `super.method()` invokes the entire behavior, along with any side effects of the method that would have been invoked if the object had been of the parent class type. The method does not have to be defined in that parent class; it could be inherited from some class even further up the hierarchy.




---

**Note** – In the previous example, member variables have been declared as `private`. This is not necessary but is a generally good programming practice.

---

## Polymorphism

Describing a Manager as *is an* Employee is not just a convenient way of describing the relationship between these two classes. Recall that the Manager has all the members, both attributes and methods, of the parent class Employee. This means that any operation that is legitimate on an Employee is also legitimate on a Manager. If the Employee has the method `getDetails`, then the Manager class does also.

It might seem unrealistic to create a Manager and assign deliberately the reference to it to a variable of type Employee. However, this is possible, and there are reasons why you might want to achieve this effect.

An *object* has only one form (the one that is given to it when constructed). However, a *variable* is polymorphic because it can refer to objects of different forms.

The Java programming language, like most object-oriented languages, actually permits you to refer to an object with a variable that is one of the parent class types. So you can say:

```
Employee e = new Manager(); //legal
```

Using the variable `e` as is, you can access only the parts of the object that are part of Employee; the Manager-specific parts are hidden. This is because as far as the compiler is concerned, `e` is an Employee, not a Manager. Therefore, the following is not permitted:

```
// Illegal attempt to assign Manager attribute
e.department = "Sales";
// the variable is declared as an Employee type,
// even though the Manager object has that attribute
```

## Virtual Method Invocation

Assume that the following scenario is true:

```
Employee e = new Employee();
Manager m = new Manager();
```

If you ask for `e.getDetails()` and `m.getDetails()`, you invoke different behaviors. The `Employee` object executes the version of `getDetails()` associated with the `Employee` class, and the `Manager` object executes the version of `getDetails()` associated with the `Manager` class.

What is less obvious is what happens if you have:

```
Employee e = new Manager();
e.getDetails();
```

or something similar, such as a general method argument or an item from a heterogeneous collection.

In fact, you get the behavior associated with the object to which the variable refers at runtime. The behavior is not determined by the compile time type of the variable. This is an aspect of polymorphism, and is an important feature of object-oriented languages. This behavior is often referred to as *virtual method invocation*.

In the previous example, the `e.getDetails()` method executed is from the object's real type, a `Manager`. If you are a C++ programmer, there is an important distinction to be drawn between the Java programming language and C++. In C++, you get this behavior only if you mark the method as *virtual* in the source. In *pure* object-oriented languages, however, this is not normal. C++ does this to increase execution speed.

## Heterogeneous Collections

You can create collections of objects that have a common class. Such collections are called *homogeneous* collections. For example:

```
MyDate [] dates = new MyDate [2];
dates [0] = new MyDate(22, 12, 1964);
dates [1] = new MyDate(22, 7, 1964);
```

The Java programming language has an `Object` class, so you can make collections of all kinds of elements because all classes extend class `Object`. These collections are called *heterogeneous* collections.

A *heterogeneous* collection is a collection of dissimilar items. In object-oriented languages, you can create collections of many items. All have a common ancestor class: the `Object` class. For example:

```
Employee [] staff = new Employee[1024];
```

## Polymorphism

---

```
staff[0] = new Manager();
staff[1] = new Employee();
staff[2] = new Engineer();
```

You can even write a sort method that puts the employees into age or salary order, regardless of whether some of these employees are managers.



**Note** – Every class is a subclass of Object, so you can use an array of Object as a container for any combination of objects. The only items that cannot be added to an array of Object are primitive variables. However, you can create objects from primitive data using wrapper classes, as described in “Wrapper Classes” on page 6-26.

---

## Polymorphic Arguments

You can write methods that accept a *generic* object (in this case, the class Employee) and work properly on objects of any subclass of this object.

You might produce a method in an application class that takes an employee and compares it with a certain threshold salary to determine the tax liability of that employee. Using the polymorphic features you can do this as follows:

```
public class TaxService {
 public TaxRate findTaxRate(Employee e) {
 // do calculations and return a tax rate for e
 }
}

// Meanwhile, elsewhere in the application class
TaxService taxSvc = new TaxService();
Manager m = new Manager();
TaxRate t = taxSvc.findTaxRate(m);
```

This is legal because a Manager is an Employee. However, the findTaxRate method only has access to the members (both variables and methods) that are defined in the Employee class.

## The instanceof Operator

Given that you can pass objects around using references to their parent classes, sometimes you might want to know what actual objects you have. This is the purpose of the instanceof operator. Suppose the class hierarchy is extended so that you have the following:

```
public class Employee extends Object
public class Manager extends Employee
public class Engineer extends Employee
```

---

**Note** – Remember that, while acceptable, extends Object is redundant. It is shown here only as a reminder.

---



If you receive an object using a reference of type Employee, it might turn out to be a Manager or an Engineer. You can test it by using instanceof as follows:

```
public void doSomething(Employee e) {
 if (e instanceof Manager) {
 // Process a Manager
 } else if (e instanceof Engineer) {
 // Process a Engineer
 } else {
 // Process any other type of Employee
 }
}
```

---

**Note** – In C++, you can do something similar using runtime-type information (RTTI).

---



## Casting Objects

In circumstances where you have received a reference to a parent class, and you have determined that the object is, in fact, a particular subclass by using the `instanceof` operator, you can access the full functionality of the object by casting the reference.

```
public void doSomething(Employee e) {
 if (e instanceof Manager) {
 Manager m = (Manager) e;
 System.out.println("This is the manager of "
 + m.getDepartment());
 }
 // rest of operation
}
```

If you do not make the cast, an attempt to execute `e.getDepartment()` would fail because the compiler cannot locate a method called `getDepartment` in the `Employee` class.

If you do not make the test using `instanceof`, you run the risk of the cast failing. Generally, any attempt to cast an object reference is subjected to several checks:

- Casts *upward* in the class hierarchy are always permitted and, in fact, do not require the cast operator. They can be done by simple assignment.
- For *downward* casts, the compiler must be satisfied that the cast is at least possible. For example, any attempt to cast a `Manager` reference to a `Engineer` reference is not permitted, because the `Engineer` is not a `Manager`. The class to which the cast is taking place must be some subclass of the current reference type.
- If the compiler permits the cast, then the object type is checked at runtime. For example, if it turns out that the `instanceof` check is omitted from the source, and the object being cast is not in fact an object of the type it is being cast to, then a runtime error (*exception*) occurs. Exceptions are a form of runtime error and are the subject of a later module.

## Overloading Methods

In some circumstances, you might want to write several methods in the same class that do the same basic job with different arguments. Consider a simple method that is intended to output a textual representation of its argument. This method could be called `println()`.

Now suppose that you need a different print method for printing each of the `int`, `float`, and `String` types. This is reasonable, because the various data types require different formatting and, probably, varied handling. You could create three methods, called `printInt()`, `printFloat()`, and `printString()`, respectively. However, this is tedious.

The Java programming language, along with several other programming languages, permits you to reuse a method name for more than one method. This works only if there is something in the circumstances under which the call is made that distinguishes the method that is needed. In the case of the three print methods, this distinction is based on the number and type of the arguments.

By reusing the method name, you end up with the following methods:

```
public void println(int i)
public void println(float f)
public void println(String s)
```

When you write code to call one of these methods, the appropriate method is chosen according to the type of argument or arguments you supply.

Two rules apply to overloaded methods:

- Argument lists *must* differ.

The argument lists of the calling statement must differ enough to allow unambiguous determination of the proper method to call. Normal widening promotions (for example, `float` to `double`) might be applied; this can cause confusion under some conditions.

- Return types *can* be different.

The return type of the methods can be different, but it is not sufficient for the return type to be the only difference. The argument lists of overloaded methods must differ.

## Methods Using Variable Arguments

A variation on overloading is when you need a method that takes any number of arguments of the same type. For example, imagine you need to create a method that calculates the average of a set of integers. You could define the following methods:

```
public class Statistics {
 public float average(int x1, int x2) {}
 public float average(int x1, int x2, int x3) {}
 public float average(int x1, int x2, int x3, int x4) {}
}
```

These methods can be invoked as follows:

```
float gradePointAverage = stats.average(4, 3, 4);
float averageAge = stats.average(24, 32, 27, 18);
```

These three overloaded methods all share the same functionality. It would be nice to collapse these methods into one method. J2SE version 5.0 now provides a feature, called *varargs* or *variable arguments*, to enable you to write a more generic method:

```
public class Statistics {
 public float average(int... nums) {
 int sum = 0;
 for (int x : nums) {
 sum += x;
 }
 return ((float) sum) / nums.length;
 }
}
```

This new varargs method can be invoked in the same manner as the suite of overloaded methods. Notice that the `nums` argument is an actual array object of type `int []`; this permits the method to iterate over the elements and to get the number of elements (that is, the length of the array).

# Overloading Constructors

When an object is instantiated, the program might be able to supply multiple constructors based on the data for the object being created. For example, a payroll system might be able to create an Employee object when it knows all of the basic data about the person: name, starting salary, and date of birth. Sometimes the system might not know the starting salary or the date of birth.

Code 6-4 shows four overloaded constructors for the Employee class. The first constructor (Lines 7–11) initializes all instance variables. In the second one (Lines 12–14), the date of birth is not provided. The `this` reference is being used as a forwarding call to another constructor (always within the same class); in this case, the first constructor. Likewise, the third constructor (Lines 15–17) calls the first constructor passing in the class constant `BASE_SALARY`. The fourth constructor (Lines 18–20) calls the second constructor passing the `BASE_SALARY`, which, in turn, calls the first constructor passing `null` for the date of birth.

## Code 6-4 Example Overloading Constructors

```

1 public class Employee {
2 private static final double BASE_SALARY = 15000.00;
3 private String name;
4 private double salary;
5 private Date birthDate;
6
7 public Employee(String name, double salary, Date DoB) {
8 this.name = name;
9 this.salary = salary;
10 this.birthDate = DoB;
11 }
12 public Employee(String name, double salary) {
13 this(name, salary, null);
14 }
15 public Employee(String name, Date DoB) {
16 this(name, BASE_SALARY, DoB);
17 }
18 public Employee(String name) {
19 this(name, BASE_SALARY);
20 }
21 // more Employee code...
22 }
```

The `this` keyword in a constructor must be the first line of code in the constructor. There can be more initialization code after the `this` call, but not before.

## Constructors Are Not Inherited

Although a subclass inherits all of the methods and variables from a parent class, it does not inherit constructors.

There are only two ways in which a class can gain a constructor; either you write the constructor, or, because you have not written any constructors, the class has a single default constructor. A parent constructor is always called in addition to a child constructor. This is described in detail later in this module.

## Invoking Parent Class Constructors

Like methods, constructors can call the non-private constructors of its immediate superclass.

Often you define a constructor that takes arguments and you want to use those arguments to control the construction of the parent part of an object. You can invoke a particular parent class constructor as part of a child class initialization by using the keyword `super` from the child constructor's *first* line. To control the invocation of the specific constructor, you must provide the appropriate arguments to `super()`. When there is no call to `super` with arguments, the parent constructor with zero arguments is called implicitly. In this case, if there is no parent constructor with zero arguments, a compiler error results.

The call to `super()` can take any number of arguments appropriate to the various constructors available in the parent class, but it must be the first statement in the constructor.

Assuming that the `Employee` class has the set of constructors that were defined in the Code 6-4 on page 6-17, then the following constructors in `Manager` might be defined. The constructor on Line 12 is illegal because the compiler inserts an implicit call to `super()`, and the `Employee` class has not provided a constructor without arguments.

```
1 public class Manager extends Employee {
2 private String department;
3
4 public Manager(String name, double salary, String dept) {
5 super(name, salary);
6 department = dept;
7 }
8 public Manager(String name, String dept) {
9 super(name);
10 department = dept;
11 }
12 public Manager(String dept) { // This code fails: no super()
13 department = dept;
14 }
15 }
```

When used, you must place `super` or `this` in the first line of the constructor. If you write a constructor that has neither a call to `super(...)` nor `this(...)`, the compiler inserts a call to the parent class constructor automatically, with no arguments. Other constructors can also call `super(...)` or `this(...)`, invoking a chain of constructors. What happens ultimately is that the parent class constructor (or possibly several) executes before any child class constructor in the chain.

# Constructing and Initializing Objects: A Slight Reprise

Object initialization is a rather complex process. In the section “Constructing and Initializing Objects” in Module 3, “Identifiers, Keywords, and Types,” you were exposed to a rudimentary explanation. In this section, you see the whole process.

First, the memory for the complete object is allocated and the default values for the instance variables are assigned. Second, the top-level constructor is called and follows these steps recursively down the inheritance tree:

1. Bind constructor parameters.
2. If explicit `this()`, call recursively, and then skip to Step 5.
3. Call recursively the implicit or explicit `super(...)`, except for `Object` because `Object` has no parent class.
4. Execute the explicit instance variable initializers.
5. Execute the body of the current constructor.

## Constructor and Initialization Examples

For an example, use the following code for the Manager and Employee classes:

```
1 public class Object {
2 public Object() {}
3 }

1 public class Employee extends Object {
2 private String name;
3 private double salary = 15000.00;
4 private Date birthDate;
5
6 public Employee(String n, Date DoB) {
7 // implicit super();
8 name = n;
9 birthDate = DoB;
10 }
11 public Employee(String n) {
12 this(n, null);
13 }
14 }
```

```

1 public class Manager extends Employee {
2 private String department;
3
4 public Manager(String n, String d) {
5 super(n);
6 department = d;
7 }
8 }

```

The following are the steps to construct new Manager ("Joe Smith", "Sales"):

- 0 Basic initialization
  - 0.1 Allocate memory for the complete Manager object
  - 0.2 Initialize all instance variables to their default values
- 1 Call constructor: Manager("Joe Smith", "Sales")
  - 1.1 Bind constructor parameters: n="Joe Smith", d="Sales"
  - 1.2 No explicit this() call
  - 1.3 Call super(n) for Employee(String)
    - 1.3.1 Bind constructor parameters: n="Joe Smith"
    - 1.3.2 Call this(n, null) for Employee(String, Date)
      - 1.3.2.1 Bind constructor parameters: n="Joe Smith", DoB=null
      - 1.3.2.2 No explicit this() call
      - 1.3.2.3 Call super() for Object()
        - 1.3.2.3.1 No binding necessary
        - 1.3.2.3.2 No this() call
        - 1.3.2.3.3 No super() call (Object is the root)
        - 1.3.2.3.4 No explicit variable initialization for Object
        - 1.3.2.3.5 No method body to call
      - 1.3.2.4 Initialize explicit Employee variables: salary=15000.00;
      - 1.3.2.5 Execute body: name="Joe Smith"; date=null;
    - 1.3.3 - 1.3.4 Steps skipped
    - 1.3.5 Execute body: No body in Employee(String)
  - 1.4 No explicit initializers for Manager
  - 1.5 Execute body: department="Sales"

The Object Class

# The Object Class

The Object class is the root of all classes in the Java technology programming language. If a class is declared with no extends clause, then the compiler adds implicitly the code extends Object to the declaration; for example:

```
public class Employee {
 // more code here
}
```

is equivalent to:

```
public class Employee extends Object {
 // more code here
}
```

This enables you to override several methods inherited from the Object class. The following sections describe two important Object methods.

## The equals Method

The == operator performs an equivalent comparison. That is, for any reference values x and y, x==y returns true if and only if x and y refer to the same object.

The Object class in the java.lang package has the method public boolean equals(Object obj), which compares two objects for equality. When not overridden, an object's equals() method returns true only if the two references being compared refer to the same object. However, the intention of the equals() method is to compare the contents of two objects whenever possible. This is why the method is frequently overridden. For example, the equals() method in String class returns true if and only if the argument is not null and is a String object that represents the same sequence of characters as the String object with which the method is invoked.

---

**Note** – You should override the hashCode method whenever you override the equals method. A simple implementation could use a bitwise XOR on the hash codes of the elements tested for equality.

---



## An equals Example

In this example, the `MyDate` class has been modified to include an `equals` method that tests against the year, month, and day attributes.

```

1 public class MyDate {
2 private int day;
3 private int month;
4 private int year;
5
6 public MyDate(int day, int month, int year) {
7 this.day = day;
8 this.month = month;
9 this.year = year;
10 }
11
12 public boolean equals(Object o) {
13 boolean result = false;
14 if ((o != null) && (o instanceof MyDate)) {
15 MyDate d = (MyDate) o;
16 if ((day == d.day) && (month == d.month)
17 && (year == d.year)) {
18 result = true;
19 }
20 }
21 return result;
22 }
23
24 public int hashCode() {
25 return (day ^ month ^ year);
26 }
27 }
```

The `hashCode` method implements a bitwise XOR of the date attributes. This guarantees that hash code for equal `MyDate` objects have the same value while making it likely that different dates will return different values.

The following program tests two `MyDate` objects that are not identical, but are equal relative to the year-month-day test.

```

1 class TestEquals {
2 public static void main(String[] args) {
3 MyDate date1 = new MyDate(14, 3, 1976);
4 MyDate date2 = new MyDate(14, 3, 1976);
5 }
6 }
```

## The Object Class

---

```

6 if (date1 == date2) {
7 System.out.println("date1 is identical to date2");
8 } else {
9 System.out.println("date1 is not identical to date2");
10 }
11
12 if (date1.equals(date2)) {
13 System.out.println("date1 is equal to date2");
14 } else {
15 System.out.println("date1 is not equal to date2");
16 }
17
18 System.out.println("set date2 = date1;");
19 date2 = date1;
20
21 if (date1 == date2) {
22 System.out.println("date1 is identical to date2");
23 } else {
24 System.out.println("date1 is not identical to date2");
25 }
26 }
27 }
```

The execution of this test program generates the following output:

```

date1 is not identical to date2
date1 is equal to date2
set date2 = date1;
date1 is identical to date2
```

## The `toString` Method

The `toString` method converts an object to a `String` representation. It is referenced by the compiler when automatic string conversion takes place. For example, the `System.out.println()` call:

```
Date now = new Date();
System.out.println(now);
```

is equivalent to:

```
System.out.println(now.toString());
```

The `Object` class defines a default `toString` method that returns the class name and its reference address (not normally useful). Many classes override `toString` to provide more useful information. For example, all wrapper classes (introduced later in this module) override `toString` to provide a string form of the value they represent. Even classes representing items without a string form often implement `toString` to return object state information for debugging purposes.

## Wrapper Classes

---

# Wrapper Classes

The Java programming language does not look at primitive data types as objects. For example, numerical, Boolean, and character data are treated in the primitive form for the sake of efficiency. The Java programming language provides *wrapper* classes to manipulate primitive data elements as objects. Such data elements are *wrapped* in an object created around them. Each Java primitive data type has a corresponding wrapper class in the `java.lang` package. Each wrapper class object encapsulates a single primitive value. (See Table 6-2.)



**Note** – These wrapper classes implement *immutable* objects. That means that after the primitive value is initialized in the wrapper object, then there is no way to change that value.

**Table 6-2** Wrapper Classes

| Primitive Data Type  | Wrapper Class          |
|----------------------|------------------------|
| <code>boolean</code> | <code>Boolean</code>   |
| <code>byte</code>    | <code>Byte</code>      |
| <code>char</code>    | <code>Character</code> |
| <code>short</code>   | <code>Short</code>     |
| <code>int</code>     | <code>Integer</code>   |
| <code>long</code>    | <code>Long</code>      |
| <code>float</code>   | <code>Float</code>     |
| <code>double</code>  | <code>Double</code>    |

You can construct a wrapper class object by passing the value to be wrapped into the appropriate constructor, as shown in Code 6-5.

### Code 6-5 Examples of Primitive Boxing Using Wrapper Classes

```
int pInt = 420;
Integer wInt = new Integer(pInt); // this is called boxing
int p2 = wInt.intValue(); // this is called unboxing
```

Wrapper classes are useful when converting primitive data types because of the many wrapper class methods available; for example:

```
int x = Integer.valueOf(str).intValue();
```

or:

```
int x = Integer.parseInt(str);
```

## Autoboxing of Primitive Types

If you have to change the primitive data types to their object equivalents (called *boxing*), then you need to use the wrapper classes. Also to get the primitive data type from the object reference (called *unboxing*), you need to use the wrapper class methods. All of this boxing and unboxing can clutter up your code and thus make your code difficult to understand. In J2SE version 5.0, the autoboxing feature enables you to assign and retrieve primitive types without the need of the wrapper classes.

Code 6-6 shows two simple cases of autoboxing and autounboxing. Compare this with Code 6-5 on page 6-26.

### **Code 6-6 Examples of Primitive Autoboxing**

```
int pInt = 420;
Integer wInt = pInt; // this is called autoboxing
int p2 = wInt; // this is called autounboxing
```

The J2SE version 5.0 compiler will now create the wrapper object automatically when assigning a primitive to a variable of the wrapper class type. The compiler will also extract the primitive value when assigning from a wrapper object to a primitive variable.

This can be done when passing parameters to methods or even within arithmetic expressions.



**Caution** – Do not overuse the autoboxing feature. There is a hidden performance impact when a value is autoboxed or autounboxed. Mixing primitives and wrapper objects in arithmetic expressions within a tight loop might have a negative impact on the performance and throughput of your applications.



## Module 7

---

# Advanced Class Features

---

## Objectives

Upon completion of this module, you should be able to:

- Create static variables, methods, and initializers
- Create final classes, methods, and variables
- Create and use enumerated types
- Use the static import statement
- Create abstract classes and methods
- Create and use an interface

This module completes the discussion about the object-oriented features of the Java technology programming language.

---

## Relevance

# Relevance



**Discussion** – The following questions are relevant to the material presented in this module:

- How can you create a constant?

---

---

---

- How can you declare data that is shared by all instances of a given class?

---

---

---

- How can you keep a class or method from being subclassed or overridden?

---

---

---

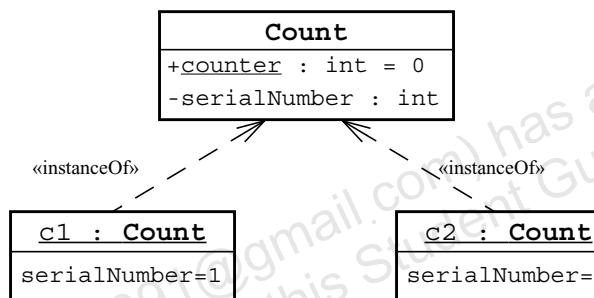
# The static Keyword

The `static` keyword declares members (attributes, methods, and nested classes) that are associated with the class rather than the instances of the class.

The following sections describe the most common uses of the `static` keyword: class variables and class methods.

## Class Attributes

Sometimes it is desirable to have a variable that is shared among all instances of a class. For example, you could use this variable as the basis for communication between instances or to keep track of the number of instances that have been created (see Figure 7-1).



**Figure 7-1** UML Object Diagram of the Count Class and Two Unique Instances

You achieve this shared effect by marking the variable with the keyword `static`. Such a variable is sometimes called a *class variable* to distinguish it from a member or instance variable, which is not shared.

```

1 public class Count {
2 private int serialNumber;
3 public static int counter = 0;
4
5 public Count() {
6 counter++;
7 serialNumber = counter;
8 }
9 }

```

## The static Keyword

---

In this example, every object that is created is assigned a unique serial number, starting at 1 and counting upwards. The variable counter is shared among all instances, so when the constructor of one object increments counter, the next object to be created receives the incremented value.

A static variable is similar in some ways to a global variable in other languages. The Java programming language does not have global variables, but a static variable is a single variable that is accessible from any instance of the class.

If a static variable is not marked as private, you can access it from outside the class. To do this, you do not need an instance of the class, you can refer to it through the class name.

```

1 public class OtherClass {
2 public void incrementNumber() {
3 Count1.counter++;
4 }
5 }
```

## Class Methods

Sometimes you need to access program code when you do not have an instance of a particular object available. A method that is marked using the keyword static can be used in this way and is sometimes called a *class method*.

```

1 public class Count2 {
2 private int serialNumber;
3 private static int counter = 0;
4
5 public static int getTotalCount() {
6 return counter;
7 }
8
9 public Count2() {
10 counter++;
11 serialNumber = counter;
12 }
13 }
```

You should access methods that are static using the class name rather than an object reference, as follows:

```
1 public class TestCounter {
```

```
2 public static void main(String[] args) {
3 System.out.println("Number of counter is "
4 + Count2.getTotalCount());
5 Count2 count1 = new Count2();
6 System.out.println("Number of counter is "
7 + Count2.getTotalCount());
8 }
9 }
```

The output of the TestCounter program is:

```
Number of counter is 0
Number of counter is 1
```

Because you can invoke a static method without any instance of the class to which it belongs, there is no this value. The consequence is that a static method cannot access any variables other than the local variables, static attributes, and its parameters. Attempting to access non-static attributes causes a compiler error.

Non-static attributes are bound to an instance and can be accessed through instance references only.

```
1 public class Count3 {
2 private int serialNumber;
3 private static int counter = 0;
4
5 public static int getSerialNumber() {
6 return serialNumber; // COMPILER ERROR!
7 }
8 }
```

## The static Keyword

---

You should be aware of the following when using static methods:

- You cannot override a static method but you can hide it.  
For a method to be *overridden* it must be non-static. Two static methods with the same signature in a class hierarchy simply means that there are two independent class methods. If a class method is applied to an object reference, the method invoked is the one for the class for which the variable was declared.
- The `main()` method is a static method because the JVM does not create an instance of the class when executing the `main` method. So if you have member data, you must create an object to access it.

## Static Initializers

A class can contain code in *static blocks* that are not part of normal methods. Static block code executes once when the class is loaded. If a class contains more than one static block, they are executed in the order of their appearance in the class.

```

1 public class Count4 {
2 public static int counter;
3 static {
4 counter = Integer.getInteger("myApp.Count4.counter").intValue();
5 }
6 }

1 public class TestStaticInit {
2 public static void main(String[] args) {
3 System.out.println("counter = " + Count4.counter);
4 }
5 }

```

The code on Line 4 of the `Count4` class uses a static method of the `Integer` class `getInteger(String)`, which returns an `Integer` object that represents the value of a system property. This property, named `myApp.Count4.counter`, is set on the command line using the `-D` option. The `intValue` method on the `Integer` object returns the value as an `int`.

The result is the following:

```
java -DmyApp.Count4.counter=47 TestStaticInit
counter = 47
```

# The final Keyword

This section describes final classes, final methods, and final variables.

## Final Classes

The Java programming language permits you to apply the keyword `final` to classes. If you do this, the class cannot be subclassed. For example, the class `java.lang.String` is a `final` class. This is done for security reasons, because it ensures that if a method references a string, it is a definite string of class `String` and not a string of a class that is a modified subclass of `String` that might have been changed.

## Final Methods

You can also mark individual methods as `final`. Methods marked `final` cannot be overridden. For security reasons, you should make a method `final` if the method has an implementation that should not be changed and is critical to the consistent state of the object.

Methods declared `final` can be optimized. The compiler can generate code that causes a direct call to the method, rather than the usual, virtual-method invocation that involves a runtime lookup. Methods marked as `static` or `private` can be optimized by the compiler as if they had been marked `final`, because dynamic binding cannot be applied in either case.

## Final Variables

If a variable is marked as `final`, the effect is to make it a constant. Any attempt to change the value of a `final` variable causes a compiler error. The following example shows a `final` variable that is defined properly:

```
public class Bank {
 private static final double DEFAULT_INTEREST_RATE=3.2;
 // more declarations
}
```

## The `final` Keyword



**Note** – If you mark a variable of reference type (that is, any class type) as `final`, that variable cannot refer to any other object. However, you can change the object's contents, because only the reference itself is `final`.

### Blank Final Variables

A *blank final variable* is a final variable that is not initialized in its declaration. The initialization is delayed. A blank final instance variable must be assigned in a constructor, but it can be set once only. A blank final variable that is a local variable can be set at any time in the body of the method, but it can be set once only. The following code fragment is an example of how a blank final variable can be used in a class:

```
public class Customer {
 private final long customerID;

 public Customer() {
 customerID = createID();
 }
 public long getID() {
 return customerID;
 }
 private long createID() {
 return ... // generate new ID
 }
 ... // more declarations
}
```

## Enumerated Types

A common idiom in programming is to have finite set of symbolic names that represent the values of an attribute. For example, to represent the suits of playing cards you might create a set of symbols: SPADES, HEARTS, CLUBS, and DIAMONDS. This is often called an *enumerated type*.

---

**Note** – Do not get confused by the term *enumerated type* and the `java.util.Enumeration` class. They are unrelated in all respects.

---



## Old-Style Enumerated Type Idiom

Code 7-1 illustrates the old-style (before J2SE version 5.0) idiom for creating an enumerated type using integer constants for the symbolic names of the enumerated values.

### Code 7-1 Old-Style Enumerated Type Example

```
1 package cards.domain;
2
3 public class PlayingCard {
4
5 // pseudo enumerated type
6 public static final int SUIT_SPADES = 0;
7 public static final int SUIT_HEARTS = 1;
8 public static final int SUIT_CLUBS = 2;
9 public static final int SUIT_DIAMONDS = 3;
10
11 private int suit;
12 private int rank;
13
14 public PlayingCard(int suit, int rank) {
15 this.suit = suit;
16 this.rank = rank;
17 }
18
19 public int getSuit() {
20 return suit;
21 }
```

**Note** – The rank attribute is also a good candidate for an enumerated type, but this code does not use an enumerated type for brevity.

---



Furthermore, the user interface will need to display strings rather than integers to represent the card suit. Code 7-2 shows the `getSuitName` method of the `PlayingCard` class. Notice Lines 37 and 38; this code is used when the `suit` attribute holds an invalid integer value.

### Code 7-2 The `getSuitName` Method

```
22 public String getSuitName() {
23 String name = "";
24 switch (suit) {
25 case SUIT_SPADES:
26 name = "Spades";
27 break;
28 case SUIT_HEARTS:
29 name = "Hearts";
30 break;
31 case SUIT_CLUBS:
32 name = "Clubs";
33 break;
34 case SUIT_DIAMONDS:
35 name = "Diamonds";
36 break;
37 default:
38 System.err.println("Invalid suit.");
39 }
40 return name;
41 }
```

## Enumerated Types

---

Code 7-3 shows the `TestPlayingCard` program that creates two playing card objects and then displays their rank and suit. This program illustrates the primary problem with the old-style of enumerated type idiom. The `PlayingCard` constructor takes two arguments: an integer for the suit and an integer for the rank. The first call to the `PlayingCard` constructor (Line 9) uses an appropriate symbolic constant, but the second call to the `PlayingCard` constructor (Line 14) uses an arbitrary integer value. Both constructor calls are valid from the perspective of the compiler.

### Code 7-3 The `TestPlayingCard` Program

```

1 package cards.tests;
2
3 import cards.domain.PlayingCard;
4
5 public class TestPlayingCard {
6 public static void main(String[] args) {
7
8 PlayingCard card1
9 = new PlayingCard(PlayingCard.SUIT_SPADES, 2);
10 System.out.println("card1 is the " + card1.getRank()
11 + " of " + card1.getSuitName());
12
13 // You can create a playing card with a bogus suit.
14 PlayingCard card2 = new PlayingCard(47, 2);
15 System.out.println("card2 is the " + card2.getRank()
16 + " of " + card2.getSuitName());
17 }
18 }
19

```

However, when this program is executed you can see that the second card object is invalid and does not display properly:

```

> java cards.tests.TestPlayingCard
card1 is the 2 of Spades
Invalid suit.
card2 is the 2 of

```

The statement `Invalid suit.` is the error message sent by the `getSuitName` method on Line 38 in Code 7-2.

This enumerated type idiom has several problems:

- Not type-safe – Because a `suit` attribute is just an `int`, you can pass in any other `int` value where a suit is required. A programmer could also apply arithmetic operations on two suits, which makes no sense.

- No namespace – You must prefix constants of an `int enum` with a string (in this case, `SUIT_`) to avoid collisions with other `int enum` types. For example, if you used an `int enum` for the rank value, then you would probably create a set of `RANK_XYZ` `int` constants.
- Brittle character – Because `int enums` are compile-time constants, they are compiled into clients that use them. If a new constant is added between two existing constants or the order is changed, clients must be recompiled. If they are not, they will still run, but their behavior will be undefined.
- Uninformative printed values – Because they are just `int` values, if you print one out all you see is a number, which tells you nothing about what it represents, or even what type it is. That is why the `PlayingCard` class needs the `getSuitName` method.

## The New Enumerated Type

The Java SE version 5.0 of the Java programming language includes a type-safe enumerated type feature. Code 7-4 shows an enumerated type for the suits of a playing card. You can think of the `Suit` type as a class with a finite set of values that are given the symbolic names listed in the type definition. For example, `Suit.SPADES` is of type `Suit`.

### **Code 7-4** The Suit Enumerated Type

```

1 package cards.domain;
2
3 public enum Suit {
4 SPADES,
5 HEARTS,
6 CLUBS,
7 DIAMONDS
8 }
```

Code 7-5 shows the `PlayingCard` class using the `Suit` type for the data type of the `suit` attribute.

### **Code 7-5** The PlayingCard Class Using the Suit Type

```

1 package cards.domain;
2
3 public class PlayingCard {
4
5 private Suit suit;
6 private int rank;
```

## Enumerated Types

---

```
7
8 public PlayingCard(Suit suit, int rank) {
9 this.suit = suit;
10 this.rank = rank;
11 }
12
13 public Suit getSuit() {
14 return suit;
15 }
16 public String getSuitName() {
17 String name = "";
18 switch (suit) {
19 case SPADES:
20 name = "Spades";
21 break;
22 case HEARTS:
23 name = "Hearts";
24 break;
25 case CLUBS:
26 name = "Clubs";
27 break;
28 case DIAMONDS:
29 name = "Diamonds";
30 break;
31 default:
32 // No need for error checking as the Suit
33 // enum is finite.
34 }
35 return name;
36 }
```

This solves the type-safety issues with the old-style enumerated type idiom. Code 7-6 shows an updated test program. Line 14, if uncommented, would result in a compiler error because the int value of 47 is not of type Suit.

### Code 7-6 The TestPlayingCard Program

```
1 package cards.tests;
2
3 import cards.domain.PlayingCard;
4 import cards.domain.Suit;
5
6 public class TestPlayingCard {
7 public static void main(String[] args) {
8
9 PlayingCard card1
10 = new PlayingCard(Suit.SPADES, 2);
11 System.out.println("card1 is the " +
12 card1.getRank()
13 + " of " + card1.getSuitName());
14 // PlayingCard card2 = new PlayingCard(47, 2);
15 // This will not compile.
16 }
17 }
```

## Advanced Enumerated Types

Unfortunately, the `PlayingCard` class still requires a `getSuitName` method to display a user-friendly name in the user interface. If the program were to display the actual `Suit` value, it would show the symbolic name of the type value; for example, `Suit.SPADES` would display as `SPADES`. This is more user-friendly than “0,” but is still not as user-friendly as `Spades`.

Furthermore, the name of the suit should not be part of the `PlayingCard` class, but rather part of the `Suit` type. The new enumerated type feature permits both attributes and methods, just like regular classes. Code 7-7 shows a refinement of the first `Suit` type (Code 7-4 on page 7-13) with a `name` attribute and `getName` method. Notice the use of proper information hiding with the private attribute and the public accesser method.

### Code 7-7 The Suit Type with an Attribute

```
1 package cards.domain;
2
3 public enum Suit {
4 SPADES ("Spades"),
5 HEARTS ("Hearts"),
6 CLUBS ("Clubs"),
7 DIAMONDS ("Diamonds");
8
9 private final String name;
10
11 private Suit(String name) {
12 this.name = name;
13 }
14
15 public String getName() {
16 return name;
17 }
18 }
```

An `enum` constructor should always use the `private` accessibility. The arguments to the constructor are supplied after each declared value. For example, on Line 4 the string “`Spades`” is the argument to the `enum` constructor for the `SPADES` value. Enumerated types can have any number of attributes and methods.

Finally, Code 7-8 shows the modified test program that uses the new `getName` method on the `Suit` type. On Line 12, the `getSuit` method returns a `Suit` value and the `getName` method returns the `name` attribute of that `Suit` value.

### Code 7-8 The TestPlayingCard Program Using Suit Methods

```
1 package cards.tests;
2
3 import cards.domain.PlayingCard;
4 import cards.domain.Suit;
5
6 public class TestPlayingCard {
7 public static void main(String[] args) {
8
9 PlayingCard card1
10 = new PlayingCard(Suit.SPADES, 2);
11 System.out.println("card1 is the " +
12 card1.getRank()
13 + " of " +
14 card1.getSuit().getName());
15
16 }
17 }
```

## Static Imports

---

# Static Imports

If you have to access the static members of a class, then it is necessary to qualify the references with the class from which they come. This is also true of the enumeration type values. Line 10 of Code 7-8 on page 7-17 shows accessing the SPADES value of the Suit type using the dot-notation `Suit.SPADES`.

J2SE version 5.0 provides the static import feature that enables unqualified access to static members without having to qualify them with the class name. Code 7-9 shows the use of static imports. Line 4 tells the compiler to include the enumerated type values (or any static member of that type) in the symbol table when compiling this program. Therefore, Line 9 can use SPADES without the `Suit.` namespace prefix.

### Code 7-9 The TestPlayingCard Program Using Static Imports

```

1 package cards.tests;
2
3 import cards.domain.PlayingCard;
4 import static cards.domain.Suit.*;
5
6 public class TestPlayingCard {
7 public static void main(String[] args) {
8
9 PlayingCard card1 = new PlayingCard(SPADES, 2);
10 System.out.println("card1 is the " +
11 card1.getRank() +
12 + " of " +
13 card1.getSuit().getName());
14
15 }
16 }
```

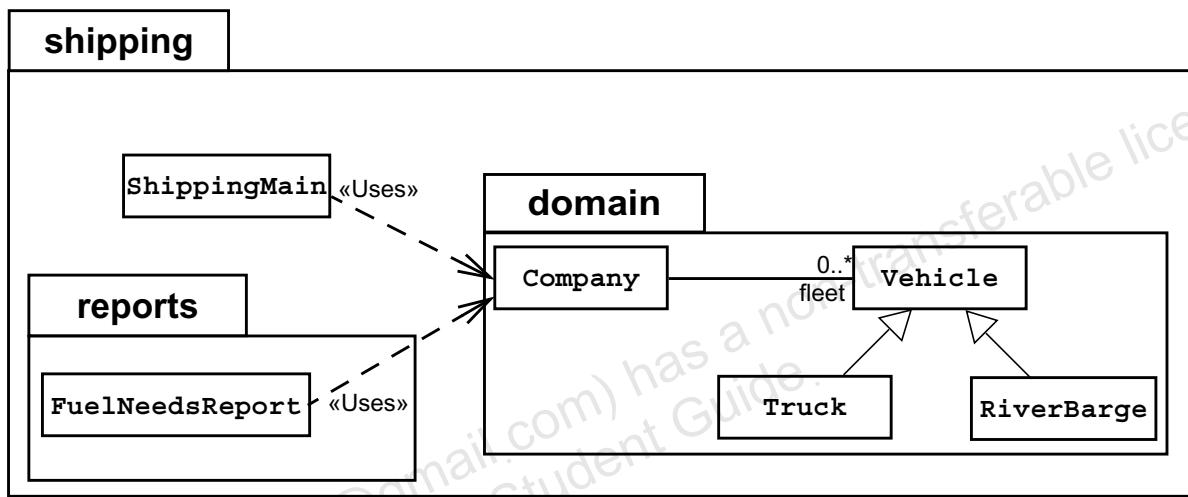


**Caution** – Use static imports sparingly. If you overuse the static import feature, it can make your program unreadable and unmaintainable, polluting its namespace with all of the static members that you import. Readers of your code (including you, a few months after you wrote it) will not know from which class a static member comes. Importing *all* of the static members from a class can be very harmful to readability; if you need one or two members only, import them individually. Used appropriately, static import can make your program *more* readable, by removing the boilerplate of repetition of class names.

# Abstract Classes

In the shipping example, suppose that the system needed to supply a weekly report that lists each vehicle in the company's fleet and the fuel needs for their upcoming trips. Assume that the Shipping System has a `ShippingMain` class that populates the company's vehicle fleet list and generates the Fuel Needs report.

Figure 7-2 shows the UML model of the company and its heterogeneous collection of vehicles (the fleet association).



**Figure 7-2** UML Model of the Company Fleet

The `ShippingMain` program shows the initialization of an example fleet.

```

1 public class ShippingMain {
2 public static void main(String[] args) {
3 Company c = new Company();
4
5 // populate the company with a fleet of vehicles
6 c.addVehicle(new Truck(10000.0));
7 c.addVehicle(new Truck(15000.0));
8 c.addVehicle(new RiverBarge(500000.0));
9 c.addVehicle(new Truck(9500.0));
10 c.addVehicle(new RiverBarge(750000.0));
11
12 FuelNeedsReport report = new FuelNeedsReport(c);
13 report.generateText(System.out);
14 }
15 }
```

## Abstract Classes

---

You should write the report code as follows:

```

1 public class FuelNeedsReport {
2 private Company company;
3
4 public FuelNeedsReport(Company company) {
5 this.company = company;
6 }
7
8 public void generateText(PrintStream output) {
9 Vehicle v;
10 double fuel;
11 double total_fuel = 0.0;
12
13 for (int i = 0; i < company.getFleetSize(); i++) {
14 v = company.getVehicle(i);
15
16 // Calculate the fuel needed for this trip
17 fuel = v.calcTripDistance() / v.calcFuelEfficiency();
18
19 output.println("Vehicle " + v.getName() + " needs "
20 + fuel + " liters of fuel.");
21 total_fuel += fuel;
22 }
23 output.println("Total fuel needs is " + total_fuel + " liters.");
24 }
25 }
```

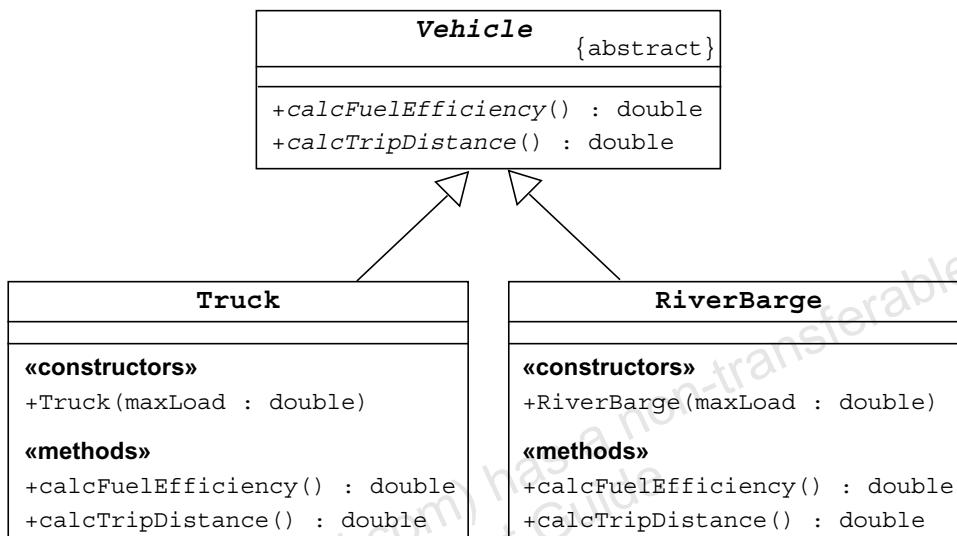
The calculation for the fuel requirements is the trip distance (in kilometers) divided by the vehicle's fuel efficiency (in kilometers per liter).

## The Problem

The calculations to determine fuel efficiency of a truck as compared with a river barge might differ radically. The `Vehicle` class can not supply these two methods, but its subclasses (`Truck` and `RiverBarge`) can.

## The Solution

The Java programming language enables a class designer to specify that a superclass declares a method that does not supply an implementation. This is called an *abstract method*. The implementation of this method is supplied by the subclasses. Any class with one or more abstract methods is called an *abstract class* (see Figure 7-3).



**Figure 7-3** UML Model of the Abstract Vehicle Class

Figure 7-3 presents a UML model of the solution. **Vehicle** is an abstract class with two public, abstract methods.



**Note** – UML uses the italic font to indicate abstract elements in a class diagram. You may also indicate an abstract class with the `{abstract}` constraint flag in the name compartment.

The Java compiler prevents you from instantiating an abstract class. For example, the statement `new Vehicle()` is illegal.

The declaration of the **Vehicle** class is:

```

1 public abstract class Vehicle {
2 public abstract double calcFuelEfficiency();
3 public abstract double calcTripDistance();
4 }

```

The **Truck** class must create an implementation:

## Abstract Classes

---

```
1 public class Truck extends Vehicle {
2 public Truck(double maxLoad) {...}
3 public double calcFuelEfficiency() {
4 // calculate the fuel consumption of a truck
5 }
6 public double calcTripDistance() {
7 // calculate the distance of this trip on highway
8 }
9 }
```

The RiverBarge class must create an implementation:

```
1 public class RiverBarge extends Vehicle {
2 public RiverBarge(double maxLoad) {...}
3 public double calcFuelEfficiency() {
4 // calculate the fuel consumption of a river barge
5 }
6 public double calcTripDistance() {
7 // calculate the distance of this trip along rivers
8 }
9 }
```

However, abstract classes can have data attributes, concrete methods, and constructors. For example, the Vehicle class might include load and maxLoad attributes and a constructor to initialize them. It is a good practice to make these constructors protected rather than public, because it is only meaningful for these constructors to be invoked by the subclasses of the abstract class. Making these constructors protected makes it more obvious to the programmer that the constructor should not be called from arbitrary classes.

## Interfaces

The *public interface* of a class is a contract between the *client code* and the class that provides the service. Concrete classes implement each method. However, an abstract class can defer the implementation by declaring the method to be abstract, and a Java interface declares only the contract and no implementation.

A concrete class implements an interface by defining all methods declared by the interface. Many classes can implement the same interface. These classes do not need to share the same class hierarchy. Also, a class can implement more than one interface. This is described in the following sections.

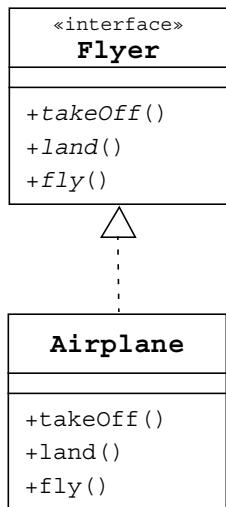
As with abstract classes, use an interface name as a type of reference variable. The usual dynamic binding takes effect. References are cast to and from interface types, and you use the `instanceof` operator to determine if an object's class implements an interface.



**Note** – All methods declared in an interface are `public` and `abstract`, no matter if we explicitly mention these modifiers in the code or not. Similarly, all attributes are `public`, `static` and `final`; in other words, you can only declare constant attributes.

## The Flyer Example

Imagine a group of objects that all share the same ability: they fly. You can construct a public interface, called `Flyer`, that supports three operations: `takeOff`, `land`, and `fly`. (see Figure 7-4).



**Figure 7-4** The Flyer Interface and Airplane Implementation

```

1 public interface Flyer {
2 public void takeOff();
3 public void land();
4 public void fly();
5 }

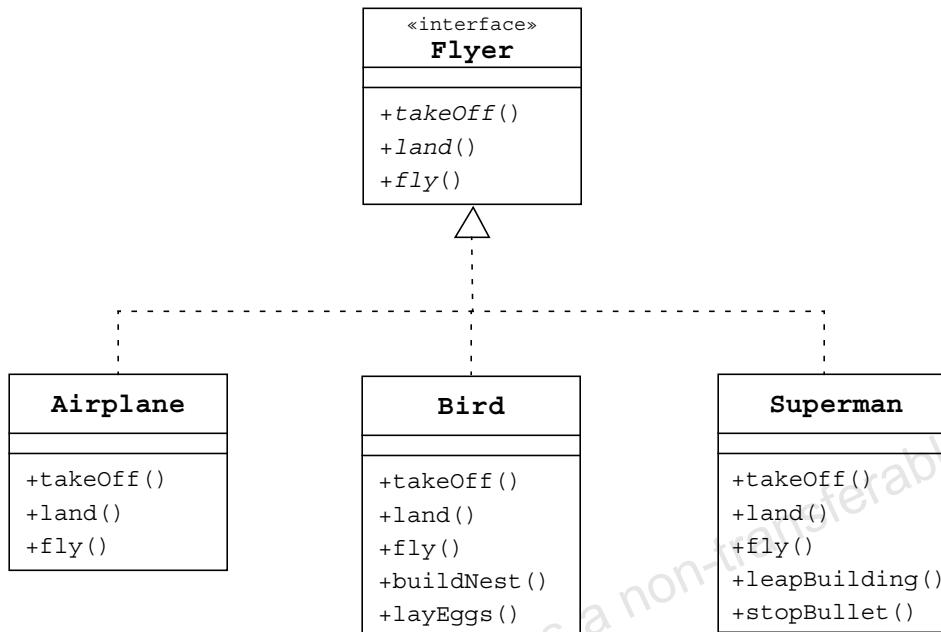
```

```

1 public class Airplane implements Flyer {
2 public void takeOff() {
3 // accelerate until lift-off
4 // raise landing gear
5 }
6 public void land() {
7 // lower landing gear
8 // decelerate and lower flaps until touch-down
9 // apply brakes
10 }
11 public void fly() {
12 // keep those engines running
13 }
14 }

```

There can be multiple classes that implement the Flyer interface, as shown in Figure 7-5. An airplane can fly, a bird can also fly, Superman can fly, and so on.

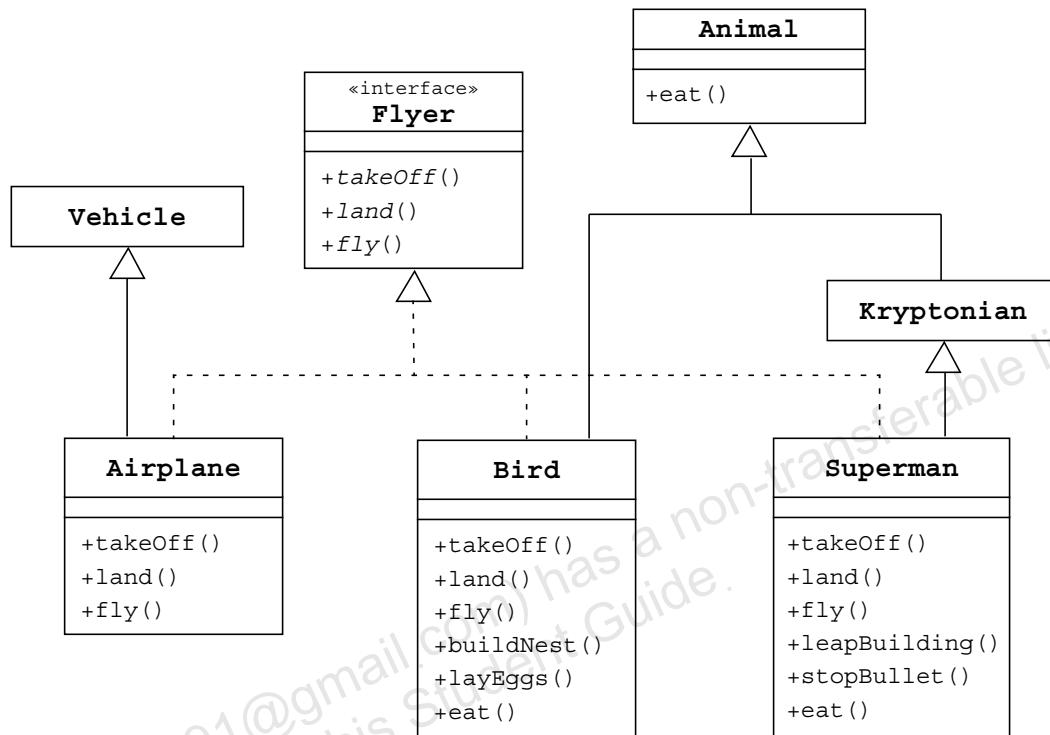


**Figure 7-5** Multiple Implementations of the Flyer Interface

An Airplane is a Vehicle, and it can fly. A Bird is an Animal, and it can fly. These examples show that a class can inherit from one class but also implement some other interface.

## Interfaces

This sounds like multiple inheritance, but it is not quite that. The danger of multiple inheritance is that a class could inherit two distinct implementations of the same method. This is not possible with interfaces because an interface method declaration supplies no implementation, as shown in Figure 7-6.



**Figure 7-6** A Mixture of Inheritance and Implementation

The following describes the Bird class:

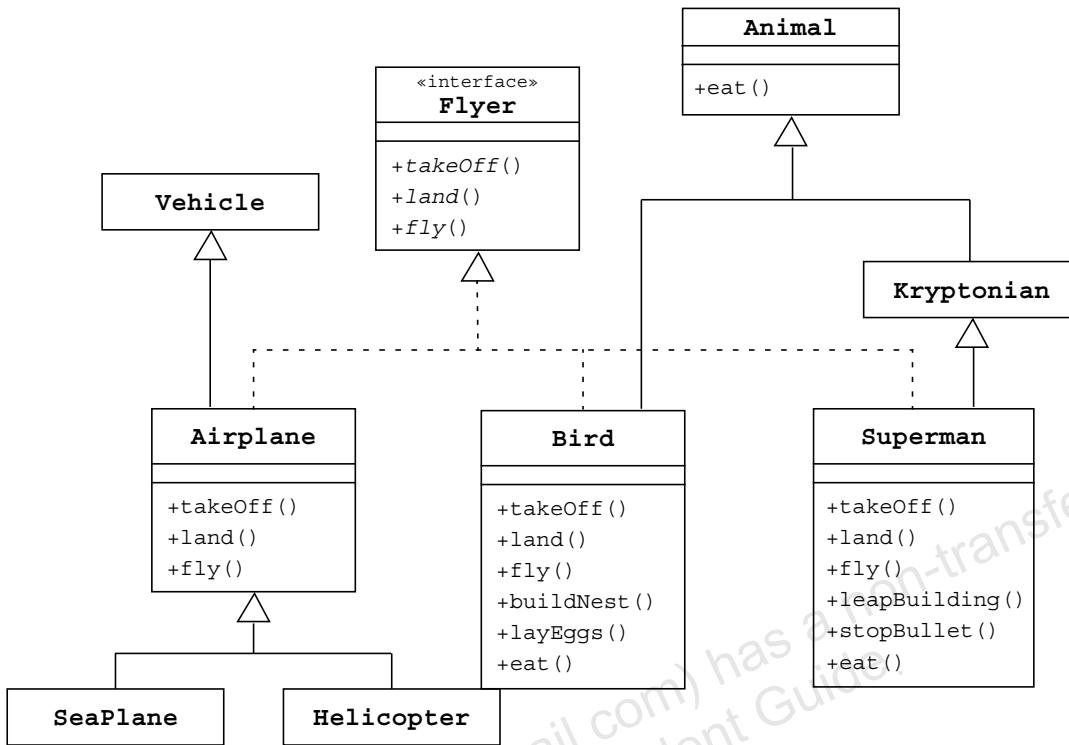
```

public class Bird extends Animal implements Flyer {
 public void takeOff() { /* take-off implementation */ }
 public void land() { /* landing implementation */ }
 public void fly() { /* fly implementation */ }
 public void buildNest() { /* nest building behavior */ }
 public void layEggs() { /* egg laying behavior */ }
 public void eat() { /* override eating behavior */ }
}

```

The `extends` clause must come before the `implements` clause. The Bird class can supply its own methods (`buildNest` and `layEggs`), as well as override the Animal class methods (`eat`).

Suppose that you are constructing an aircraft control software system. It must grant permission to land and take off for flying objects of all types. This is shown in Figure 7-7.



**Figure 7-7** Class Hierarchy for the Airport Example

The code for the airport could look like the following:

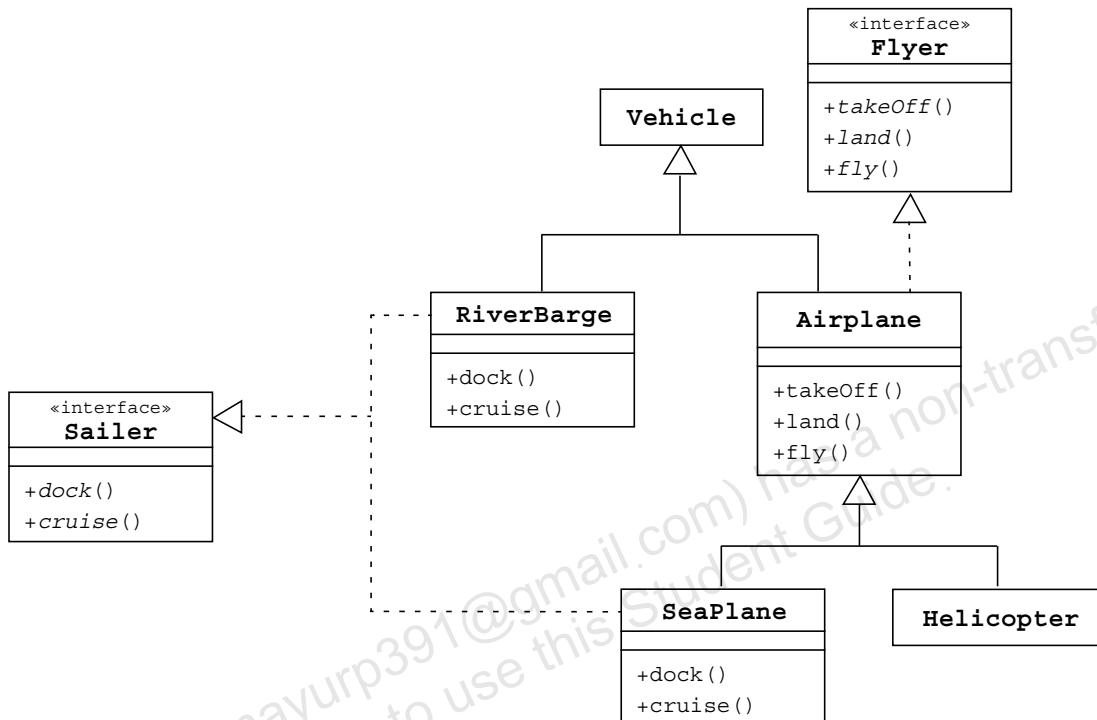
```

1 public class Airport {
2 public static void main(String[] args) {
3 Airport metropolisAirport = new Airport();
4 Helicopter copter = new Helicopter();
5 SeaPlane sPlane = new SeaPlane();
6
7 metropolisAirport.givePermissionToLand(copter);
8 metropolisAirport.givePermissionToLand(sPlane);
9 }
10
11 private void givePermissionToLand(Flyer f) {
12 f.land();
13 }
14 }

```

## Multiple Interface Example

A class can implement more than one interface. Not only can the SeaPlane fly, but it can also sail. The SeaPlane class extends the Airplane class, so it inherits that implementation of the Flyer interface. The SeaPlane class also implements the Sailer interface. This is shown in Figure 7-8.



**Figure 7-8** An Example of Multiple Implementations

Now write the Harbor class to give docking permission:

```

1 public class Harbor {
2 public static void main(String[] args) {
3 Harbor bostonHarbor = new Harbor();
4 RiverBarge barge = new RiverBarge();
5 SeaPlane sPlane = new SeaPlane();
6
7 bostonHarbor.givePermissionToDock(barge);
8 bostonHarbor.givePermissionToDock(sPlane);
9 }
10 private void givePermissionToDock(Sailer s) {
11 s.dock();
12 }
13 }

```

The seaplane can take off from Metropolis airport and dock in Boston harbor.

## Uses of Interfaces

You use interfaces to:

- Declare methods that one or more classes are expected to implement
- Reveal an object's programming interface without revealing the actual body of the class (this can be useful when shipping a package of classes to other developers)
- Capture similarities between unrelated classes without forcing a class relationship
- Simulate multiple inheritance by declaring a class that implements several interfaces



## Module 8

---

# Exceptions and Assertions

---

## Objectives

Upon completion of this module, you should be able to:

- Define exceptions
- Use try, catch, and finally statements
- Describe exception categories
- Identify common exceptions
- Develop programs to handle your own exceptions
- Use assertions
- Distinguish appropriate and inappropriate uses of assertions
- Enable assertions at runtime

This module covers the error handling facilities built into the Java programming language.

---

## Relevance

# Relevance

**Discussion** – The following question is relevant to the material presented in this module:

- In most programming languages, how do you resolve runtime errors?

---

---

---

- If you make assumptions about the way your code works, and those assumptions are wrong, what might happen?

---

---

---

- Is it always necessary or desirable to expend CPU power testing assertions in production programs?

---

---

---

## Exceptions and Assertions

*Exceptions* are a mechanism used by many programming languages to describe what to do when something unexpected happens. Typically, something unexpected is an error of some sort, for example when a method is invoked with unacceptable arguments, or a network connection fails, or the user asks to open a non-existent file.

*Assertions* are a way to test certain assumptions about the logic of a program. For example, if you believe that at a particular point the value of a variable will always be positive, then an assertion can test this. Assertions are used generally to test assumptions about local program logic inside a method, and usually not to test that external expectations are met.

One important feature of assertions is that they can be removed entirely from the code when it runs. This makes it possible for you to enable assertions during program development, but to not have the tests executed at runtime when the final product is delivered to a customer. This is an important difference between assertions and exceptions.

## Exceptions

---

# Exceptions

The Java programming language provides two broad categories of exceptions, known as *checked* and *unchecked* exceptions.

Checked exceptions are those that the programmer is expected to handle in the program, and that arise from external conditions that can readily occur in a working program. Examples would be a requested file not being found or a network failure.

Unchecked exceptions might arise from conditions that represent bugs, or situations that are considered generally too difficult for a program to handle reasonably. They are called *unchecked* because you are not required to check for them or do anything about them if they occur. Exceptions that arise from a category of situations that probably represent bugs are called *runtime exceptions*. An example of a runtime exception is attempting to access beyond the end of an array.

Exceptions that arise as a result of environmental issues that are rare enough or hard enough to recover from are called *errors*; an example of an error is running out of memory. Errors are also unchecked exceptions.

The `Exception` class is the base class that represents checked and unchecked exceptions. Rather than terminating the program, it is best if you write code to handle the exception and continue.

The `Error` class is the base class used for the unchecked, serious error conditions from which your program is not expected to attempt recovery. In most cases, you should let the program terminate when you encounter this type of error, although you might attempt to preserve as much of the user's work as possible.

The `RuntimeException` class is the base class that is used for the unchecked exceptions that might arise as a result of program bugs. In most cases, you should terminate the program when one of these arises. Again, attempting to preserve as much of the user's work as possible is a good idea.

When an exception occurs in your program, the method that finds the error can handle the exception itself or *throw* the exception back to its caller to signal that a problem has occurred. The calling method then has the same choices: handle the exception or throw it back to its caller. If an exception reaches the top of a thread, then that thread is killed. This scheme gives the programmer the option of writing a *handler* to deal with the exception at an appropriate point in the code.

You can determine the checked exceptions a method throws by browsing the API. Sometimes you might also find documentation describing one or more unchecked exceptions, but this is usually unnecessary, and you cannot rely on such information being listed.

## Exception Example

Code 8-1 shows a program that adds all of the command-line arguments.

### **Code 8-1 Example Program that Throws an Exception**

```

1 public class AddArguments {
2 public static void main(String args[]) {
3 int sum = 0;
4 for (int i = 0; i < args.length; i++) {
5 sum += Integer.parseInt(args[i]);
6 }
7 System.out.println("Sum = " + sum);
8 }
9 }
```

This program works fine if all of the command-line arguments are integers. For example:

```
java AddArguments 1 2 3 4
Sum = 10
```

This program fails if any of the arguments are not integers:

```
java AddArguments 1 two 3.0 4
Exception in thread "main" java.lang.NumberFormatException:
For input string: "two"at
java.lang.NumberFormatException.forInputString(NumberFormat
Exception.java:48)
at java.lang.Integer.parseInt(Integer.java:447)
at java.lang.Integer.parseInt(Integer.java:497)
at AddArguments.main(AddArguments.java:5)
```

# The try-catch Statement

The Java programming language provides a mechanism for figuring out which exception was thrown and how to recover from it. Code 8-2 demonstrates a modified AddArguments program that uses a try-catch statement to catch the exception, in this case a `java.lang.NumberFormatException`.

## Code 8-2 Example Program that Handles an Exception

```
1 public class AddArguments2 {
2 public static void main(String args[]) {
3 try {
4 int sum = 0;
5 for (int i = 0; i < args.length; i++) {
6 sum += Integer.parseInt(args[i]);
7 }
8 System.out.println("Sum = " + sum);
9 } catch (NumberFormatException nfe) {
10 System.err.println("One of the command-line "
11 + "arguments is not an integer.");
12 }
13 }
14 }
```

This program captures the exception and then quits with an error message:

```
java AddArguments2 1 two 3.0 4
One of the command-line arguments is not an integer.
```

## Fine-grained Exception Handling

The try-catch statement can be used on smaller chunks of code.

Code 8-3 shows the try-catch statement used inside of the for loop.

This enables the program to discard only those command-line arguments that are not integers. This code demonstrates a more graceful degradation in behavior than the AddArguments2 program.

### Code 8-3 A Refined AddArguments Program

```

1 public class AddArguments3 {
2 public static void main (String args[]) {
3 int sum = 0;
4 for (int i = 0; i < args.length; i++) {
5 try {
6 sum += Integer.parseInt(args[i]);
7 } catch (NumberFormatException nfe) {
8 System.err.println("[" + args[i] + "] is not an integer"
9 + " and will not be included in the sum.");
10 }
11 }
12 System.out.println("Sum = " + sum);
13 }
14 }
```

This program captures the exception for each non-integer, command-line argument and generates an warning message. However, this program proceeds to sum all of the valid integers:

```

java AddArguments3 1 two 3.0 4
[two] is not an integer and will not be included in the
sum.
[3.0] is not an integer and will not be included in the
sum.
Sum = 5
```

## Using Multiple catch Clauses

There can be multiple `catch` blocks after a `try` block, each handling a different exception type. Code 8-4 illustrates the syntax of using multiple `catch` clauses.

### **Code 8-4** Example Using Multiple catch Clauses

```
try {
 // code that might throw one or more exceptions

} catch (MyException e1) {
 // code to execute if a MyException is thrown

} catch (MyOtherException e2) {
 // code to execute if a MyOtherException is thrown

} catch (Exception e3) {
 // code to execute if any other exception is thrown
}
```

If you have multiple `catch` clauses for a `try` block, the order of the `catch` clauses matters. That is because an exception thrown from the `try` block will be handled by the first `catch` clause that can handle it. If in this example the `Exception` `catch` clause is put first, then it would handle all exceptions, and the `MyException` or `MyOtherException` `catch` clauses would never be invoked.

## Call Stack Mechanism

If a statement throws an exception, and that exception is not handled in the immediately enclosing method, then that exception is thrown to the calling method. If the exception is not handled in the calling method, it is thrown to the caller of that method. This process continues. If the exception is still not handled by the time it gets back to the `main()` method and `main()` does not handle it, the exception terminates the program abnormally.

Consider a case in which the `main()` method calls another method named `first()`, and this, in turn, calls another method named `second()`. If an exception occurs in `second()` and is not handled there, it is thrown back to `first()`. Imagine that in `first()` there is a catch for that type of exception. In this case the exception is handled and goes no further. However, if `first()` does not have any catch for this type of exception then the next method in the call stack, `main()`, is checked. If the exception is not handled in `main()`, then the exception is printed to the standard output and the program stops executing.

## The finally Clause

The `finally` clause defines a block of code that *always* executes, regardless of whether an exception was caught. The following sample code and description is taken from the white paper, *Low Level Security in Java*, by Frank Yellin:

```

1 try {
2 startFaucet();
3 waterLawn();
4 } catch (BrokenPipeException e) {
5 logProblem(e);
6 } finally {
7 stopFaucet();
8 }

```

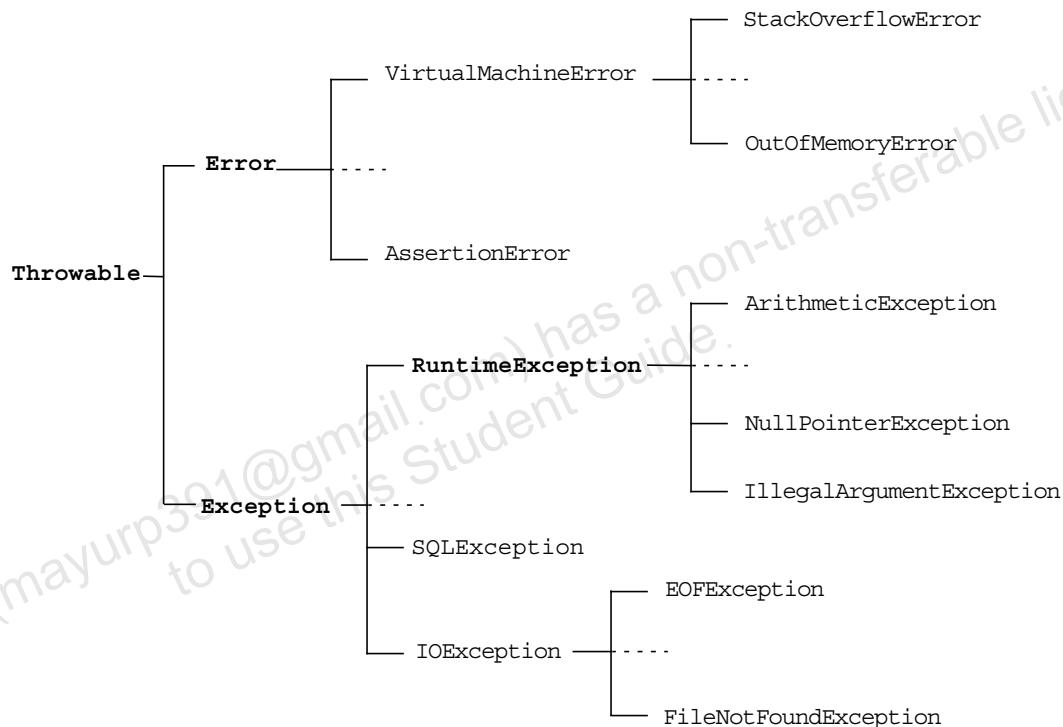
In the previous example, the faucet is turned off regardless of whether an exception occurs while starting the faucet or while watering the lawn. The code inside the braces after the `try` is called the *protected code*.

The only situations that prevent the `finally` clause executing are virtual machine shutdown (for example if the `System.exit` method is executed, or the machine is shut down or has its power turned off). This implies that the control flow can deviate from normal sequential execution. For example, if a `return` statement is embedded in the code inside the `try` block, the code in the `finally` clause executes before the `return`.

## Exception Categories

# Exception Categories

Earlier in this module you were introduced to three broad categories of exceptions. This section examines the class hierarchy that represents those categories. The class `java.lang.Throwable` acts as the parent class for all objects that can be thrown and caught using the exception-handling mechanisms. Methods defined in the `Throwable` class retrieve the error message associated with the exception and print the stack trace showing where the exception occurred. There are three key subclasses of `Throwable`: `Error`, `RuntimeException`, and `Exception`, which are shown in Figure 8-1.



**Figure 8-1** Subclasses and Exceptions

You should not use the `Throwable` class; instead, use one of the subclass exceptions to describe any particular exception. The following describes the purpose of each exception:

- The value `Error` indicates a severe problem from which recovery is difficult, if not impossible. An example is running out of memory. A program is not expected to handle such conditions.
- The value `RuntimeException` indicates a design or implementation problem. That is, it indicates conditions that should never happen if the program is operating properly. A `NullPointerException`, for example, should never be thrown if the developer remembers to associate reference variables with new or existing objects on the heap. Because a program that is designed and implemented correctly never issues this type of exception, it is usual to leave it unhandled. This results in a message at runtime, and ensures that action is taken to correct the problem, rather than hiding it where (you think) no one will notice.
- Other exceptions indicate a difficulty at runtime that is most often caused by environmental effects and can be handled. Examples include a file not found or invalid URL exceptions (the user typed a wrong URL), both of which could occur easily if the user mistyped something. Because these occur usually as a result of user error, you are encouraged to handle them.

---

## Common Exceptions

# Common Exceptions

The Java programming language provides several predefined exceptions. Some of the more common exceptions are:

- `NullPointerException`

This is an attempt to access an attribute or method of an object using a variable that does not refer to an object, for example, when the variable has not been initialized or when no object has been instantiated.

```
Employee emp = null;
System.out.println(emp.getName());
```

- `FileNotFoundException`

This exception is an attempt to read from a file that does not exist.

- `NumberFormatException`

This exception is an attempt to parse a string into a number (integer or floating point number) that has an illegal number format.

- `ArithmaticException`

This is the result of a divide-by-zero operation for integers.

```
int i = 0;
int j = 12 / i;
```

- `SecurityException`

Typically thrown in a browser, the `SecurityManager` class throws an exception for applets that attempt to perform any operation that might be dangerous to the host or its files. The following are examples of operations that can cause security exceptions:

- Access the local file system.
- Open a socket to a host that is not the same host that served the applet.
- Execute another program in a runtime environment.

## The Handle or Declare Rule

To encourage the writing of robust code, the Java programming language requires that if any checked exception (that is, a subclass of `Exception` but not a subclass of `RuntimeException`) might occur at any given point in the code, then the method that contains that point must define explicitly what action is to be taken if the problem arises.

You can do the following to satisfy this requirement:

- Handle the exception by using the `try-catch-finally` block.

Have the enclosing method handle the exception by including a `try-catch` block around the possible problem. The `catch` clause must name either the expected exception class or a superclass. This counts as handling the situation, even if the `catch` clause is empty.

- Declare exceptions that a method can throw.

Have the calling method indicate that it does not handle the exception, and that the exception is thrown back to *its* calling method. This is declaring the exception and clearly passes responsibility to declare or handle to the calling method.

A method can declare that an exception might be thrown in the body of the method with a `throws` clause as follows:

```
void trouble() throws IOException { ... }
```

Following the keyword `throws` is a list of all the exceptions that the method can throw back to its caller. Although only one exception is shown here, you can use a comma-separated list if this method throws multiple possible exceptions, like this:

```
void trouble() throws IOException, OtherException { ... }
```

You do not need to declare runtime exceptions or errors. All subclasses of `RuntimeException` and `Error` are called unchecked and do not need to be included in the `throws` clause of the method declaration.

## The Handle or Declare Rule

---

You can choose to handle runtime exceptions. Usually, most runtime exceptions indicate a programming logic bug. It is most appropriate to discover these bugs during testing and fix them before the code is put into production. Therefore, there is no need to include catch clauses for runtime exceptions. There are a few notable exceptions to this rule. For example, catching a `NumberFormatException` when parsing a user-entered string is useful.

Whether you choose to handle or declare an exception depends on whether you consider yourself or your caller a more appropriate candidate for dealing with the exception.

---

**Note** – Because the exception classes are organized into hierarchies, the same as other classes, and because you can use a class whenever a subclass is expected, you can catch *groups* of exceptions and handle them with the same catch code. For example, although there are several different types of `IOExceptions` (`EOFException`, `FileNotFoundException`, and so on), by trapping `IOException` you can also catch instances of any subclass of `IOException`.

---



## Method Overriding and Exceptions

When overriding a method that throws exceptions, the overriding method can declare only exceptions that are either the same class or a subclass of the exceptions. For example, if the superclass method throws an `IOException`, then the overriding method can throw an `IOException`, a `FileNotFoundException` (a subclass of `IOException`), but not an `Exception` (the superclass of `IOException`). You can declare fewer or more specific exceptions in the `throws` clause.

In the following example, three classes are declared: `TestA`, `TestB1`, and `TestB2`. `TestA` is the superclass of `TestB1` and `TestB2`.

```

1 public class TestA {
2 public void methodA() throws IOException {
3 // do some number crunching
4 }
5 }

1 public class TestB1 extends TestA {
2 public void methodA() throws EOFException {
3 // do some number crunching
4 }
5 }

1 public class TestB2 extends TestA {
2 public void methodA() throws Exception {
3 // do some number crunching
4 }
5 }

```

The class `TestB1` compiles because `EOFException` is a subclass of `IOException`. However, class `TestB2` fails to compile because `Exception` is a superclass of `IOException`.

It is permitted to declare that an overriding method throws fewer exceptions than the superclass method, including no exceptions at all. The new restricted set will become the limits on what might be thrown by any sub-subclasses that are created.

---

## Creating Your Own Exceptions

# Creating Your Own Exceptions

User-defined exceptions are created by extending the `Exception` class. Exception classes contain anything that a *regular* class contains. The following is an example of a user-defined exception class containing a constructor, some variables, and methods.

```
1 public class ServerTimedOutException extends Exception {
2 private int port;
3
4 public ServerTimedOutException(String message, int port) {
5 super(message);
6 this.port = port;
7 }
8
9 public int getPort() {
10 return port;
11 }
12 }
```

---

**Note –** Use the `getMessage` method, inherited from the `Exception` class, to get the reason the exception was made

---



To throw an exception that you have created, use the following syntax:

```
throw new ServerTimedOutException("Could not connect", 80);
```

Always instantiate the exception on the same line on which you throw it, because the exception can carry line number information that will be added when the exception is created.

## Throwing a User-Defined Exception

Consider a client-server program. In the client code, you try to connect to the server and expect the server to respond within five seconds. If the server does not respond, your code could throw an exception (such as a user-defined `ServerTimedOutException`) as follows:

```
1 public void connectMe(String serverName)
2 throws ServerTimedOutException {
3 boolean successful;
4 int portToConnect = 80;
5
6 successful = open(serverName, portToConnect);
7
8 if (! successful) {
9 throw new ServerTimedOutException("Could not connect",
10 portToConnect);
11 }
12 }
```

## Handling a User-Defined Exception

To catch your exception, use the `try` statement:

```
1 public void findServer() {
2 try {
3 connectMe(defaultServer);
4 } catch (ServerTimedOutException e) {
5 System.out.println("Server timed out, trying alternative");
6 try {
7 connectMe(alternativeServer);
8 } catch (ServerTimedOutException e1) {
9 System.out.println("Error: " + e1.getMessage() +
10 " connecting to port " + e1.getPort());
11 }
12 }
13 }
```

---

**Note** – You can nest the `try` and `catch` blocks, as shown in the previous example.

---



You can also process an exception partially and then throw it; for example:

```
try {
 connectMe(defaultServer);
} catch (ServerTimedOutException e) {
 System.out.println("Error caught and rethrown");
 throw e;
}
```

## Assertions

Two syntactic forms are permitted for assertion statements, these are:

```
assert <boolean_expression> ;
assert <boolean_expression> : <detail_expression> ;
```

In either case, if the Boolean expression evaluates to `false`, then an `AssertionError` is thrown. This error should not be caught, and the program should be terminated abnormally.

If the second form is used, then the second expression, which can be of any type, is converted to a `String` and used to supplement the message that prints when the assertion is reported.




---

**Note** – Because the `assert` keyword is relatively new, the assertion mechanism can break code that was written for the JDK version 1.3 or older if it uses the word `assert` as a variable name or label. You can use the `-source 1.3` option to tell the `javac` command to compile the code for the JDK 1.3 version.

---

## Recommended Uses of Assertions

Assertions can provide valuable documentation about the programmer's assumptions and expectations. Because of this, assertions can be valuable especially during maintenance when other programmers are working on the code.

Assertions should be used generally to verify the internal logic of a single method or a small group of tightly coupled methods. Generally, assertions should not be used to verify that code is being used correctly, but rather that its own internal expectations are being met.

Examples of good uses of assertions are internal invariants, control flow invariants, postconditions, and class invariants.

### Internal Invariants

An internal invariant exists when you believe that a situation is always or never the case and you code accordingly. For example consider this code:

```
1 if (x > 0) {
```

## Assertions

---

```

2 // do this
3 } else {
4 // do that
5 }

```

If you have made the assumption that `x` has the value 0, and is never negative, and yet `x` turns out to be negative, then the code will do something unplanned. Almost certainly the behavior will not be correct. Worse, because the code does not stop and report the problem, you will not find out about the problem until much later when far more damage has been done. The original source of the problem then becomes very difficult to determine.

In this case, adding an assertion statement to the opening of the `else` block helps ensure the correctness of your assumption, and brings the bugs to light quickly if you are wrong, or if someone changes the code so as to invalidate the assumption during maintenance. With the assertion in place, the code would look like this:

```

1 if (x > 0) {
2 // do this
3 } else {
4 assert (x == 0);
5 // do that, unless x is negative
6 }

```

## Control Flow Invariants

Control flow invariants describe assumptions similar to the assumptions of internal invariants, but they relate to the way the execution flows, rather than to the value or relationships of variables. For example, in a `switch` statement, you might believe that you have enumerated every possible value of the control variable, and that therefore a `default` statement would never be executed. This should be verified by adding an assertion statement like this:

```

1 switch (suit) {
2 case Suit.CLUBS: // ...
3 break;
4 case Suit.DIAMONDS: // ...
5 break;
6 case Suit.HEARTS: // ...
7 break;
8 case Suit.SPADES: // ...
9 break;

```

```

10 default: assert false : "Unknown playing card
11 suit";
12 }

```

## Postconditions and Class Invariants

Postconditions are assumptions about the value or relationships of variables at the completion of a method. A simple example of a postcondition test would be that after a pop method on a stack, the stack should contain one less element than when the method was called, unless the stack was already empty. This might be coded like this:

```

1 public Object pop() {
2 int size = this.getElementCount();
3 if (size == 0) {
4 throw new RuntimeException("Attempt to pop from empty stack");
5 }
6
7 Object result = /* code to retrieve the popped element */ ;
8
9 // test the postcondition
10 assert (this.getElementCount() == size - 1);
11
12 return result;
13 }

```

Notice here that if the pop method did not throw an exception if called on an empty stack, then it would be harder to express the assertion, because an original size of zero would still result in a zero final size. Also notice that the precondition test, that is the test that determines if the method is being called on an empty stack, does not use an assertion. This is because it is not an error in the local logic of the method if such a situation occurs, rather it is an error in the way the object is being used. Such tests on external behavior should, in general, not use assertions, but should use simple exceptions. This ensures that the test is always made and cannot be disabled as is the case with assertions.

A class invariant is something that can be tested at the end of every method call for the class. For the example of a stack class, an invariant condition is one in which the element count is never negative.

## Assertions

---

### Inappropriate Uses of Assertions

Do not use assertions to check the parameters of a public method. Instead the method should test each parameter and thrown an appropriate exception, such as `IllegalArgumentException` or `NullPointerException`. The reason why you should not use assertions for parameter checking is because the assertion mechanism can be turned off, but you still want to perform parameter checking.

Do not use methods in the assertion check that can cause side-effects, because the assertion checks may be turned off at runtime. If your application depends upon these side-effects, then your application will exhibit different behavior when the assertion checks are turned off.

### Controlling Runtime Evaluation of Assertions

One of the major advantages of assertions over using exceptions is that assertion checking can be disabled at runtime. If this is done, then there is no overhead involved in checking the assertions, and the code runs as fast as if the assertion test had never been present. This is better than using conditional compilation for two reasons. First, it does not require a different compile phase for production. Second, if required, the assertions can be re-enabled in the field to determine if the assertions have become invalid due to some unforeseen environmental effect.

By default, assertions are disabled. To turn assertions on, use either of these forms:

```
java -enableassertions MyProgram
java -ea MyProgram
```

Assertions can be controlled on packages, package hierarchies, or individual classes. For information on this, consult the documentation at: [docs/guide/language/assert.html](http://docs/guide/language/assert.html) under the installation of your JDK software.

## Module 9

---

# Collections and Generics Framework

---

## Objectives

Upon completion of this module, you should be able to:

- Describe the general purpose implementations of the core interfaces in the Collections framework
- Examine the Map interface
- Examine the legacy collection classes
- Create natural and custom ordering by implementing the Comparable and Comparator interfaces
- Use generic collections
- Use type parameters in generic classes
- Refactor existing non-generic code
- Write a program to iterate over a collection
- Examine the enhanced for loop

---

## Additional Resources



**Additional resources** – The following references provide additional information on the topics described in this module:

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.

# The Collections API

A *collection* is a single object managing a group of objects. The objects in the collection are called *elements*. Typically, collections deal with many types of objects, all of which are of a particular kind (that is, they all descend from a common parent type).

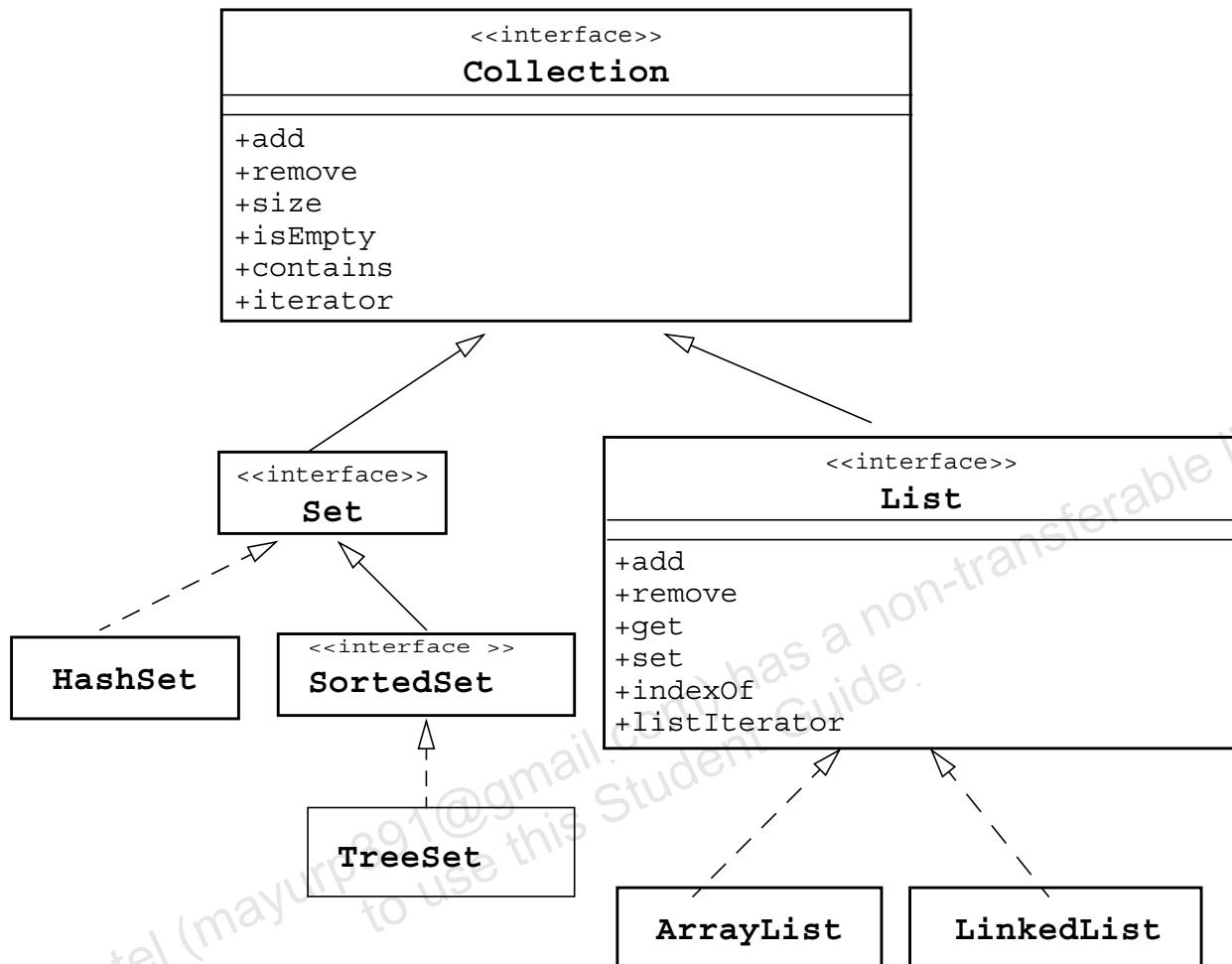
The Collections API contains interfaces that group objects as one of the following:

- Collection – A group of objects known as elements; implementations determine whether there is specific ordering and whether duplicates are permitted
- Set – An unordered collection; no duplicates are permitted
- List – An ordered collection; duplicates are permitted

Before the release of the Java SE 5.0 platform, collections maintained references to objects of type `Object`. This enables any object to be stored in the collection. It also necessitates the use of correct casting before you can use the object, after retrieving it from the collection. However, with the Java SE 5.0 platform and onwards, you can use generic collection features to specify the object type to be stored in a collection. This avoids the need to cast the object on retrieval. Generic collections are discussed in more detail in “Generics” on page 9-17.

## The Collections API

Figure 9-1 shows a UML diagram of the primary interfaces and implementation classes of the Collections API.



**Figure 9-1** The Collection Interface and Class Hierarchy

The **HashSet** is one example of a class that supplies an implementation of the **Set** interface. The **SortedSet** interface extends the **Set** interface. The classes that implement **SortedSet** impose total ordering on its elements. **TreeSet** implements the **SortedSet** interface. The **ArrayList** and **LinkedList** classes supply an implementation of the **List** interface.

**Note** – This discussion of the Collections API is a simplification of the complete API (which includes many more methods, more interfaces, and several intermediate abstract classes). For more information, read *Introduction to the Collections Framework* at the following URL:  
<http://developer.java.sun.com/developer/onlineTraining/collections/>



# Collection Implementations

There are several general purpose implementations of the core interfaces (Set, List, Map and Deque) in the Collections framework. Table 9-1 shows some of the concrete implementations of these interfaces.

**Table 9-1** General Purpose Collection Implementations

|       | Hash Table | Resizable Array | Balanced Tree | Linked List | Hash Table + Linked List |
|-------|------------|-----------------|---------------|-------------|--------------------------|
| Set   | HashSet    |                 | TreeSet       |             | LinkedHashSet            |
| List  |            | ArrayList       |               | LinkedList  |                          |
| Deque |            | ArrayDeque      |               | LinkedList  |                          |
| Map   | HashMap    |                 | TreeMap       |             | LinkedHashMap            |

The following subsections show examples of the use of HashSet and ArrayList.

## A Set Example

In the following example shown in Code 9-1, the program declares a variable (`set`) of type Set that is initialized to a new HashSet object. It then adds a few elements and prints the set to standard output. Lines 10 and 11 in Code 9-1 attempt to add duplicate values to `set`. Because duplicate values cannot be added to a Set, the add methods return false.

**Code 9-1** SetExample Program

```

1 import java.util.*;
2 public class SetExample {
3 public static void main(String[] args) {
4 Set set = new HashSet();
5 set.add("one");
6 set.add("second");
7 set.add("3rd");
8 set.add(new Integer(4));
9 set.add(new Float(5.0F));
10 set.add("second"); // duplicate, not added
11 set.add(new Integer(4)); // duplicate, not added
12 System.out.println(set);
13 }
14 }
```



The output generated from this program might be:

[one, second, 5.0, 3rd, 4]

You should note that the order of the elements is not the same as the order in which they were added.

---

**Note** – In Line 13, the program prints the set object to standard output. This works because the HashSet class overrides the inherited `toString` method and creates a comma-separated sequence of the items delimited by the open and close braces.

---

## A List Example

In the following example shown in Code 9-2, the program declares a variable (`list`) of type `List` that is assigned to a new `ArrayList` object. It then adds a few elements and prints the list to standard output. Because lists allow duplicates, the `add` methods in lines 10 and 11 in Code 9-2 return `true`.

### Code 9-2 ListExample Program

```
1 import java.util.*;
2 public class ListExample {
3 public static void main(String[] args) {
4 List list = new ArrayList();
5 list.add("one");
6 list.add("second");
7 list.add("3rd");
8 list.add(new Integer(4));
9 list.add(new Float(5.0F));
10 list.add("second"); // duplicate, is added
11 list.add(new Integer(4)); // duplicate, is added
12 System.out.println(list);
13 }
14 }
```

The output generated from this program is:

[one, second, 3rd, 4, 5.0, second, 4]

The order of the elements is the order in which they were added.



**Note** – The Collections API contains many useful, concrete implementations of the Collection, Set, and List interfaces. You are encouraged to check the API documentation and become familiar with the implementations. Some even implement hybrid behavior, such as the LinkedHashMap, which uses hashing to implement fast searches, and also maintains a doubly linked list internally so that it can return objects from an iterator in a meaningful order.

---

## The Map Interface

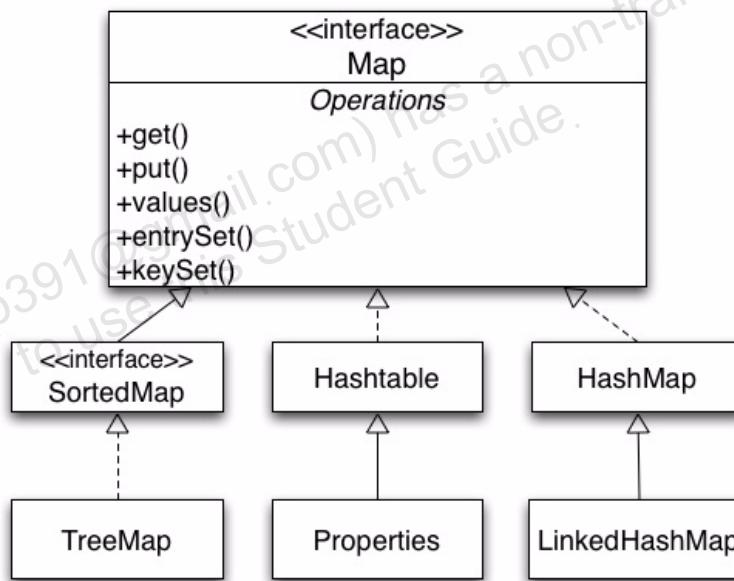
# The Map Interface

Maps are sometimes called associative arrays. A Map object describes mappings from keys to values. By definition, a Map object does not allow duplicate keys and a key can map to one value at most.

The Map interface provides three methods that allow map contents to be viewed as collections

- entrySet – Returns a Set that contains all the key value pairs.
- keySet – Returns a Set of all the keys in the map.
- values – Returns a Collection containing all the values contained in the map.

Figure 9-2 shows the API for the Map interface, its sub-interfaces and a few of the more well known implementing classes.



**Figure 9-2** The Map Interlace API

The Map interface does not extend the Collection interface because it represents mappings and not a collection of objects. The SortedMap interface extends the Map interface. Some of the classes that implement the Map interface are HashMap, TreeMap, IdentityHashMap, and WeakHashMap. The order presented by the iterators of these map collection implementations is specific to each iterator.

## A Map example

In the example shown in Code 9-3, the program declares a variable `map` of type `Map`, and assigns it to a new `HashMap` object. It then adds a few elements to the map by using the `put` operation. To prove that duplicate keys are not allowed in a map, the program attempts to add a new value using an existing key. This results in the previously added value for the key being replaced by the new value. The program later uses the collection view operations `keySet`, `values`, and `entrySet` for retrieving the contents of the map.

### Code 9-3 MapExample Program

```
1 import java.util.*;
2 public class MapExample {
3 public static void main(String args[]) {
4 Map map = new HashMap();
5 map.put("one", "1st");
6 map.put("second", new Integer(2));
7 map.put("third", "3rd");
8 // Overwrites the previous assignment
9 map.put("third", "III");
10 // Returns set view of keys
11 Set set1 = map.keySet();
12 // Returns Collection view of values
13 Collection collection = map.values();
14 // Returns set view of key value mappings
15 Set set2 = map.entrySet();
16 System.out.println(set1 + "\n" + collection + "\n" + set2);
17 }
18 }
```

The generated output from the program is:

```
[second, one, third]
[2, 1st, III]
[second=2, one=1st, third=III]
```

## Legacy Collection Classes

The collection classes in the JDK version 1.0 and JDK version 1.1 still exist in the current JDK with the same interface, but they have been retooled to interact with the new Collections API.

- The `Vector` class implements the `List` interface.
- The `Stack` class is an extension of `Vector` that adds the typical stack operations: `push`, `pop`, and `peek`.
- The `Hashtable` is an implementation of `Map`.
- The `Properties` class is an extension of `Hashtable` that only uses `Strings` for keys and values.
- Each of these collections has an `elements` method that returns an `Enumeration` object. The `Enumeration` is an interface similar to, but incompatible with, the `Iterator` interface. For example, `hasNext` is replaced by `hasMoreElements` in the `Enumeration` interface.

# Ordering Collections

The Comparable and Comparator interfaces are useful for ordering collections. The Comparable interface imparts natural ordering to classes that implement it. The Comparator interface is used to specify order relation. It can also be used to override natural ordering. These interfaces are useful for sorting the elements in a collection.

## The Comparable Interface

The Comparable interface is a member of the `java.lang` package. When you declare a class, the JVM implementation has no means of determining the order you intend for objects of that class. You can, by implementing the Comparable interface, provide order to the objects of any class. You can sort collections that contain objects of classes that implement the Comparable interface.

Examples of some Java classes that implement the Comparable interface, are `Byte`, `Long`, `String`, `Date`, and `Float`. Of these, the numeric classes use a numeric implementation; the `String` class uses an alphabetic implementation and the `Date` class uses a chronological implementation. Passing an `ArrayList` containing `String` type elements to the static `sort` method of the `Collections` class returns a list sorted in alphabetical order. A list containing `Date` type elements sorts in chronological order and a list containing `Integer` type elements sorts in numerical order.

To write custom Comparable types, you need to implement the `compareTo` method of the Comparable interface. Code 9-4 shows how to implement the Comparable interface. The `Student` class implements the Comparable interface so that the objects of this class can be compared to each other based on grade point average (GPA).

### Code 9-4 Example Comparable Interface Implementation

```
1 class Student implements Comparable {
2 String firstName, lastName;
3 int studentID=0;
4 double GPA=0.0;
5 public Student(String firstName, String lastName, int studentID,
6 double GPA) {
7 if (firstName == null || lastName == null || studentID == 0
8 || GPA == 0.0) {throw new IllegalArgumentException();}
9 this.firstName = firstName;
10 this.lastName = lastName;
```

## Ordering Collections

---

```

11 this.studentID = studentID;
12 this.GPA = GPA;
13 }
14 public String firstName() { return firstName; }
15 public String lastName() { return lastName; }
16 public int studentID() { return studentID; }
17 public double GPA() { return GPA; }
18 // Implement compareTo method.
19 public int compareTo(Object o) {
20 double f = GPA - ((Student)o).GPA;
21 if (f == 0.0)
22 return 0; // 0 signifies equals
23 else if (f < 0.0)
24 return -1; // negative value signifies less than or before
25 else
26 return 1; // positive value signifies more than or after
27 }
28 }
```

The StudentList program in Code 9-5 tests the Comparable interface implementation created in the Code 9-4 on page 9-11. The StudentList program creates four Student objects and prints them. Because the Student objects are added to the TreeSet, which is a sorted set, the objects are sorted.

### **Code 9-5 StudentList Program**

```

1 import java.util.*;
2 public class ComparableTest {
3 public static void main(String[] args) {
4 TreeSet studentSet = new TreeSet();
5 studentSet.add(new Student("Mike", "Hauffmann", 101, 4.0));
6 studentSet.add(new Student("John", "Lynn", 102, 2.8));
7 studentSet.add(new Student("Jim", "Max", 103, 3.6));
8 studentSet.add(new Student("Kelly", "Grant", 104, 2.3));
9 Object[] studentArray = studentSet.toArray();
10 Student s;
11 for(Object obj : studentArray) {
12 s = (Student) obj;
13 System.out.printf("Name = %s %s ID = %d GPA = %.1f\n",
14 s.firstName(), s.lastName(), s.studentID(), s.GPA());
15 }
16 }
17 }
```

The output generated by the StudentList program is

```
Name = Kelly Grant ID = 104 GPA = 2.3
Name = John Lynn ID = 102 GPA = 2.8
Name = Jim Max ID = 103 GPA = 3.6
Name = Mike Hauffmann ID = 101 GPA = 4.0
```

Observe that the students are compared based on the GPA. This happens because while ordering the elements in the TreeSet collection, the TreeSet looks if the objects have their natural order, and in this case, uses the `compareTo` method to compare the objects.

Some collections, such as TreeSet, are sorted. The TreeSet implementation needs to know how to order elements. If the elements have a natural order, TreeSet uses the natural order. Otherwise, you have to assist it. For example, the TreeSet class has the following constructor that takes Comparator as a parameter.

```
TreeSet(Comparator comparator)
```

This constructor creates a new, empty tree set, sorted according to the specified Comparator. The following section provides a detailed discussion of the use of the Comparator interface.

## The Comparator Interface

The Comparator interface provides greater flexibility with ordering. For example, if you consider the Student class described previously, the sorting of students was restricted to sorting on GPAs. It was not possible to sort the student based on first name or some other criteria. This section demonstrates how sorting flexibility can be enhanced using the Comparator interface.

The Comparator interface is a member of the `java.util` package. It is used to compare objects in the custom order instead of the natural order. For example, it can be used to sort objects in an order other than the natural order. It is also used to sort objects that do not implement the Comparable interface.

To write a custom Comparator, you need to provide an implementation for the `compare` method in the interface:

```
int compare(Object o1, Object o2)
```

## Ordering Collections

---

This method compares two arguments for order and returns a negative integer if the first argument is less than second, returns zero if both are equal, and returns a positive integer if the first argument is greater than the second. Code 9-6 shows the version of the Student class.

### Code 9-6 Student Class

```

1 class Student {
2 String firstName, lastName;
3 int studentID=0;
4 double GPA=0.0;
5 public Student(String firstName, String lastName,
6 int StudentID, double GPA) {
7 if (firstName == null || lastName == null || StudentID == 0 ||
8 GPA == 0.0) throw new IllegalArgumentException();
9 this.firstName = firstName;
10 this.lastName = lastName;
11 this.studentID = studentID;
12 this.GPA = GPA;
13 }
14 public String firstName() { return firstName; }
15 public String lastName() { return lastName; }
16 public int studentID() { return studentID; }
17 public double GPA() { return GPA; }
18 }
```

Several classes can be created to compare the students based on firstName, or lastName, or studentID, or GPA. The class NameComp in Code 9-7 implements the Comparator interface to compare students based on firstName.

### Code 9-7 Example Comparator Interface Implementation

```

1 import java.util.*;
2 public class NameComp implements Comparator {
3 public int compare(Object o1, Object o2) {
4 return
5 (((Student)o1).firstName.compareTo(((Student)o2).firstName));
6 }
7 }
```

The class GradeComp in Code 9-8 on page 9-15 implements the Comparator interface to compare students based on their GPAs.

**Code 9-8 Another Example Comparator Interface Implementation.**

```

1 import java.util.*;
2 public class GradeComp implements Comparator {
3 public int compare(Object o1, Object o2) {
4 if (((Student)o1).GPA == ((Student)o2).GPA)
5 return 0;
6 else if (((Student)o1).GPA < ((Student)o2).GPA)
7 return -1;
8 else
9 return 1;
10 }
11 }
```

The ComparatorTest class in Code 9-9 tests the NameComp comparator. In line 4 notice that the name comparator is passed as a parameter to TreeSet

**Code 9-9 ComparatorTest Program.**

```

1 import java.util.*;
2 public class ComparatorTest {
3 public static void main(String[] args) {
4 Comparator c = new NameComp();
5 TreeSet studentSet = new TreeSet(c);
6 studentSet.add(new Student("Mike", "Hauffmann", 101, 4.0));
7 studentSet.add(new Student("John", "Lynn", 102, 2.8));
8 studentSet.add(new Student("Jim", "Max", 103, 3.6));
9 studentSet.add(new Student("Kelly", "Grant", 104, 2.3));
10 Object[] studentArray = studentSet.toArray();
11 Student s;
12 for(Object obj : studentArray){
13 s = (Student) obj;
14 System.out.printf("Name = %s %s ID = %d GPA = %.1f\n",
15 s.firstName(), s.lastName(), s.studentID(), s.GPA());
16 }
17 }
18 }
```

The output generated by this program is:

```

Name = Jim Max ID = 103 GPA = 3.6
Name = John Lynn ID = 102 GPA = 2.8
Name = Kelly Grant ID = 104 GPA = 2.3
Name = Mike Hauffmann ID = 101 GPA = 4.0
```

## Ordering Collections

If Code 9-9 was modified to use a GradeComp object, students would be sorted based on the GPA.

# Generics

Collection classes use the `Object` type to permit different input and return types. You need to cast down explicitly to retrieve the object you need. This is not type-safe.

Although the existing collections framework does support homogeneous collections (that is collections of one type of object, for example `Date` objects), there was no mechanism to prevent other object types from being inserted into the collection. Also a retrieval almost always required a cast.

The solution for this problem is to make use of *generics* functionality. This was introduced in the Java SE 5.0 platform. It provides information for the compiler about the type of collection used. Hence, type checking is resolved automatically at run time. This eliminates the explicit casting of the data types to be used in the collection. With the addition of autoboxing of primitive types, you can use generics to write simpler and more understandable code.

Before generics, code might look like the following in Code 9-10:

### **Code 9-10** Using Non-generic Collections

```
ArrayList list = new ArrayList();
list.add(0, new Integer(42));
int total = ((Integer)list.get(0)).intValue();
```

In Code 9-10, you need an `Integer` wrapper class for typecasting while retrieving the integer value from the `list`. At runtime, the program needs the type checking for the `list`.

With the application of generics, the `ArrayList` should be declared as `ArrayList<Integer>`, to inform the compiler of the type of collection to be used. When retrieving the value, there is no need for an `Integer` wrapper class. The use of generics for the original code in Code 9-10 is shown in Code 9-11:

### **Code 9-11** Using Generic Collections

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = list.get(0).intValue();
```

## Generics

---

The autoboxing facility fits well with the Generics API. Using autoboxing, the code example could be written as shown in Code 9-12.

### Code 9-12 Using Autoboxing With Collections

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, 42);
int total = list.get(0);
```

Code 9-12 uses the generic declaration and instantiation to declare and instantiate an `ArrayList` instance of `Integer` objects. Consequently, the addition of a non-`Integer` type to the array list generates a compilation error.

**Note** – Generics are enabled by default since the Java SE 5.0 platform. You can disable generics by using the `-source 1.4` option on the `javac` command.



## Generic Set Example

In the following example, the program declares a variable (`set`) of type `Set<String>` and assigns it to a new `HashSet<String>` object. It then adds a few `String` elements and prints the set to standard output. Line 8 causes a compilation error (because an `Integer` is not a `String`). Code 9-13 shows the `Set` implementation using generics. This can be compared to the non-generic `Set` implementation shown in Code 9-1 on page 9-5.

### Code 9-13 Generic Set Example

```
1 import java.util.*;
2 public class GenSetExample {
3 public static void main(String[] args) {
4 Set<String> set = new HashSet<String>();
5 set.add("one");
6 set.add("second");
7 set.add("3rd");
8 // This line generates compile error
9 set.add(new Integer(4));
10 // Duplicate, not added
11 set.add("second");
12 System.out.println(set);
13 }
14 }
```

## Generic Map Example

The MapPlayerRepository class in Code 9-14 shows a more practical way to use generic collections. The program creates a repository of players. It declares a variable (`players`) of type `HashMap<String, String>` which stores players based on their position and name. It defines two methods `put()` and `get()` to add the elements to the map repository and to retrieve the elements from the map repository respectively.

### Code 9-14 Generic Map Implementation

```

1 import java.util.*;
2
3 public class MapPlayerRepository {
4 HashMap<String, String> players;
5
6 public MapPlayerRepository() {
7 players = new HashMap<String, String> ();
8 }
9
10 public String get(String position) {
11 String player = players.get(position);
12 return player;
13 }
14
15 public void put(String position, String name) {
16 players.put(position, name);
17 }
18
19 public static void main(String[] args) {
20 MapPlayerRepository dreamteam = new MapPlayerRepository();
21
22 dreamteam.put("forward", "henry");
23 dreamteam.put("rightwing", "ronaldo");
24 dreamteam.put("goalkeeper", "cech");
25 System.out.println("Forward is " + dreamteam.get("forward"));
26 System.out.println("Right wing is "+ dreamteam.get("rightwing"));
27 System.out.println("Goalkeeper is "+dreamteam.get("goalkeeper"));
28 }
29 }
```

The program produces the following output.

```

Forward is henry
Right wing is ronaldo
Goalkeeper is cech
```

# Generics: Examining Type Parameters

This section examines the use of type parameters in (the class, constructor and method declarations of) generic classes. Table 9-2 compares the non-generic version (pre-Java SE 5.0 platform) and the generic version (since the Java SE 5.0 platform) of the `ArrayList` class.

**Table 9-2** Comparing the non Generic and Generic `ArrayList` Classes

| Category                      | Non Generic Class                                                                | Generic Class                                                                                                |
|-------------------------------|----------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|
| Class declaration             | <code>public class ArrayList<br/>extends AbstractList<br/>implements List</code> | <code>public class ArrayList&lt;E&gt;<br/>extends AbstractList&lt;E&gt;<br/>implements List &lt;E&gt;</code> |
| Constructor declaration       | <code>public ArrayList<br/>(int capacity)</code>                                 | <code>public ArrayList<br/>(int capacity)</code>                                                             |
| Method declaration            | <code>public void add(Object o)<br/><br/>public Object get(int index)</code>     | <code>public void add(E o)<br/><br/>public E get(int index)</code>                                           |
| Variable declaration examples | <code>ArrayList list1;<br/><br/>ArrayList list2;</code>                          | <code>ArrayList &lt;String&gt; a3;<br/><br/>ArrayList &lt;Date&gt; a4;</code>                                |
| Instance declaration examples | <code>list1 = new ArrayList(10);<br/><br/>list2 = new ArrayList(10);</code>      | <code>a3= new ArrayList&lt;String&gt; (10);<br/><br/>a4= new ArrayList&lt;Date&gt; (10);</code>              |

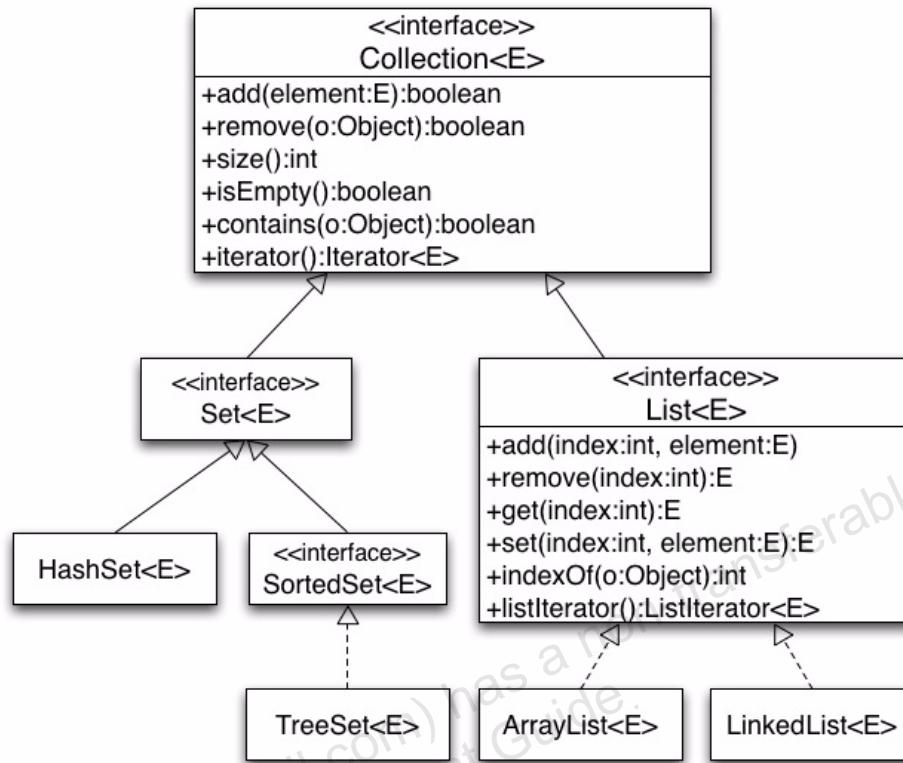
The major difference is the introduction of the type parameter `E`. In the following `ArrayList` declaration and instantiation, the `String` type substitutes the parametric type variable `E`.

```
ArrayList <String> a3 = new ArrayList <String> (10);
```

In the following `ArrayList` declaration and instantiation, the `Date` type substitutes the parametric type variable `E`.

```
ArrayList <Date> a3 = new ArrayList <Date> (10);
```

Figure 9-3 shows a UML diagram of the primary interfaces and implementation classes of the generic Collections API.



**Figure 9-3** Generic Collections API

**Note** – The type variable `E` in each interface declaration stands for the type of the elements in the collection.

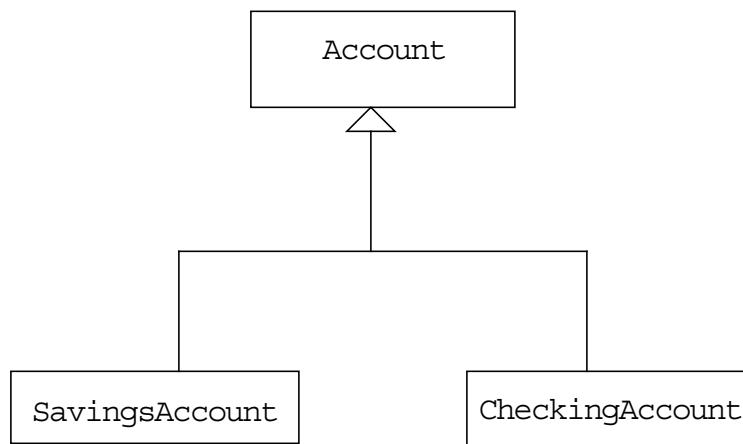
It is common practice to use upper case letters for type parameters. Java libraries use:

- E for element type of collection
- K and V for key-value pairs
- T for all the other types



## Wild Card Type Parameters

The discussions contained in this section use the inheritance hierarchy of the Account class shown in Figure 9-4.



**Figure 9-4** The Account Class and its Subclasses

This section introduces wildcard type parameters.

### The Type-Safety Guarantee

Examine the code shown in Code 9-15.

#### Code 9-15 Type-Safety Discussion Code

```

1 import com.mybank.domain.*;
2 import java.util.*;
3
4 public class TestTypeSafety {
5
6 public static void main(String[] args) {
7 List<CheckingAccount> lc = new ArrayList<CheckingAccount>();
8
9 lc.add(new CheckingAccount("Fred")); // OK
10 lc.add(new SavingsAccount("Fred")); // Compile error!
11
12 // therefore...
13 CheckingAccount ca = lc.get(0); // Safe, no cast required
14 }
15 }
```

If generic collections mean that inappropriate object types can never be added, then the guarantee is that objects retrieved from the collection can be directly and safely assigned to variables of the same type as the actual type parameter.

## The Invariance Challenge

Consider the code contained in Code 9-16.

### **Code 9-16** Invariance of Assigning Collections of Different Types

```

1 import com.mybank.domain.*;
2 import java.util.*;
3
4 public class TestInvariance {
5
6 public static void main(String[] args) {
7 List<Account> la;
8 List<CheckingAccount> lc = new ArrayList<CheckingAccount>();
9 List<SavingsAccount> ls = new ArrayList<SavingsAccount>();
10
11 //if the following were possible...
12 la = lc;
13 la.add(new CheckingAccount("Fred"));
14
15 //then the following must also be possible...
16 la = ls;
17 la.add(new CheckingAccount("Fred"));
18
19 //so...
20 SavingsAccount sa = ls.get(0); //aarrgghh!!
21 }
22 }
```

In fact, `la=lc;` is illegal, so, even though a `CheckingAccount` is an Account, an `ArrayList<CheckingAccount>` is not an `ArrayList<Account>`.

For the type-safety guarantee always to be valid, it must be impossible to assign a collection of one type to a collection of a different type, even if the second type is a subclass of the first type.

This is at odds with traditional polymorphism and would appear at first glance to make generic collections somewhat inflexible.

## The Covariance Response

Consider the code shown in Code 9-17.

### Code 9-17 Using Covariant Types

```

1 import com.mybank.domain.*;
2 import java.util.*;
3
4 public class TestCovariance {
5
6 public static void printNames(List <? extends Account> lea) {
7 for (int i=0; i < lea.size(); i++) {
8 System.out.println(lea.get(i).getName());
9 }
10 }
11
12 public static void main(String[] args) {
13 List<CheckingAccount> lc = new ArrayList<CheckingAccount>();
14 List<SavingsAccount> ls = new ArrayList<SavingsAccount>();
15
16 printNames(lc);
17 printNames(ls);
18
19 //but...
20 List<? extends Object> leo = lc; //OK
21 leo.add(new CheckingAccount("Fred")); //Compile error!
22 }
23 }
```

Wildcards allow a degree of flexibility when working with generic collections. The `printNames` method is declared with an argument that includes a wildcard. The wildcard '`?`' in `List <? extends Account>` might be interpreted as "any kind of list of unknown elements that are of type `Account` or a subclass of `Account`." The upper bound (`Account`) means that the elements in the collection can safely be assigned to an `Account` variable. Thus, the two collections of `Account` subtypes can both be passed to the `printNames` method.

This covariance response is designed to be read from, rather than be written to. Because of the invariance principle, it is illegal to add to a collection that uses a wildcard with the `extends` keyword.

# Generics: Refactoring Existing Non-Generic Code

With generic collections, you can specify generic types without type arguments, which are called raw types. This feature is allowed to provide compatibility with the non-generic code.

At compile time, all the generic information from the generic code is removed, leaving behind a raw type. This enables interoperability with the legacy code as the class files generated by the generic code and the legacy code would be the same. At runtime, `ArrayList<String>` and `ArrayList<Integer>` get translated into `ArrayList`, which is a raw type.

Using the new Java SE 5.0 or later compiler on older, non-generic code, generates a warning. Code 9-18 illustrates a class that generates the compile-time warning.

## Code 9-18 A Class That Issues a Warning

```
1 import java.util.*;
2 public class GenericsWarning {
3 public static void main(String[] args) {
4 List list = new ArrayList();
5 list.add(0, new Integer(42));
6 int total = (Integer) list.get(0);
7 }
8 }
```

If you compile the `GenericsWarning` class using the following command:

```
javac GenericsWarning.java
```

you observe the following warning:

Note: `GenericsWarning.java` uses unchecked or unsafe operations.

Note: Recompile with `-Xlint:unchecked` for details.

Alternatively, if you compile the class using the following command:

```
javac -Xlint:unchecked GenericsWarning.java
```

you observe the warning:

```
GenericsWarning.java:5: warning: [unchecked] unchecked call
to add(int,E) as a member of the raw type java.util.List
list.add(0, new Integer(42));
```

## Generics: Refactoring Existing Non-Generic Code

---

1 warning

Although the class compiles fine and the warnings can be ignored, you should heed these warnings and modify your code to be generics-friendly. To resolve this warning in the `GenericsWarning` class, you need to change line 4 to read:

```
List<Integer> list = new ArrayList<Integer>();
```

# Iterators

You can scan (iterate over) a collection using an iterator. The basic `Iterator` interface enables you to scan forward through any collection. In the case of an iteration over a set, the order is non-deterministic. The order of an iteration over a list moves forward through the list elements. A `List` object also supports a `ListIterator`, which permits you to scan the list backwards and insert or modify list elements.

---

**Note** – The order of the set is deterministic if the set is an instance of some class that guarantees the order. For example if the set is an instance of `TreeSet`, which implements `SortedSet`, the order of the set is deterministic.

---

Code 9-19 demonstrates the use of an iterator:

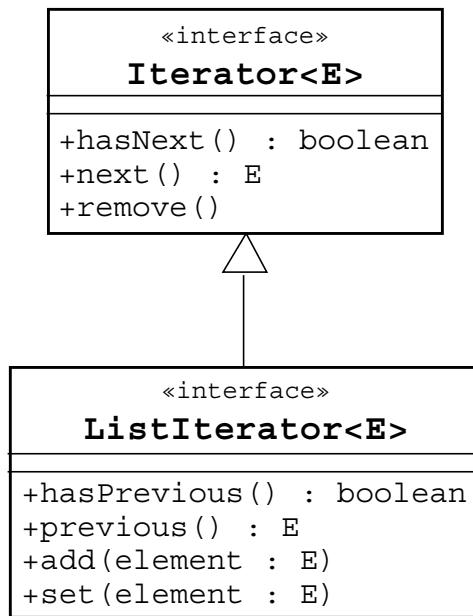
### Code 9-19 Using Iterators

```
1 List list<Student> = new ArrayList<Student>();
2 // add some elements
3 Iterator<Student> elements = list.iterator();
4 while (elements.hasNext()) {
5 System.out.println(elements.next());
6 }
```

## Iterators

---

Figure 9-5 shows a UML diagram of the generic Iterator interfaces for the Collections API.



**Figure 9-5** UML Diagram: Generic Iterator Interfaces

The `remove` method enables the code to remove the current item in the iteration (the item returned by the most recent `next` or `previous` method). If removal is not supported by the underlying collection, then an `UnsupportedOperationException` is thrown.

While using a `ListIterator`, it is common to move through the list in only one direction: either forward using `next` or backward using `previous`. If you use `previous` immediately after `next`, then you receive the same element; likewise, for calling `next` after `previous`.

The `set` method changes the element of the collection currently referenced by the iterator's cursor. The `add` method inserts the new element into the collection immediately before the iterator's cursor. Therefore, if you call `previous` after an `add`, then it returns the newly added element. However, a call to `next` is not affected. If setting or adding is not supported by the underlying collection, then an `UnsupportedOperationException` is thrown.

## The Enhanced for Loop

Code 9-20 illustrates the use of an iterator in combination with a traditional `for` loop to iterate over a collection.

### Code 9-20 Using an Iterator With Traditional `for` Loop

```
1 public void deleteAll(Collection<NameList> c) {
2 for (Iterator<NameList> i=c.iterator(); i.hasNext();) {
3 NameList nl = i.next();
4 nl.deleteItem();
5 }
6 }
```

In Code 9-20, the method `deleteAll` uses variable `i` three times in the `for` loop. This provides opportunity for coding errors to be introduced.

Alternatively, you can iterate through a collection by using an enhanced `for` loop. The enhanced `for` loop makes traversing through a collection simple, understandable, and safe. The enhanced `for` loop eliminates the usage of separate iterator methods and minimizes the number of occurrences of the iterator variable. Code 9-21 illustrates the method `deleteAll` with the enhanced `for` loop.

### Code 9-21 Iterating Using the Enhanced `for` Loop in Collections

```
1 public void deleteAll(Collection<NameList> c) {
2 for (NameList nl: c) {
3 nl.deleteItem();
4 }
5 }
```

## The Enhanced for Loop

---

The functionality of the enhanced `for` loop, makes nested `for` loops traversal simpler and easier to understand in comparison with the traditional `for` loop. One reason for this is, using enhanced `for` loops reduces the ambiguity of the variables used in the nested loops. Code 9-22 contains an example of nested enhanced `for` loops.

### Code 9-22 Nesting Enhanced for Loops

```
1 List<Subject> subjects=...;
2 List<Teacher> teachers=...;
3 List<Course> courseList = new ArrayList<Course>();
4 for (Subject subj: subjects) {
5 for (Teacher tchr: teachers) {
6 courseList.add(new Course(subj, tchr));
7 }
8 }
```

## Module 10

---

# I/O Fundamentals

---

## Objectives

Upon completion of this module, you should be able to:

- Write a program that uses command-line arguments and system properties
- Examine the Properties class
- Construct node and processing streams, and use them appropriately
- Serialize and deserialize objects
- Distinguish readers and writers from streams, and select appropriately between them

---

## Additional Resources



**Additional resources** – The following references provide additional information on the topics described in this module:

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.
- Bates, Sierra. *Head First Java, First Edition*. O'Reilly Media. 2003.

# Command-Line Arguments

When a Java technology program is launched from a terminal window, you can provide the program with zero or more *command-line arguments*. These command-line arguments allow the user to specify the configuration information for the application. These arguments are strings: either standalone tokens, such as arg1, or quoted strings, such as "another arg".

The sequence of arguments follows the name of the program class and is stored in an array of `String` objects passed to the static `main` method; Code 10-1 shows an example:

## Code 10-1 Testing the Command-Line Arguments

```
1 public class TestArgs {
2 public static void main(String[] args) {
3 for (int i = 0; i < args.length; i++) {
4 System.out.println("args[" + i + "] is: " + args[i]);
5 }
6 }
7 }
```

This program displays each command-line argument that is passed to the `TestArgs` program. For example:

```
java TestArgs arg1 arg2 "another arg"
args[0] is: arg1
args[1] is: arg2
args[2] is: another arg
```



---

**Note** – If an application requires command-line arguments other than type `String`, for example numeric values, the application should convert the `String` arguments to their respective types using the wrapper classes, such as `Integer.parseInt` method, which can be used to convert the `String` argument that represents the numeric integer to type `int`.

---

---

## System Properties

System properties are another mechanism used to parameterize a program at runtime. A *property* is a mapping between a property name and its value; both are strings. The `Properties` class represents this kind of mapping. The `System.getProperties` method returns the system properties object. The `System.getProperty(String)` method returns the string value of the property named in the `String` parameter. There is another method, `System.getProperty(String, String)`, that enables you to supply a default string value (the second parameter), which is returned if the named property does not exist.

---

**Note** – Every JVM implementation must supply a default set of properties. (Refer to the documentation for the `System.getProperties` method for details.) Moreover, a particular JVM implementation vendor can supply others.

---

There are also static methods in the wrapper classes that perform conversion of property values: `Boolean.getBoolean(String)`, `Integer.getInteger(String)`, and `Long.getLong(String)`. The string argument is the name of the property. If the property does not exist, then `false` or `null` (respectively) is returned.



# The Properties Class

An object of the Properties class contains a mapping between property names (String) and values (String). It has two main methods for retrieving a property value: `getProperty(String)` and `getProperty(String, String)`; the latter method provides the capability of specifying a default value that is returned if the named property does not exist.

You can iterate through the complete set of property names using the `propertyNames` method. By calling `getProperty` on each name, you can retrieve all of the values.

Finally, property sets can be stored and retrieved from any I/O stream using the `store` and `load` methods.

The Code 10-2 program lists the complete set of properties that exist when the program executes:

### **Code 10-2** Retrieving the System Properties

```

1 import java.util.Properties;
2
3 public class TestProperties {
4 public static void main(String[] args) {
5 Properties props = System.getProperties();
6 props.list(System.out);
7 }
8 }
```

Line 6 retrieves the set of system properties and Line 7 prints the properties using the `list` method of the `Properties` class.

**java -DmyProp=theValue TestProperties**

The following is an excerpt of the output:

```

java.runtime.name=Java (TM) SE Runtime Environment
sun.boot.library.path=C:\jse\jdk1.6.0\jre\bin
java.vm.version=1.6.0-b105
java.vm.vendor=Sun Microsystems Inc.
java.vm.name=Java HotSpot (TM) Client VM
file.encoding.pkg=sun.io
user.country=US
myProp=theValue
```

# I/O Stream Fundamentals

A *stream* is a flow of data from a *source* to a *sink*. Typically, your program is one end of that stream, and some other node (for example, a file) is the other end.

Sources and sinks are also called *input streams* and *output streams*, respectively. You can read from an input stream, but you cannot write to it. Conversely, you can write to an output stream, but you cannot read from it. Table 10-1 shows the fundamental stream classes.

**Table 10-1** Fundamental Stream Classes

| Stream         | Byte Streams | Character Streams |
|----------------|--------------|-------------------|
| Source streams | InputStream  | Reader            |
| Sink streams   | OutputStream | Writer            |

## Data Within Streams

Java technology supports two types of data in streams: raw bytes or Unicode characters. Typically, the term *stream* refers to byte streams and the terms *reader* and *writer* refer to character streams.

More specifically, character input streams are implemented by subclasses of the `Reader` class and character output streams are implemented by subclasses of the `Writer` class. Byte input streams are implemented by subclasses of the `InputStream` class and byte output streams are implemented by subclasses of the `OutputStream` class.

# Byte Streams

The following sections describe the fundamental byte streams.

## The InputStream Methods

The following three methods provide access to the data from the input stream:

```
int read()
int read(byte[] buffer)
int read(byte[] buffer, int offset, int length)
```

The first method returns an `int`, which contains either a byte read from the stream, or a `-1`, which indicates the end of file condition. The other two methods read the stream into a byte array and return the number of bytes read. The two `int` arguments in the third method indicate a sub range in the target array that needs to be filled.

---

**Note** – For efficiency, always read data in the largest practical block, or use buffered streams.

---



```
void close()
```

When you have finished with a stream, close it. If you have a *stack* of streams, use filter streams to close the stream at the top of the stack. This operation also closes the lower streams.

```
int available()
```

This method reports the number of bytes that are immediately available to be read from the stream. An actual read operation following this call might return more bytes.

```
long skip(long n)
```

This method discards the specified number of bytes from the stream.

```
boolean markSupported()
void mark(int readlimit)
void reset()
```

## Byte Streams

---

You can use these methods to perform *push-back* operations on a stream, if supported by that stream. The `markSupported()` method returns `true` if the `mark()` and `reset()` methods are operational for that particular stream. The `mark(int)` method indicates that the current point in the stream should be noted and a buffer big enough for at least the specified argument number of bytes should be allocated. The parameter of the `mark(int)` method specifies the number of bytes that can be re-read by calling `reset()`. After subsequent `read()` operations, calling the `reset()` method returns the input stream to the point you marked. If you read past the marked buffer, `reset()` has no meaning.

## The OutputStream Methods

The following methods write to the output stream:

```
void write(int)
void write(byte[] buffer)
void write(byte[] buffer, int offset, int length)
```

As with input, always try to write data in the largest practical block.

```
void close()
```

You should close output streams when you have finished with them. Again, if you have a stack and close the top one, this closes the rest of the streams.

```
void flush()
```

Sometimes an output stream accumulates writes before committing them. The `flush()` method enables you to force writes.

# Character Streams

The following sections describe the fundamental character streams.

## The Reader Methods

The following three methods provide access to the character data from the reader:

```
int read()
int read(char[] cbuf)
int read(char[] cbuf, int offset, int length)
```

The first method returns an `int`, which contains either a Unicode character read from the stream, or a `-1`, which indicates the end of file condition. The other two methods read into a character array and return the number of bytes read. The two `int` arguments in the third method indicate a sub range in the target array that needs to be filled.

---

**Note** – Use the largest practical block for efficiency.

---



```
void close()
boolean ready()
long skip(long n)
boolean markSupported()
void mark(int readAheadLimit)
void reset()
```

These methods are analogous to the input stream versions.

## The Writer Methods

The following methods write to the writer:

```
void write(int c)
void write(char[] cbuf)
void write(char[] cbuf, int offset, int length)
void write(String string)
void write(String string, int offset, int length)
```

Similar to output streams, writers include the `close` and `flush` methods.

```
void close()
void flush()
```

# Node Streams

In the Java JDK, there are three fundamental types of nodes (Table 10-2):

- Files
- Memory (such as arrays or String objects)
- Pipes (a channel from one process or thread [a light-weight process] to another; the output pipe stream of one thread is attached to the input pipe stream of another thread)

It is possible to create new node stream classes, but it requires handling native function calls to a device driver, and this is non-portable. Table 10-2 shows the node streams.

**Table 10-2** Types of Node Streams

| Type              | Character Streams                  | Byte Streams                                  |
|-------------------|------------------------------------|-----------------------------------------------|
| File              | FileReader<br>FileWriter           | FileInputStream<br>FileOutputStream           |
| Memory:<br>array  | CharArrayReader<br>CharArrayWriter | ByteArrayInputStream<br>ByteArrayOutputStream |
| Memory:<br>string | StringReader<br>StringWriter       | N/A                                           |
| Pipe              | PipedReader<br>PipedWriter         | PipedInputStream<br>PipedOutputStream         |

## A Simple Example

Code 10-3 reads characters from a file named by the first command-line argument and writes the character out to a file named by the second command-line argument. Thus, it copies the file. This is how the program might be invoked:

```
java TestNodeStreams file1 file2
```

**Code 10-3** The TestNodeStreams Program

```
1 import java.io.*;
2
3 public class TestNodeStreams {
4 public static void main(String[] args) {
```

## Node Streams

---

```
5 try {
6 FileReader input = new FileReader(args[0]);
7 try {
8 FileWriter output = new FileWriter(args[1]);
9 try {
10 char[] buffer = new char[128];
11 int charsRead;
12
13 // read the first buffer
14 charsRead = input.read(buffer);
15 while (charsRead != -1) {
16 // write buffer to the output file
17 output.write(buffer, 0, charsRead);
18
19 // read the next buffer
20 charsRead = input.read(buffer);
21 }
22
23 } finally {
24 output.close();
25 } finally {
26 input.close();
27 } catch (IOException e) {
28 e.printStackTrace();
29 }
30 }
31 }
```

As easy as this was, handling the buffer is tedious and error-prone. It turns out that there are classes that handle the buffering for you and present you with the capability to read a stream a *line at a time*. It is called a `BufferedReader` and is a type of a stream called a *processing stream*.

## Buffered Streams

The Code 10-4 performs the same function as the program in Code 10-3 on page 10-11 but uses BufferedReader and BufferedWriter.

### Code 10-4 The TestBufferedStreams Program

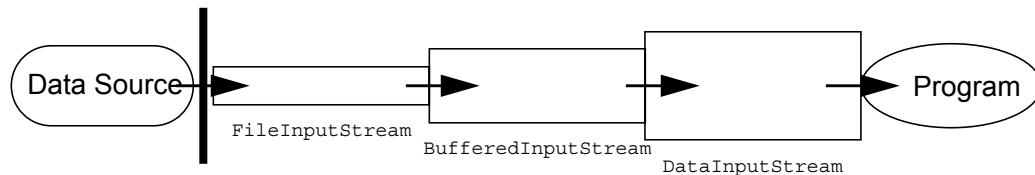
```

1 import java.io.*;
2 public class TestBufferedStreams {
3 public static void main(String[] args) {
4 try {
5 FileReader input = new FileReader(args[0]);
6 BufferedReader bufInput = new BufferedReader(input);
7 try {
8 FileWriter output = new FileWriter(args[1]);
9 BufferedWriter bufOutput= new BufferedWriter(output);
10 try {
11 String line;
12 // read the first line
13 line = bufInput.readLine();
14 while (line != null) {
15 // write the line out to the output file
16 bufOutput.write(line, 0, line.length());
17 bufOutput.newLine();
18 // read the next line
19 line = bufInput.readLine();
20 }
21 } finally {
22 bufOutput.close();
23 }
24 } finally {
25 bufInput.close();
26 }
27 } catch (IOException e) {
28 e.printStackTrace();
29 }
30 }
31 }
```

The flow of this program is the same as before. Instead of reading a buffer, this program reads a *line at a time* using the `line` variable to hold the String returned by the `readLine` method (Lines 14 and 20), which provides greater efficiency. Line 7 chains the file reader object within a buffered reader stream. You manipulate the outer-most stream in the chain (`bufInput`), which manipulates the inner-most stream (`input`).

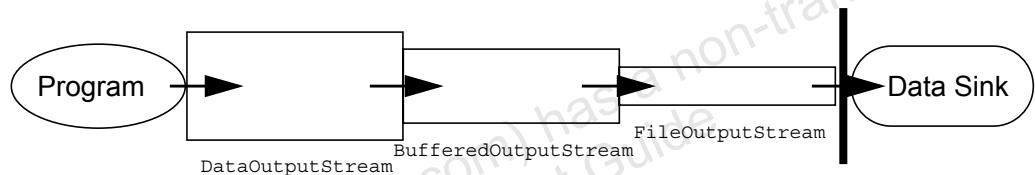
## I/O Stream Chaining

A program rarely uses a single stream object. Instead, it chains a series of streams together to process the data. Figure 10-1 demonstrates an example input stream; in this case, a file stream is *buffered* for efficiency and then converted into data (Java primitives) items.



**Figure 10-1** An Input Stream Chain Example

Figure 10-2 demonstrates an example output stream; in this case, data is written, then buffered, and finally written to a file.



**Figure 10-2** An Output Stream Chain Example

# Processing Streams

A *processing stream* performs some sort of conversion on another stream. Processing streams are also known as *filter streams*. A filter input stream is created with a connection to an existing input stream. This is done so that when you try to read from the filter input stream object, it supplies you with characters that originally came from the other input stream object. This enables you to convert the raw data into a more usable form for your application. Table 10-3 lists the built-in processing streams that are included in the `java.io` package.

**Table 10-3** List of Processing Streams by Type

| Type                                   | Character Streams                       | Byte Streams                                |
|----------------------------------------|-----------------------------------------|---------------------------------------------|
| Buffering                              | BufferedReader<br>BufferedWriter        | BufferedInputStream<br>BufferedOutputStream |
| Filtering                              | FilterReader<br>FilterWriter            | FilterInputStream<br>FilterOutputStream     |
| Converting between bytes and character | InputStreamReader<br>OutputStreamWriter |                                             |
| *Performing object serialization       |                                         | ObjectInputStream<br>ObjectOutputStream     |
| Performing data conversion             |                                         | DataInputStream<br>DataOutputStream         |
| Counting                               | LineNumberReader                        | LineNumberInputStream                       |
| Peeking ahead                          | PushbackReader                          | PushbackInputStream                         |
| Printing                               | PrintWriter                             | PrintStream                                 |

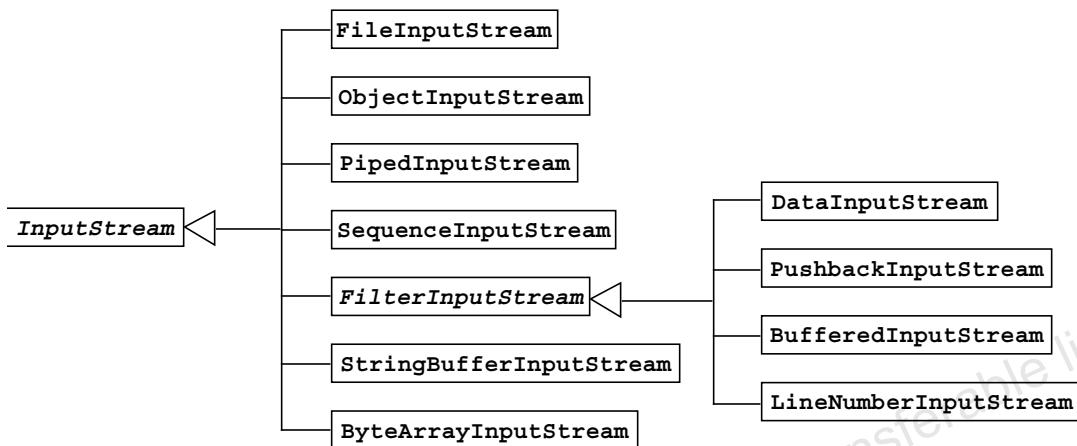
 **Note** – The `FilterXyz` streams are abstract classes and cannot be used directly. You can subclass them to implement your own processing streams.

It is easy to create new processing streams. This is described in the next section.

 **Note** – Performing object serialization is described later in this module.

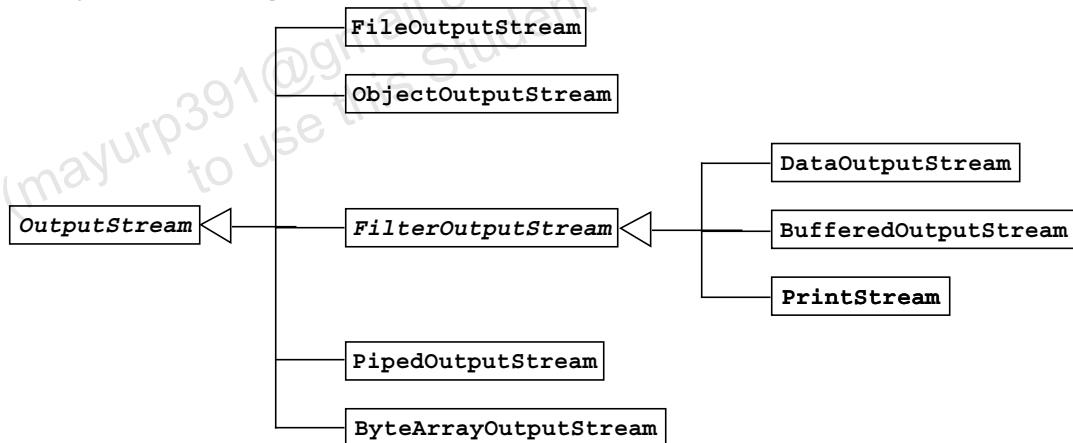
## Basic Byte Stream Classes

Figure 10-3 illustrates the hierarchy of input byte stream classes in the `java.io` package. Some of the more common byte stream classes are described in the following sections.



**Figure 10-3** The `InputStream` Class Hierarchy

Figure 10-4 illustrates the hierarchy of the output byte stream classes in the `java.io` package.



**Figure 10-4** The `OutputStream` Class Hierarchy

## The FileInputStream and FileOutputStream Classes

The `FileInputStream` and `FileOutputStream` classes are node streams and, as the name suggests, they use disk files. The constructors for these classes enable you to specify the path of the file to which they are connected. To construct a `FileInputStream`, the associated file must exist and be readable. If you construct a `FileOutputStream`, the output file is overwritten if it already exists.

```
FileInputStream infile
= new FileInputStream("myfile.dat");

FileOutputStream outfile
= new FileOutputStream("results.dat");
```

## The BufferedInputStream and BufferedOutputStream Classes

Use the `BufferedInputStream` and `BufferedOutputStream` class filter streams to increase the efficiency of I/O operations.

## The PipedInputStream and PipedOutputStream Classes

You use piped streams for communicating between threads. A `PipedInputStream` object in one thread receives its input from a complementary `PipedOutputStream` object in another thread. The piped streams must have both an input side and an output side to be useful.

## The DataInputStream and DataOutputStream Classes

The `DataInputStream` and `DataOutputStream` called filter streams enable reading and writing of Java primitive types and some special formats using streams. The following methods are provided for the different primitives.

## Basic Byte Stream Classes

---

### The DataInputStream Methods

The DataInputStream methods are as follows:

```
byte readByte()
long readLong()
double readDouble()
```

### The DataOutputStream Methods

The DataOutputStream methods are as follows:

```
void writeByte(byte)
void writeLong(long)
void writeDouble(double)
```

The methods of DataInputStream are paired with the methods of DataOutputStream.

These streams have methods for reading and writing strings but do not use these methods. They have been deprecated and replaced by readers and writers that are described later.

## The ObjectInputStream and ObjectOutputStream Classes

The ObjectInputStream and ObjectOutputStream classes enable reading from and writing Java Objects to streams.

Writing an object to a stream primarily involves writing the values of all the fields of the object. If the fields are objects themselves, these objects should also be written to the stream.

---

**Note –** If the fields are declared as transient or static, their values are not written to the stream. This is discussed in the next section.

---



Reading an object from the stream involves, reading the object type, creating the blank object of that type and filling it with the data that was written.

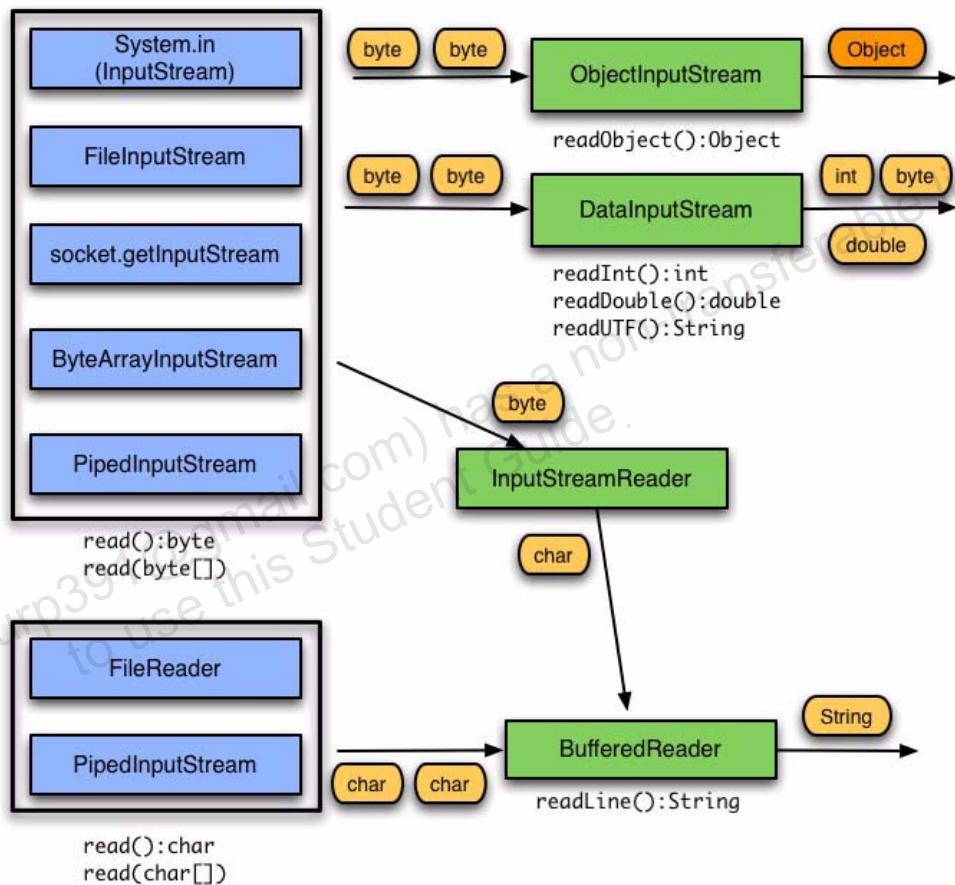
Persistent storage of objects can be accomplished if files (or other persistent storages) are used as the stream.

The Java API provides a standard mechanism that completely automates the process of writing and reading objects from streams.



**Note** – If the stream is a network socket stream, the objects are serialized before sending and deserialized after receiving on another host or process.

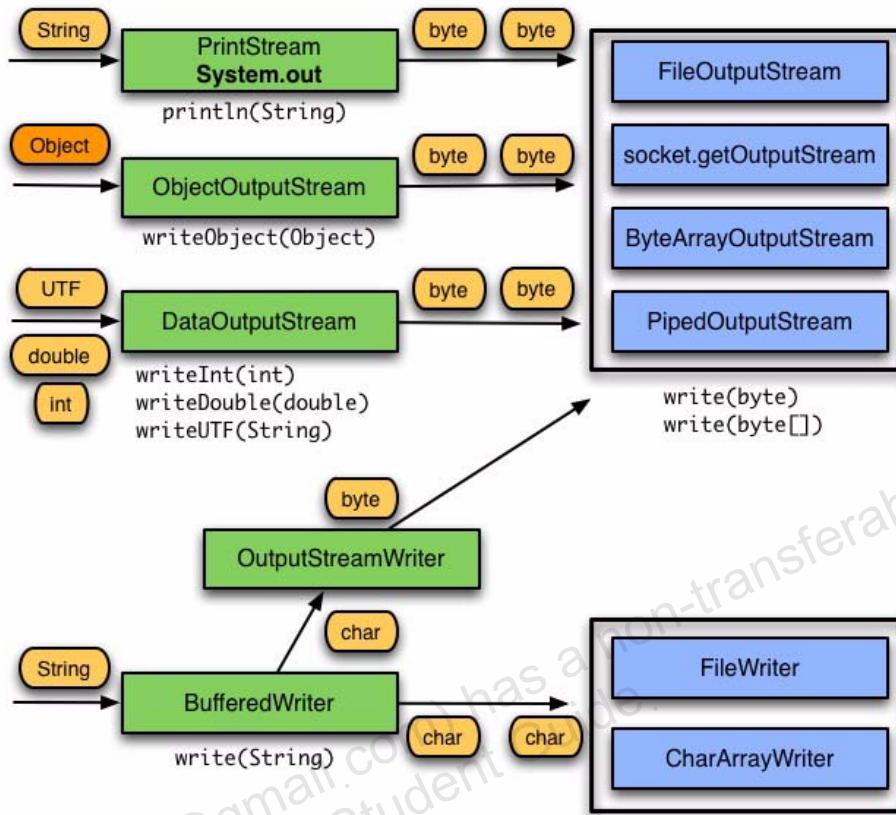
Figure 10-5 provides an overview illustration of the possible input stream and reader classes you could chain together.



**Figure 10-5** Input Chaining Combinations: A Review

## Basic Byte Stream Classes

Figure 10-6 provides an overview illustration of the possible output stream and writer classes you could chain together.



**Figure 10-6** Output Chaining Combinations: A Review

# Serialization

Saving an object to some type of permanent storage is called *persistence*. An object is said to be *persistent-capable* when you can store that object on a disk or tape or send it to another machine to be stored in memory or on disk. The non-persisted object exists only as long as the Java Virtual Machine is running.

Serialization is a mechanism for saving the objects as a sequence of bytes and later, when needed, rebuilding the byte sequence back into a copy of the object.

For objects of a specific class to be serializable, the class must implement the `java.io.Serializable` interface. The `Serializable` interface has no methods and only serves as a *marker* that indicates that the class that implements the interface can be considered for serialization.

## Serialization and Object Graphs

When an object is serialized, only the fields of the object are preserved; methods and constructors are not part of the serialized stream. When a field is a reference to an object, the fields of that referenced object are also serialized if that object's class is serializable. The tree, or structure of an object's fields, including these sub-objects, constitutes the object *graph*.

Some object classes are not serializable because the data they represent contain references to transient operating system resources. For example, `java.io.FileInputStream` and `java.lang.Thread` classes. If a serializable object contains a reference to a non-serializable element, the entire serialization operation fails and a `NotSerializableException` is thrown.

If the object graph contains a non-serializable object reference, the object can still be serialized if the reference is marked with the `transient` keyword.

Code 10-5 shows an example of a class that implements the `Serializable` interface.

### Code 10-5 The Serializable Example

```
1 public class MyClass implements Serializable {
2 public transient Thread myThread;
```

## Serialization

---

```

3 private String customerID;
4 private int total;
5 }
```

The field access modifier (public, protected, *default*, and private) has no effect on the data field being serialized. Data is written to the stream in byte format and with strings represented as file system safe universal character set transformation format (UTF) characters. The transient keyword prevents the data from being serialized.

```

1 public class MyClass implements Serializable {
2 public transient Thread myThread;
3 private transient String customerID;
4 private int total;
5 }
```

**Note** – The values stored in static fields are not serialized. When the object is deserialized the values of the static fields are set to the values stored in the corresponding class variables.

---



## Writing and Reading an Object Stream

Writing and reading an object to a stream is a simple process. This section provides examples of writing to and reading from an object stream.

### Writing

The code fragment contained in Code 10-6 sends an instance of a `java.util.Date` object to a file.

#### Code 10-6 The SerializeDate Class

```

1 import java.io.*;
2 import java.util.Date;
3
4 public class SerializeDate {
5
6 SerializeDate() {
7 Date d = new Date ();
8
9 try {
10 FileOutputStream f =
11 new FileOutputStream ("date.ser");
```

```

12 ObjectOutputStream s =
13 new ObjectOutputStream (f) ;
14 s.writeObject (d) ;
15 s.close () ;
16 } catch (IOException e) {
17 e.printStackTrace () ;
18 }
19 }
20
21 public static void main (String args[]) {
22 new SerializeDate() ;
23 }
24 }
```

The serialization starts at Line 14 when `writeObject()` method is invoked.

## Reading

Reading the object is as simple as writing it, but with one caveat—the `readObject()` method returns the stream as an `Object` type, and it must be cast to the appropriate class name before methods on that class can be executed. The Code 10-7 illustrates how to deserialize data from a stream.

### Code 10-7 The DeSerializeDate Class

```

1 import java.io.*;
2 import java.util.Date;
3
4 public class DeSerializeDate {
5
6 DeSerializeDate () {
7 Date d = null;
8
9 try {
10 FileInputStream f =
11 new FileInputStream ("date.ser");
12 ObjectInputStream s =
13 new ObjectInputStream (f);
14 d = (Date) s.readObject ();
15 s.close ();
16 } catch (Exception e) {
17 e.printStackTrace ();
18 }
19 }
```

## Serialization

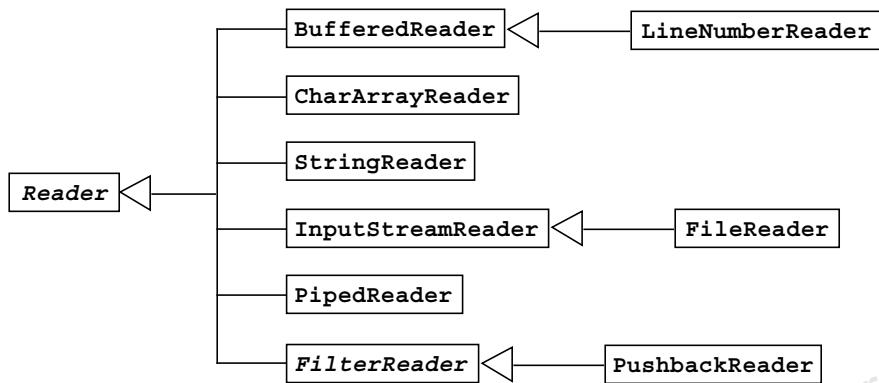
---

```
20 System.out.println(
21 "Deserialized Date object from date.ser");
22 System.out.println("Date: "+d);
23 }
24
25 public static void main (String args[]) {
26 new DeSerializeDate();
27 }
28 }
```

The object deserialization occurs at Line 14, when the `readObject()` method is invoked.

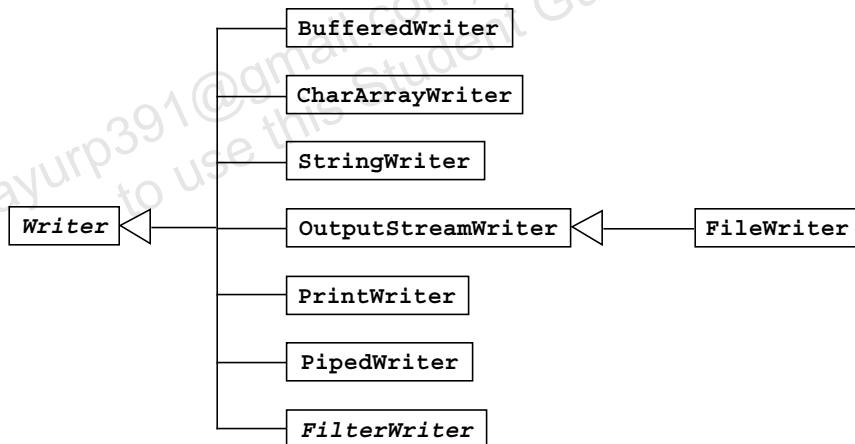
## Basic Character Stream Classes

Figure 10-7 illustrates the hierarchy of Reader character stream classes in the `java.io` package. Some of the more common character stream classes are described in the following sections.



**Figure 10-7** The Reader Class Hierarchy

Figure 10-8 illustrates the hierarchy of the Writer character stream classes in the `java.io` package.



**Figure 10-8** The Writer Class Hierarchy

## Basic Character Stream Classes

### The InputStreamReader and OutputStreamWriter Methods

The most important versions of readers and writers are `InputStreamReader` and `OutputStreamWriter`. These classes are used to interface between byte streams and character readers and writers.

When you construct an `InputStreamReader` or `OutputStreamWriter`, conversion rules are defined to change between 16-bit Unicode and other platform-specific representations.

### Byte and Character Conversions

By default, if you construct a reader or writer connected to a stream, the conversion rules change between bytes using the default platform character encoding and Unicode. In English-speaking countries, the byte encoding used is *International Organization for Standardization (ISO) 8859-1*.

Specify an alternative byte encoding by using one of the supported encoding forms. If you have the documentation installed, you can find a list of the supported encoding forms in the documentation found in the following URL:

<http://java.sun.com/javase/6/docs/technotes/guides/intl/encoding.doc.html>.

Using this conversion scheme, Java technology uses the full flexibility of the local platform character set while still retaining platform independence through the internal use of Unicode.

### Using Other Character Encoding

If you need to read input from a character encoding that is not your local one (for example, reading from a network connection with a different type of machine), you can construct the `InputStreamReader` with an explicit character encoding, such as:

```
InputStreamReader ir
= new InputStreamReader(System.in, "ISO-8859-1");
```



**Note** – If you are reading characters from a network connection, use this form. If you do not, your program will always attempt to convert the characters it reads as if they were in the local representation, which is probably not correct. ISO 8859-1 is the Latin-1 encoding scheme that maps onto ASCII.

## The FileReader and FileWriter Classes

The `FileReader` and `FileWriter` classes are node streams that are the Unicode character analogues of the `InputStream` and `OutputStream` classes.

## The BufferedReader and BufferedWriter Classes

Use the `BufferedReader` and `BufferedWriter` class filter character streams to increase the efficiency of I/O operations.

## The StringReader and StringWriter Classes

The `StringReader` and `StringWriter` classes are node character streams that *read* from or *write* to Java technology `String` objects.

Suppose that you wrote a set of report classes that contains methods that accept a `Writer` parameter (the destination of the report text). Because the method makes calls against a generic interface, the program can pass in a `FileWriter` object or a `StringWriter` object; the method code does not care. You use the former object to write the report to a file. You might use the latter object to write the report into memory within a `String` to be displayed within a GUI text area. In either case, the report writing code remains the same.

## The PipedReader and PipedWriter Classes

You use piped streams for communicating between threads. A `PipedReader` object in one thread receives its input from a complementary `PipedWriter` object in another thread. The piped streams must have both an input side and an output side to be useful.



## Module 11

---

# Console I/O and File I/O

---

## Objectives

Upon completion of this module, you should be able to:

- Read data from the console
- Write data to the console
- Describe files and file I/O

---

## Additional Resources



**Additional resources** – The following references provide additional information on the topics described in this module:

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.
- Bates, Sierra. *Head First Java, First Edition*. O'Reilly Media. 2003.

## Console I/O

Most applications must interact with the user. Such interaction is sometimes accomplished with text input and output to the console (using the keyboard as the standard input and using the terminal window as the standard output).

Java JDK supports console I/O with three public static variables on the `java.lang.System` class:

- The variable `System.out` is a `PrintStream` object that refers (initially) to the terminal window that launched the Java technology application.
- The variable `System.in` is an `InputStream` object that refers (initially) to the user's keyboard.
- The variable `System.err` is a `PrintStream` object that refers (initially) to the terminal window that launched the Java technology application.

It is possible to reroute these streams using the static methods: `System.setOut`, `System.setIn`, and `System.setErr`. For example, you could reroute standard error to a file stream.

## Writing to Standard Output

You can write to standard output through the `System.out.println(String)` method. This `PrintStream` method prints the string argument to the console and adds a newline character at the end. The following methods are also supported to print other types: primitives, a character buffer, and an arbitrary object. All of these methods add a newline character at the end of the output.

```
void println(boolean)
void println(char)
void println(double)
void println(float)
void println(int)
void println(long)
void println(char[])
void println(Object)
```

There is also a corresponding set of overloaded methods, called `print`, that do not add the newline character.

## Reading From Standard Input

The example Code 11-1, shows a technique that you should use to read String information from the console standard input:

### Code 11-1 KeyboardInput Program

```

1 import java.io.*;
2
3 public class KeyboardInput {
4 public static void main (String[] args) {
5 String s;
6 // Create a buffered reader to read
7 // each line from the keyboard.
8 InputStreamReader ir
9 = new InputStreamReader(System.in);
10 BufferedReader in = new BufferedReader(ir);
11 System.out.println("Unix: Type ctrl-d to exit." +
12 "\nWindows: Type ctrl-z to exit");
13

```

Line 5 declares a String variable, `s`, that the program uses to hold each line read from standard input. Lines 8–10 wrap `System.in` with two support objects that massage the stream of bytes coming from standard input. The `InputStreamReader` (`ir`) reads characters and converts the raw bytes into Unicode characters. The `BufferedReader` (`in`) provides the `readLine` method which enables the program to read from standard input one line at a time.

**Note** – The Control-d character on the UNIX platform indicates the *end of file* condition. In the Microsoft Windows environment, use the keystroke sequence Control-z followed by pressing the Enter key to indicate end-of-file (EOF).

The code for `KeyboardInput` continues:

```

14 try {
15 // Read each input line and echo it
16 s = in.readLine();
17 while (s != null) {
18 System.out.println("Read: " + s);
19 s = in.readLine();
20 }

```



Line 16 reads the first line of text from standard input. The while loop (Lines 17 – 20) iteratively prints out the current line and reads the next line. This code could be rewritten more succinctly (but more cryptically) as:

```
while ((s = in.readLine()) != null) {
 System.out.println("Read: " + s);
}
```

Because the `readLine` method can throw an I/O exception, you must wrap all of this code in a try-catch block.

```
21
22 // Close the buffered reader.
23 in.close();
```

Line 23 closes the outer most input stream to release any system resources related to creating these stream objects.

```
24 } catch (IOException e) { // Catch any exceptions.
25 e.printStackTrace();
26 }
27 }
28 }
```

Finally, the program handles any I/O exceptions that might occur.

---

**Note** – The call to the `close` method on Line 23 should really be performed in a `finally` clause. This is not done in this code example for reasons of brevity.

---



## Simple Formatted Output

The Java programming language version 5 provided C language-style `printf` functionality. It provides standard formatted output from the program. This enables programmers to migrate from legacy code.

You can use the `printf` as normal C and C++ syntax.

```
System.out.printf ("%s %5d %f%n", name, id, salary);
```

This formatting functionality is also available in the `String` class in the `format` method. The same output can also be generated as follows.

```
String s = String.format ("%s %5d %f%n", name, id, salary);
System.out.print(s);
```

Table 11-1 shows a few common formatting codes.

**Table 11-1** Common Formatting Codes

| Code                                            | Description                                                                                          |
|-------------------------------------------------|------------------------------------------------------------------------------------------------------|
| <code>%s</code>                                 | Formats the argument as a string, usually by calling the <code>toString</code> method on the object. |
| <code>%d</code> <code>%o</code> <code>%x</code> | Formats an integer, as a decimal, octal, or hexadecimal value.                                       |
| <code>%f</code> <code>%g</code>                 | Formats a floating point number. The <code>%g</code> code uses scientific notation.                  |
| <code>%n</code>                                 | Inserts a newline character to the string or stream.                                                 |
| <code>%%</code>                                 | Inserts the <code>%</code> character to the string or stream.                                        |

You can use `%n` for the newline character, instead of `\n`, for platform independence.

---

**Note** – For more information about the functionality of `printf` and the formatting codes, read the API document for the class `java.util.Formatter`.





## Simple Formatted Input

The Scanner class provides formatted input functionality. It is part of the `java.util` package. It provides methods for getting primitive values and strings and blocks as it waits for input from the user. An example is shown in Code 11-2.

### Code 11-2 Example of the Scanner Class

```
1 import java.io.*;
2 import java.util.Scanner;
3 public class ScanTest {
4 public static void main(String [] args) {
5 Scanner s = new Scanner(System.in);
6 String param = s.next();
7 System.out.println("the param 1" + param);
8 int value = s.nextInt();
9 System.out.println("second param" + value);
10 s.close();
11 }
12 }
```

In Code 11-2, Line 5 creates a Scanner reference by passing directly the console input. Line 6 retrieves the string value from the given input by using the Scanner method `next`. Line 8 gets the integer value of the given input by using the `nextInt` method. Line 10 closes the input from the user.

**Note** – Scanner class can be used to break down the formatted input into different tokens of primitive and String types. You can also use regular expressions to scan streams. For more information, look at the following link:

<http://java.sun.com/docs/books/tutorial/essential/io/scanning.html>

---

## Files and File I/O

Java technology includes a rich set of I/O streams which are described in the previous chapter. This section examines several simple techniques for reading and writing to files with a focus on character data, including:

- Creating `File` objects
- Manipulating `File` objects
- Reading and writing to file streams

### Creating a New `File` Object

The `File` class provides several utilities for handling files and obtaining information about them. In Java technology, a directory is just another file. You can create a `File` object that represents a directory and then use it to identify other files, as shown in the third bullet.

- ```
File myFile;
myFile = new File("myfile.txt");
```
- ```
myFile = new File("MyDocs", "myfile.txt");
```
- ```
File myDir = new File("MyDocs");
myFile = new File(myDir, "myfile.txt");
```

The constructor that you use often depends on the other file objects that you can access. For example, if you use only one file in your application, use the first constructor. However, if you use several files from a common directory, using the second or third constructors might be easier.

The class `File` defines platform-independent methods for manipulating a file maintained by a native file system. However, it does not permit you to access the contents of the file.

Note – You can use a `File` object as the constructor argument for `FileReader` and `FileWriter` objects in place of a string. This gives you independence from the local file system conventions and is recommended, in general.



The File Tests and Utilities

After you create a `File` object, you can use any methods in the following sections to gather information about the file.

File Names

The following methods return file names:

- `String getName()`
- `String getPath()`
- `String getAbsolutePath()`
- `String getParent()`
- `boolean renameTo(File newName)`

Directory Utilities

The following methods provide directory utilities:

- `boolean mkdir()`
- `String[] list()`

General File Information and Utilities

The following methods return general file information:

- `long lastModified()`
- `long length()`
- `boolean delete()`

File Tests

The following methods return information about file attributes:

- `boolean exists()`
- `boolean canWrite()`
- `boolean canRead()`
- `boolean isFile()`
- `boolean isDirectory()`

Files and File I/O

- boolean isAbsolute()
- boolean isHidden()

File Stream I/O

The Java SE Development Kit supports file input in two forms:

- Use the `FileReader` class to read characters
- Use the `BufferedReader` class to use the `readLine` method

The Java SE Development Kit supports file output in two forms:

- Use the `FileWriter` class to write characters
- Use the `PrintWriter` class to use the `print` and `println` methods

File Input Example

Code 11-3 reads a text file and echoes each line to standard output, which prints the file.

Code 11-3 Reading From a File

```
1 import java.io.*;
2 public class ReadFile {
3     public static void main (String [] args) {
4         // Create file
5         File file = new File(args[0]);
6
7         try {
8             // Create a buffered reader
9             // to read each line from a file.
10            BufferedReader in
11                = new BufferedReader(new FileReader(file));
12            String s;
13        }
```

Line 5 creates a new `File` object based on the first command-line argument to the program. Lines 10 and 11 creates a buffered reader that wraps around a file reader. This code throws a `FileNotFoundException` if the file does not exist.

Code 11-4 reads a text file and echoes each line to standard output, thus printing the file.

Code 11-4 Printing a File

```
14         try {
15             // Read each line from the file
16             s = in.readLine();
17             while (s != null) {
18                 System.out.println("Read: " + s);
19                 s = in.readLine();
20             }
21         } finally {
22             // Close the buffered reader
23             in.close();
24         }
25
26     } catch (FileNotFoundException e1) {
27         // If this file does not exist
28         System.err.println("File not found: " + file);
29
30     } catch (IOException e2) {
31         // Catch any other IO exceptions.
32         e2.printStackTrace();
33     }
34
35 }
36 }
```

In Code 11-4, the `while` loop in Lines 17–20 is exactly the same as in the `KeyboardInput` program (Code 11-1 on page 11-4); it reads each text line in the buffered reader and echoes it to standard output.

Line 23 closes the buffered reader, which in turn closes the file reader that the buffered reader object decorates.

The exception handling code in Lines 26–28 catches the `FileNotFoundException` that might be thrown by the `FileReader` constructor. Lines 30–33 handle any other I/O-based exception that might be thrown (by the `readLine` and `close` methods).

File Output Example

Code 11-5 reads input lines from the keyboard and echoes each line to a file.

Code 11-5 File Output Example

```

1 import java.io.*;
2
3 public class WriteFile {
4     public static void main (String[] args) {
5         // Create file
6         File file = new File(args[0]);
7
8         try {
9             // Create a buffered reader
10            InputStreamReader isr
11                = new InputStreamReader(System.in);
12            BufferedReader in
13                = new BufferedReader(isr);
14            // Create a print writer on this file.
15            PrintWriter out
16                = new PrintWriter(new FileWriter(file));
17            String s;

```

Just as in Code 11-4 on page 11-11, Line 6 of Code 11-5 creates a `File` object based on the first command-line argument. Lines 10–11 create a character reader stream from the binary stream (`System.in`). Lines 12–13 create a buffered reader for the standard input. Lines 15–16 create a print writer that decorates a file writer for the file created in Line 6.

```

18         System.out.print("Enter file text.  ");
19         System.out.println("[Type ctrl-d to stop.]");
20
21         // Read each input line
22         while ((s = in.readLine()) != null) {
23             out.println(s);
24         }
25

```

Lines 18–19 prompt the user to enter lines of text to be placed in the file and to type Control-d to stop. Lines 22–24 read from the input stream and prints to the file, one line at a time.



Note – The Control-d character (which represents *end-of-file*) must be used in this example and not Control-c, because Control-c terminates the JVM machine before the program closes the file stream properly.

```
26          // Close the buffered reader  
27          in.close();  
28          out.close();  
29  
30      } catch (IOException e) {  
31          // Catch any IO exceptions.  
32          e.printStackTrace();  
33      }  
34  }  
35 }
```

Lines 27 and 28 close the input and output streams. Lines 30–33 handle any I/O exceptions that might be thrown.

Module 12

Building Java GUIs Using the Swing API

Objectives

Upon completion of this module, you should be able to:

- Describe the JFC Swing technology
- Define Swing
- Identify the Swing packages
- Describe the GUI building blocks: containers, components, and layout managers
- Examine top-level, general-purpose, and special-purpose properties of container
- Examine components
- Examine layout managers
- Describe the Swing single-threaded model
- Build a GUI using Swing components

Additional Resources



Additional resources – The following references provide additional information on the topics described in this module:

Using all the functionality of JFC/Swing technology requires a lot more practice and study. Some references for further study of JFC/Swing technology are:

- JFC/Swing technology tutorial in the free Java tutorials, located at the following URL:
<http://java.sun.com/docs/books/tutorial/>
- The code of the SwingSet demo
- The API documentation
Start by studying the `javax.swing` package, and let that study lead you as necessary to the sub-packages.
- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.
- Bates, Sierra. *Head First Java, Second Edition*. O'Reilly Media. 2005.

What Are the Java Foundation Classes (JFC)?

JFC, or Java Foundation Classes, is a set of Graphical User Interface (GUI) support packages that are available as part of the Java SE platform and which became core application program interfaces (APIs) in JDK 1.2. As a result of this, some of the boundaries between JFC and core JDK have become blurred, but for the purposes of this course, JFC includes the following features:

- The Swing component set – *Swing* is an enhanced component set that provides replacement components for those in the original AWT and a number of more advanced components.
- 2D graphics – Using the Java 2D API, you can perform advanced drawing, complex color manipulations, shape and transformation (rotate, shear, stretch, and so forth) handling, and treat text as shapes that can be manipulated. 2D graphics are not discussed in this course.
- Pluggable look-and-feel. This feature provides Swing components a choice of look-and-feel. The same program can be rendered in Microsoft Windows, Motif, and Metal look-and-feel formats.
- Accessibility – National governments are increasingly mandating that computer programs used in their departments are accessible to those with disabilities. The Swing component set facilitates such programming by means of the accessibility APIs. It provides interfaces for associative technologies, such as screen readers, screen magnifiers, audible text readers (speech processing), and so on.
- Drag-and-drop – GUI-based data transfer, both between elements of one program and between different programs, has been a feature of modern GUI systems for a number of years. This type of transfer falls into two forms from the user's point of view. These are cut-and-paste and drag-and-drop. The Java JDK releases include support for both these communications media.
- Internationalization – The JFC classes support different character sets such as Japanese, Chinese, and Korean. This allows developers to build applications that can interact with users worldwide in their own languages.

Additional Resources

What Is Swing?

Swing is an enhanced component set that provides replacement components for those in the original AWT and a number of more advanced components. These components enable you to create user interfaces with the type of functionality that has become expected in modern applications. Such components include trees, tables, advanced text editors, and tear-off toolbars.

Swing also has special features. For example, using Swing, you can write a program that adopts either the *look-and-feel* of the host platform or that uses a common look-and-feel written especially for the Java programming language (Metal). In fact, you can create your own look-and-feel from scratch, or modify an existing one and *plug it in* to your program, either hard-coded or by the user or system administrator selecting a look-and-feel at runtime.

Note – The term look-and-feel occurs frequently in this module. Look refers to appearance of components, while feel refers to the way they react to user actions, such as mouse clicks. Writing a look-and-feel refers to writing the necessary classes to define new appearance and input behavior. Writing a look-and-feel is outside the scope of this module.



Pluggable Look-and-Feel

Pluggable look-and-feel enables developers to build applications that execute on any platform as if they were developed for that specific platform. A program executed in the Microsoft Windows environment appears as if it was developed for this environment; and the same program executed on the UNIX platform appears as if it was developed for the UNIX environment.

Developers can create their own custom Swing components, with any kind of look-and-feel that they choose to design. This increases the consistency of applications and applets deployed across platforms. An entire application's GUI can switch from one look-and-feel to a different one at runtime.

Pluggable look-and-feel provided by the Swing components is facilitated by their underlying architecture. The next section describes the Swing architecture and explains how it facilitates the pluggable look-and-feel.

Swing Architecture

Swing components are designed based on the Model-View-Controller (MVC) architecture. The Swing architecture is not strictly based on the MVC architecture but has its roots in the MVC.

Model-View-Controller Architecture

According to the MVC architecture, a component can be modeled as three separate parts. Figure 12-1 shows the MVC architecture.

- Model – The model stores the data used to define the component.
- View – The view represents the visual display of the component. This display is governed by the data in the model.
- Controller – The controller deals with the behavior of the components when a user interacts with it. This behavior can include any updates to the model or view.

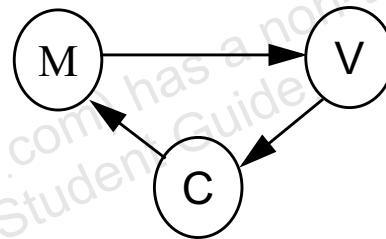


Figure 12-1 Model-View-Controller Architecture

Theoretically, these three types of architecture (Model, View, Controller) should be represented by different class types. But, practically, this turns out to be very difficult because of the dependencies between the view and the controller. The role of the controller is heavily dependent on the implementation of the view, because the user interacts with the view. In other words, it is difficult to write a generic controller, independent of the implementation of the view. This issue is addressed by the separable model architecture.

Separable Model Architecture

The Swing components follow a separable model architecture. In this architecture the view and the controller are merged as a single composite object, because of their tight dependency on each other. The model object is treated as a separate object just like in MVC architecture. Figure 12-2 on page 12-6 shows the separable model architecture.

Additional Resources

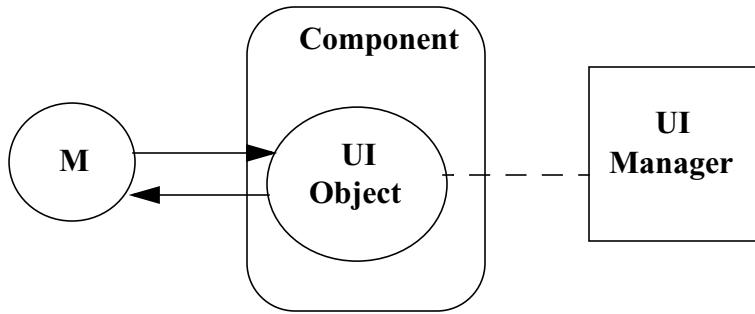


Figure 12-2 Separable Model Architecture

The UI Object in Figure 12-2 is referred to as UI delegate. With this architecture it is possible to delegate a few responsibilities of the component to an independent UI Object. This explains how the pluggable look-and-feel is facilitated by the Swing components. The components make the visual appearance of the components the responsibility of the independent UI Object. So the UI Object of the installed look-and-feel is responsible for the components look-and-feel.

Swing Packages

The Swing API has a rich and convenient set of packages that makes it powerful and flexible. Table 12-1 lists each package name and the purpose of each package.

Table 12-1 Swing Packages

Package Name	Purpose
javax.swing	Provides a set of <i>light-weight</i> components such as, JButton, JFrame, JCheckBox, and much more
javax.swing.border	Provides classes and interfaces for drawing specialized borders such as, bevel, etched, line, matte, and more
javax.swing.event	Provides support for events fired by Swing components
javax.swing.undo	Allows developers to provide support for undo/redo in applications such as text editors
javax.swing.colorchooser	Contains classes and interfaces used by the JColorChooser component
javax.swing.filechooser	Contains classes and interfaces used by the JFileChooser component
javax.swing.table	Provides classes and interfaces for handling JTable
javax.swing.tree	Provides classes and interfaces for handling JTree
javax.swing.plaf	Provides one interface and many abstract classes that Swing uses to provide its pluggable look-and-feel capabilities
javax.swing.plaf.basic	Provides user interface objects built according to the Basic look-and-feel

Swing Packages

Table 12-1 Swing Packages (Continued)

Package Name	Purpose
javax.swing.plaf.metal	Provides user interface objects built according to the Java look-and-feel
javax.swing.plaf.multi	Provides user interface objects that combine two or more look-and-feels
javax.swing.plaf.synth	Provides user interface objects for a skinnable look-and-feel in which all painting is delegated
javax.swing.text	Provides classes and interfaces that deal with editable and non-editable text components
javax.swing.text.html	Provides the class HTMLEditorKit and supporting classes for creating HTML text editors
javax.swing.text.html.parser	Provides the default HTML parser, along with support classes

Examining the Composition of a Java Technology GUI

A Swing API-based GUI is composed of the following elements.

- Containers

Containers are on top of the GUI containment hierarchy. All the components in the GUI are added to these containers. `JFrame`, `JDialog`, `JWindow`, and `JApplet` are the top-level containers.

- Components

All the GUI components are derived from the `JComponent` class, for example, `JComboBox`, `JAbstractButton`, and `JTextComponent`.

- Layout Managers

Layout managers are responsible for laying out components in a container. `BorderLayout`, `FlowLayout`, `GridLayout` are a few examples of the layout managers. There are more sophisticated and complex layout managers that give more control over the GUI.

Examining the Composition of a Java Technology GUI

Figure 12-3 shows the use of components, containers, and layout managers in the composition of the sample Swing user interface.

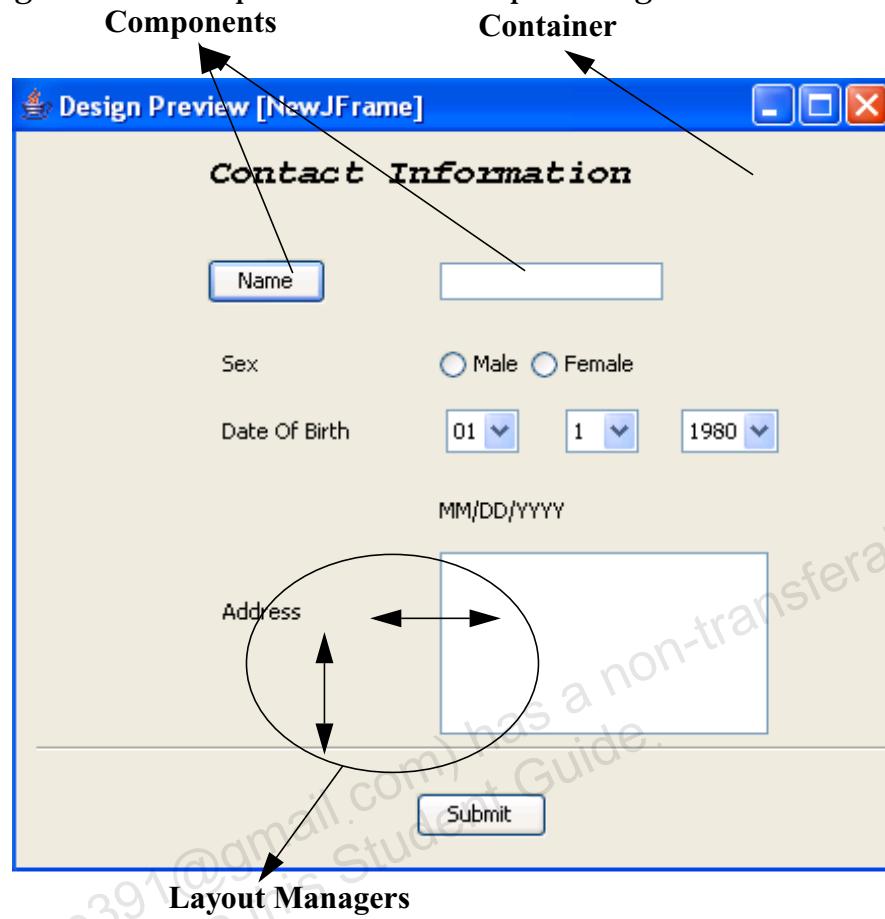


Figure 12-3 GUI Composition

Swing Containers

Swing containers can be classified into three main categories

- Top-level containers
- General-purpose containers
- Special-purpose containers

Top-level Containers

Top-level containers are at the top of the Swing containment hierarchy. There are three top-level Swing containers: `JFrame`, `JWindow`, and `JDialog`. There is also a special class, `JApplet`, which, while not strictly a top-level container, is worth mentioning here because it should be used as the top-level of any applet that uses Swing components. Figure 12-4 shows the inheritance hierarchy of the top-level containers.

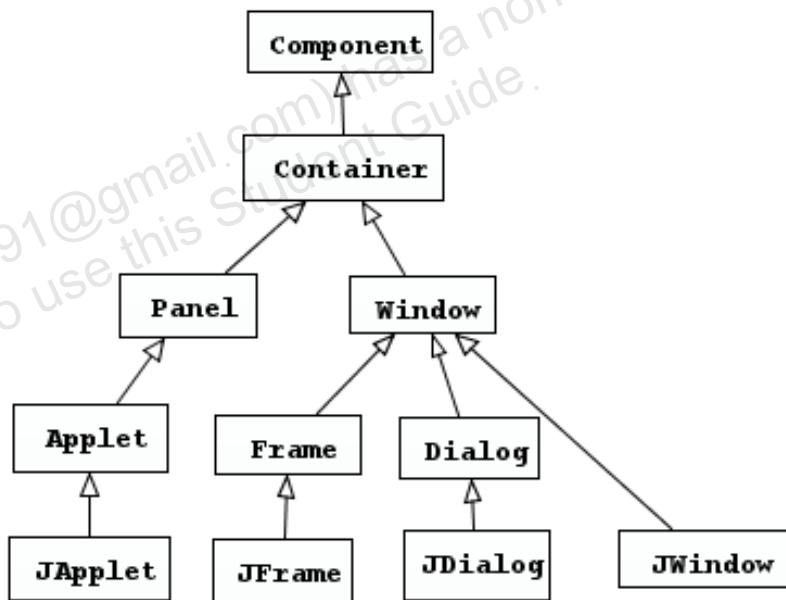


Figure 12-4 Top-Level Container Hierarchy

`JApplet`, `JFrame`, `JDialog`, and `JWindow` classes are directly derived from `Applet`, `Frame`, `Dialog`, and `Window` classes, respectively. It is important to note this because all the other Swing containers and components are derived from `JComponents`.

Swing Components

Swing GUIs use two kinds of classes: GUI classes and non-GUI support classes. The GUI classes are visual and descendants of `JComponent`, and are called `J` classes. The non-GUI classes provide services and perform vital functions for GUI classes; however, they do not produce any visual output.

Swing components primarily provide components for text handling, buttons, labels, lists, panes, combo boxes, scroll bars, scroll panes, menus, tables, and trees. Swing components can be broadly classified as follows:

- Buttons
- Text components
- Uneditable information display components
- Menus
- Formatted display components
- Other basic controls

The Swing Component Hierarchy

Figure 12-5 illustrates the hierarchy relationships of the swing components.

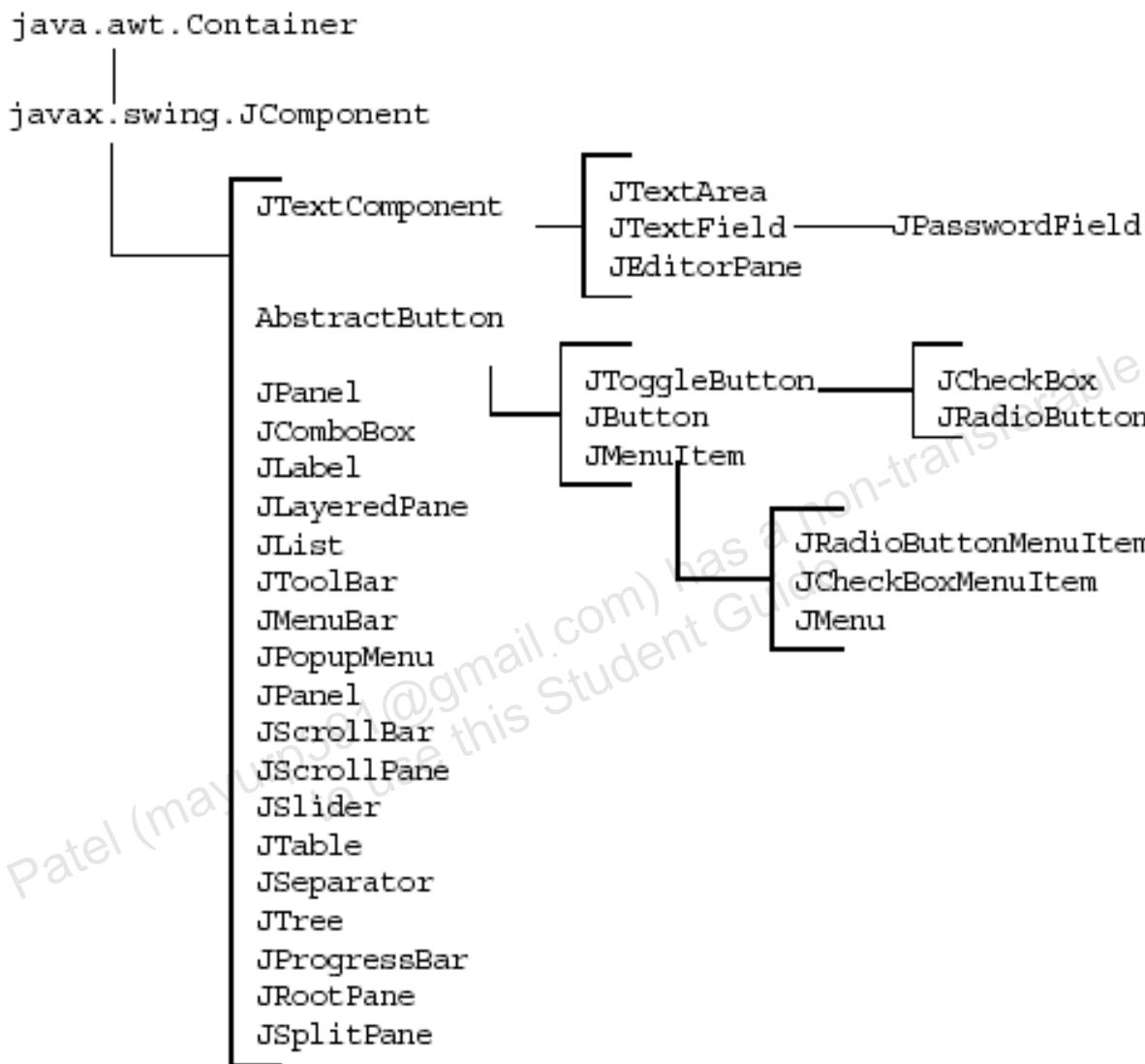


Figure 12-5 Swing Component Hierarchy

Note – The Swing components' event handling classes are examples of non-GUI classes.

Properties of Swing Components

This section describes the properties of Swing components.

Common Component Properties

All the Swing components share some common properties because they all extend the `JComponent` class. Table 12-2 shows a subset of the common properties that all Swing components inherit from `JComponent`.

Table 12-2 Common Component Properties

Property	Methods
Border	<code>Border getBorder()</code> <code>void setBorder(Border b)</code>
Background and foreground color	<code>void setBackground(Color bg)</code> <code>void setForeground(Color bg)</code>
Font	<code>voidsetFont(Font f)</code>
Opaque	<code>void setOpaque(boolean isOpaque)</code>
Maximum and minimum size	<code>void setMaximumSize(Dimension d)</code> <code>void setMinimumSize(Dimension d)</code>
Alignment	<code>void setAlignmentX(float ax)</code> <code>void setAlignmentY(float ay)</code>
Preferred size	<code>void setPreferredSize(Dimension ps)</code>

Note – Some properties, such as preferred size, are hints to the layout manager. Although a component can provide the layout manager with hints, the layout manager can ignore these hints and use other information to rendering the component.



Component-Specific Properties

This section discusses component properties specifically taking `JComboBox` as an example. `JComboBox` inherits all the properties in `JComponent` and defines more specific properties. Table 12-3 describes some properties specific to `JComboBox`.

Table 12-3 Component Specific Properties

Properties	Methods
Maximum row count	<code>void setMaximumRowCount(int count)</code>
Model	<code>void setModel(ComboBoxModel cbm)</code>
Selected index	<code>int getSelectedIndex()</code>
Selected Item	<code>Object getSelectedItem()</code>
Item count	<code>int getItemCount()</code>
Renderer	<code>void setRenderer(ListCellRenderer ar)</code>
Editable	<code>void setEditable(boolean flag)</code>

Layout Managers

A layout manager determines the size and position of the components within a container. The alternative to using layout managers is absolute positioning by pixel coordinates. Absolute positioning is achieved through setting a container's layout property to null. Absolute positioning is not platform-portable. Issues such as the sizes of fonts and screens ensure that a layout that is correct based on coordinates can potentially be unusable on other platforms.

Unlike absolute positioning, layout managers have mechanisms to cope with the following situations:

- The resizing of the GUI by the user
- Different fonts and font sizes used by different operating systems or by user customization
- The text layout requirements of different international locales (left-right, right-left, vertical)

To cope with these situations, layout managers lay out components according to a predetermined policy. For example, the policy of the GridLayout is to position child components in equal-sized cells, starting at the top left and working left to right, top to bottom until the grid is full. The following sections describe some of the layout managers that you can use. Each section highlights the policy used by the layout manager under discussion.

The BorderLayout Layout Manager

BorderLayout arranges the components in five different regions: CENTER, NORTH, SOUTH, EAST, and WEST. The border layout manager limits the number of components added to each region to one.

The position of the component should be specified. If no position is specified, by default the component is added to the CENTER. All the extra space left is used by the component in the CENTER.

BorderLayout is the default layout for JFrame, JDialog, and JApplet. Figure 12-6 on page 12-17 shows a display using a border layout. The display shows five JButtons added to a JFrame.



Figure 12-6 The BorderLayout Example

The code below shows the BorderLayout example. It adds five buttons to the JFrame.

Code 12-1 BorderLayout Example

```
1 import java.awt.*;
2 import javax.swing.*;
3
4 public class BorderExample {
5     private JFrame f;
6     private JButton bn, bs, bw, be, bc;
7
8     public BorderExample() {
9         f = new JFrame("Border Layout");
10        bn = new JButton("Button 1");
11        bc = new JButton("Button 2");
12        bw = new JButton("Button 3");
13        bs = new JButton("Button 4");
14        be = new JButton("Button 5");
15    }
16
17    public void launchFrame() {
18        f.add(bn, BorderLayout.NORTH);
19        f.add(bs, BorderLayout.SOUTH);
20        f.add(bw, BorderLayout.WEST);
21        f.add(be, BorderLayout.EAST);
22        f.add(bc, BorderLayout.CENTER);
23        f.setSize(400,200);
24        f.setVisible(true);
25    }
26
27    public static void main(String args[]) {
```

Layout Managers

```

28     BorderExample guiWindow2 = new BorderExample();
29     guiWindow2.launchFrame();
30 }
31 }
```

The FlowLayout Layout Manager

FlowLayout arranges the components in a row. By default, it arranges the components from LEFT_TO_RIGHT. This orientation can be changed using the ComponentOrientation property RIGHT_TO_LEFT. The vertical and horizontal spacing between the components can be specified. If not, the default vertical and horizontal gap of five units is used. Figure 12-7 shows the flow layout example. Similar to the border layout example, five JButtons are added to the JFrame.



Figure 12-7 The FlowLayout Example

The following code shows an example of FlowLayout. It adds five buttons to the JFrame.

Code 12-2 FlowLayout Example

```

1 import java.awt.*;
2 import javax.swing.*;
3
4 public class FlowExample {
5     private JFrame f;
6     private JButton b1;
7     private JButton b2;
8     private JButton b3;
9     private JButton b4;
10    private JButton b5;
11
12    public FlowExample() {
13        f = new JFrame("GUI example");
14        b1 = new JButton("Button 1");
15        b2 = new JButton("Button 2");
16        b3 = new JButton("Button 3");
17        b4 = new JButton("Button 4");
18        b5 = new JButton("Button 5");
```

```
19     }
20
21     public void launchFrame() {
22         f.setLayout(new FlowLayout());
23         f.add(b1);
24         f.add(b2);
25         f.add(b3);
26         f.add(b4);
27         f.add(b5);
28         f.pack();
29         f.setVisible(true);
30     }
31
32     public static void main(String args[]) {
33         FlowExample guiWindow = new FlowExample();
34         guiWindow.launchFrame();
35     }
36
37 } // end of FlowExample class
```

The BoxLayout Layout Manager

BoxLayout arranges the components either vertically or horizontally.

The BoxLayout constructor takes a parameter called axis, where the direction to align the components should be specified. This parameter can take any of the following values

- X_AXIS – Components are arranged horizontally from left to right.
- Y_AXIS – Components are arranged vertically from top to bottom.
- LINE_AXIS – Components are laid in the same direction as words in a line.
- PAGE_AXIS – Components are arranged in the same direction as the lines in a page.

Layout Managers

Figure 12-8 shows a box layout example. Similar to the border layout example, five JButtons are added to the JFrame. The parameter used in this example for aligning the components was set to Y-AXIS.

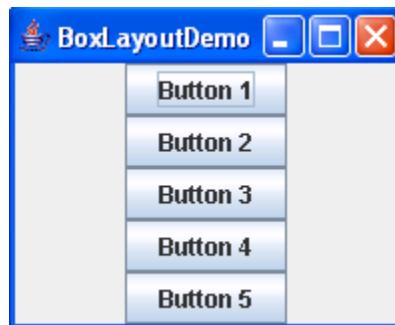


Figure 12-8 The BoxLayout Example

The CardLayout Layout Manager

CardLayout arranges the components as a stack of cards. Each card accepts a single component for display. By making this single component a container, you can display multiple components in a card. Only one card is visible at a time. Figure 12-9 illustrates the use of the card layout manager.

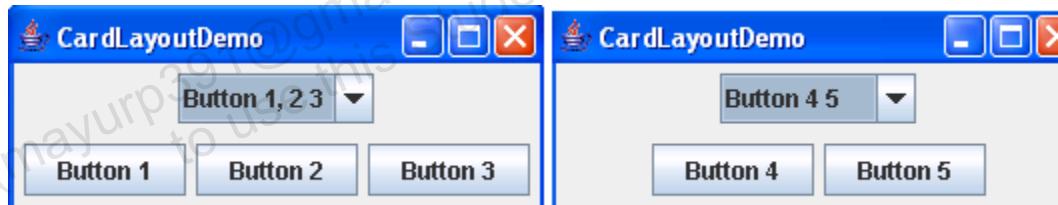


Figure 12-9 The CardLayout Example

In the example shown in Figure 12-9, Button1, Button2, and Button3 are arranged on Card1 and Button4 and Button5 are arranged on Card2. A combo box is used to select the card to be displayed.

The GridLayout Layout Manager

GridLayout arranges the components in rows and columns. Each component occupies the same amount of space in the container. When creating the grid layout, the number of rows and columns should be specified. If not specified, by default, the layout manager creates one row and one column. The vertical gap and the horizontal gap between the components can also be specified. Figure 12-10 illustrates the use of the GridLayout. Similar to the border layout shown in Figure 12-6 on page 12-17, five JButtons are added to the JFrame.

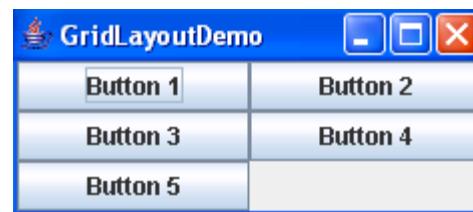


Figure 12-10 The GridLayout Example

The following code shows an example of GridLayout. It adds five buttons to the JFrame.

Code 12-3 GridLayout Example

```
1 import java.awt.*;
2 import javax.swing.*;
3
4 public class GridExample {
5     private JFrame f;
6     private JButton b1, b2, b3, b4, b5;
7
8     public GridExample() {
9         f = new JFrame("Grid Example");
10        b1 = new JButton("Button 1");
11        b2 = new JButton("Button 2");
12        b3 = new JButton("Button 3");
13        b4 = new JButton("Button 4");
14        b5 = new JButton("Button 5");
15    }
16
17    public void launchFrame() {
18        f.setLayout (new GridLayout(3,2));
19
20        f.add(b1);
21        f.add(b2);
```

Layout Managers

```

22     f.add(b3) ;
23     f.add(b4) ;
24     f.add(b5) ;
25
26     f.pack() ;
27     f.setVisible(true) ;
28 }
29
30 public static void main(String args[]) {
31     GridExample grid = new GridExample() ;
32     grid.launchFrame() ;
33 }
34 }
```

The GridBagLayout Layout Manager

GridBagLayout arranges the components in rows and columns, similar to grid layout, but provides a wide variety of flexibility options for resizing and positioning the components. This layout is used to design complex GUIs. The constraints on the components are specified using the GridBagConstraints class. Some of the constants in this class are gridwidth, gridheight,.gridx,.gridy, weightx, and weighty. Figure 12-11 illustrates the use of GridBagLayout. Five JButtons are added to the JFrame. You can notice that the components are of different size and are positioned at very specific locations.

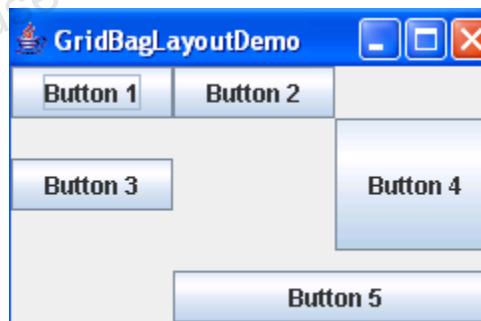


Figure 12-11 The GridBagLayout Example

The GroupLayout Layout Manager

In addition to these layout managers, GroupLayout was added to Java SE version 6. This layout manager was added for use by toolmakers. It is the layout manager used in the NetBeans IDE GUI builder tool. For more information see:

<http://java.sun.com/docs/books/tutorial/uiswing/layout/group.html>.

GUI Construction

A Java technology GUI can be created using either of the following techniques:

- Programmatic construction

This technique uses code to create the GUI. This technique is useful for learning GUI construction. However, it is very laborious for use in production environments.

- Construction using a GUI builder tool

This technique uses a GUI builder tool to create the GUI. The GUI developer uses a visual approach to drag-and-drop containers and components to a work area. The tool permits the positioning and resizing of containers and components using a pointer device such as a computer mouse. With each step, the tool automatically generates the Java technology classes required to reproduce the GUI.

Programmatic Construction

This section describes creating a simple GUI that prints Hello World. The code shown in Code 12-4 creates a container `JFrame` with a title `HelloWorld Swing`. It later adds a `JLabel` with the Accessible Name property set to Hello World.

Code 12-4 The HelloWorld Swing Application

```
1 import javax.swing.*;
2 public class HelloWorldSwing {
3     private static void createAndShowGUI() {
4         JFrame frame = new JFrame("HelloWorldSwing");
5         //Set up the window.
6         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7         JLabel label = new JLabel("Hello World");
8         // Add Label
9         frame.add(label);
10        frame.setSize(300,200);
11        // Display Window
12        frame.setVisible(true);}
13
14    public static void main(String[] args) {
15        javax.swing.SwingUtilities.invokeLater(new Runnable() {
16            //Schedule for the event-dispatching thread:
17            //creating, showing this app's GUI.
18    }
19}
```

```

18     public void run() {createAndShowGUI(); }
19 }
20 }
21 }
```

Figure 12-12 shows the GUI interface that is generated by the code. The default layout for JFrame is BorderLayout. So, by default, the component JLabel is added to the center position of the container. Also note that the label occupies the entire frame, as the center component occupies all the remaining space in the container in the BorderLayout.

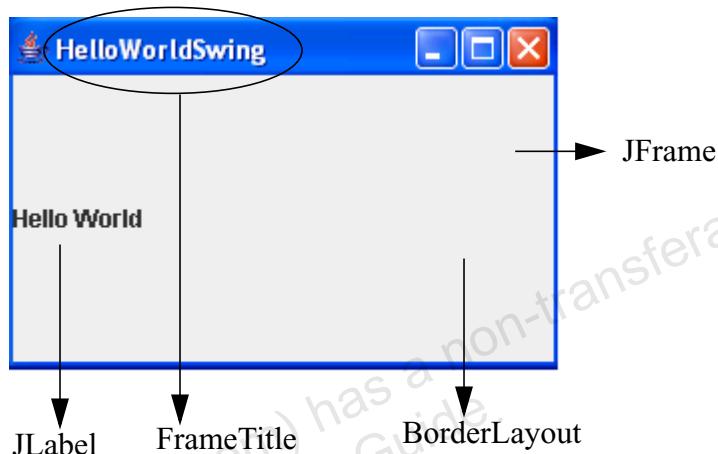


Figure 12-12 The HelloWorldString Output

Key Methods

This section explains the key methods used in Code 12-4 on page 12-24. The methods can be divided into two different categories.

1. Methods for setting up the frame and adding a label.
 - a. `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`: This method defines the behavior of the JFrame when a close operation is initiated. There are four possible ways of handling this.
 1. `DO NOTHING ON CLOSE`: Does nothing when the close operation is initiated. This constant is defined in `WindowsConstants`.
 2. `HIDE ON CLOSE`: Invokes any `WindowListener` objects and hides the frame. This constant is defined in `WindowConstants`.

GUI Construction

- Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.
3. **DISPOSE_ON_CLOSE:** Invokes any `WindowListener` objects and hides and disposes the frame. This constant is defined in `WindowConstants`.
 4. **EXIT_ON_CLOSE:** Exits the application. This constant is defined in the `JFrame` class.
- b. `setVisible(true)`: When the `JFrame` is first created, it creates an invisible frame. To make the frame visible, the parameter for the `setVisible` should be set to true. `JFrame` inherits this method from `java.awt.Component`.
 - c. `add(Component c)`: This method adds the components to the container. `JFrame` inherits this method from `java.awt.Component` class. Five different overloaded methods are defined in the `java.awt.Component`.
2. Methods for making the GUI thread-safe and efficient.
Several tasks are involved in displaying the GUI efficiently. These tasks can be broadly defined as:
 - a. Executing the application code.
This task involves starting the GUI application and executing the code for rendering the GUI.
 - b. Handling the events raised from the GUI:
Several events can be raised by the components in the GUI. For example, when a button is pressed an event is generated. Event listeners should be defined to handle this event. This task dispatches the event to the appropriate listeners that handle the event.

c. Handle some time-consuming processes:

Several activities might be time-consuming and they can be run in the background so that the GUI would be efficient and responsive. These kinds of activities are handled by this task.

To handle these tasks efficiently, the Swing framework uses threads that are light-weight processes. The tasks described above can be handled by these threads separately and concurrently. The programmer should utilize these threads. The Swing framework provides a collection of utility methods in the `SwingUtilities` class.

- `SwingUtilities.invokeLater(new Runnable())`:

In the Java programming language, threads are created using the `Runnable` interface. This interface defines a method `run` that should be implemented by all the classes using this interface. The `invokeLater` method schedules the GUI creation task to execute the `run` method asynchronously by the event-handling thread after all the pending events are completed.

Module 13

Handling GUI-Generated Events

Objectives

Upon completion of this module, you should be able to:

- Define events and event handling
- Examine the Java SE event model
- Describe GUI behavior
- Determine the user action that originated an event
- Develop event listeners
- Describe concurrency in Swing-based GUIs and describe the features of the `SwingWorker` class

Additional Resources



Additional resources – The following references provide additional information on the topics described in this module:

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.
- Bates, Sierra. *Head First Java, Second Edition*. O'Reilly Media. 2005.

What Is an Event?

When the user performs an action at the user interface level (clicks a mouse or presses a key), this causes an *event* to be issued. Events are objects that describe what has happened. A number of different types of event classes exist to describe different categories of user action.

Figure 13-1 shows an abstract view of the delegation event model. When a user clicks a GUI button, an event object is created by the JVM and the button sends this event to the event handler for that button by calling the `actionPerformed` method.

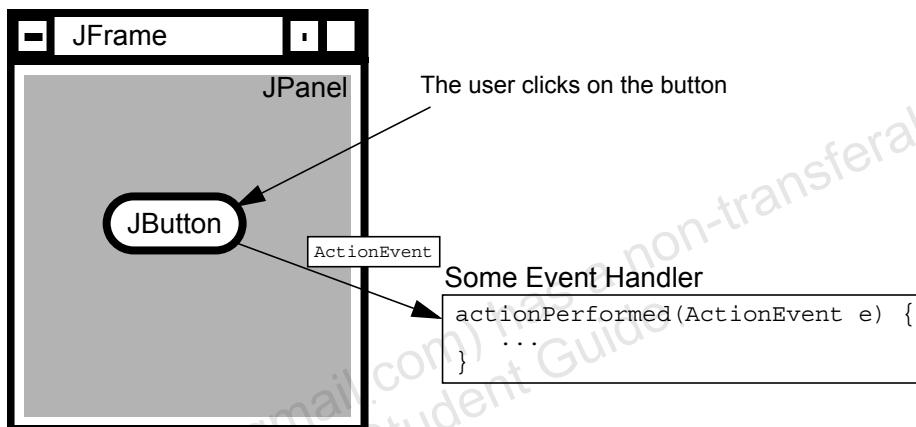


Figure 13-1 A User Action Triggers an Event

Event Sources

An *event source* is the generator of an event. For example, a mouse click on a JButton component generates an ActionEvent instance with the button as the source. The ActionEvent instance is an object that contains information about the events that just took place. The ActionEvent contains:

- `getActionCommand`, which returns the command name associated with the action
- `getModifiers`, which returns any modifiers held during the action
- `getWhen`, which returns the timestamp when the event occurred
- `paramString`, which returns a string identifying action and the associated command

Event Handlers

An *event handler* is a method that receives an event object, deciphers it, and processes the user's interaction.

Java SE Event Model

This section describes the delegation event model.

Delegation Model

The delegation event model came into existence with JDK version 1.1. With this model, events are sent to the component from which the event originated, but it is up to each component to propagate the event to one or more registered classes, called *listeners*. Listeners contain event handlers that receive and process the event. In this way, the event handler can be in an object separate from the component. Listeners are classes that implement the `EventListener` interface. Figure 13-2 shows the event delegation model.

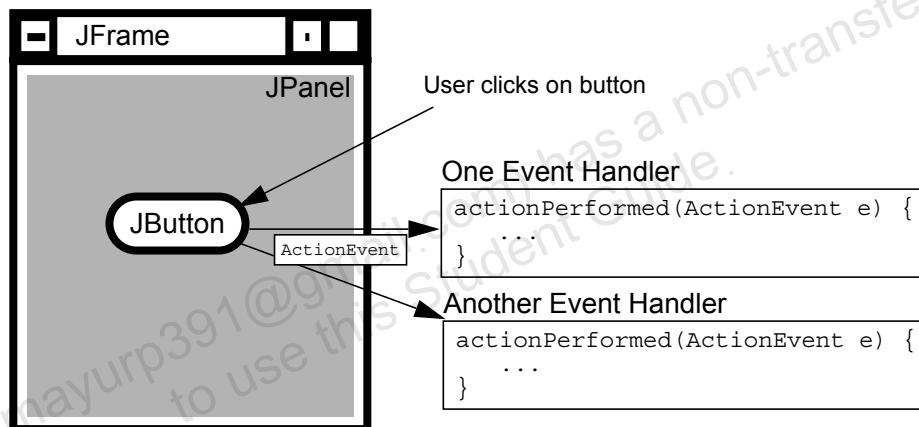


Figure 13-2 Delegation Event Model With Multiple Listeners

Events are objects that are reported only to registered listeners. Every event has a corresponding listener interface that mandates which methods must be defined in a class suited to receiving that type of event. The class that implements the interface defines those methods, and can be registered as a listener.

Events from components that have no registered listeners are not propagated.

A Listener Example

For example, Code 13-1 shows the code for a simple JFrame with a single JButton on it.

Code 13-1 The TestButton Example

```
1 import javax.swing.*;
2 import java.awt.*;
3
4 public class TestButton {
5     private JFrame f;
6     private JButton b;
7
8     public TestButton() {
9         f = new JFrame("Test");
10        b = new JButton("Press Me!");
11        b.setActionCommand("ButtonPressed");
12    }
13
14    public void launchFrame() {
15        b.addActionListener(new ButtonHandler());
16        f.add(b, BorderLayout.CENTER);
17        f.pack();
18        f.setVisible(true);
19    }
20
21    public static void main(String args[]) {
22        TestButton guiApp = new TestButton();
23        guiApp.launchFrame();
24    }
25 }
```

The ButtonHandler class, shown in Code 13-2, is the handler class to which the event is delegated.

Code 13-2 The ButtonHandler Example

```
1 import java.awt.event.*;
2
3 public class ButtonHandler implements ActionListener {
4     public void actionPerformed(ActionEvent e) {
5         System.out.println("Action occurred");
6         System.out.println("Button's command is: "
7                            + e.getActionCommand());
8     }
9 }
```

Java SE Event Model

Code 13-2 on page 13-5 has the following characteristics:

- The JButton class inherits the addActionListener(ActionListener) method from javax.swing.AbstractButton which is the superclass of JButton.
- The ActionListener interface defines a single method, actionPerformed, which receives an ActionEvent.
- After it is created, a JButton object can have an object registered as a listener for ActionEvents through the addActionListener() method. The registered listener is instantiated from a class that implements the ActionListener interface.
- When the JButton object is clicked, an ActionEvent is sent. The ActionEvent is received through the actionPerformed() method of any ActionListener that is registered on the button through its addActionListener() method.
- The method, getActionCommand(), of the ActionEvent class returns the command name associated with this action. On Line 11, the action command for this button is set to ButtonPressed.

Note – There are other ways to determine why an event has been received. In particular, the method getSource(), documented in the java.util.EventObject base class, is often useful because it enables you to obtain the reference to the object that sent the event.

Events are not handled accidentally. The objects that are scheduled to listen to particular events on a particular GUI component register themselves with that component.

- When an event occurs, only the objects that were registered receive a message that the event occurred.
- The delegation model is good for the distribution of work among objects.

Events do not have to be related to Swing components. This event model provides support for the JavaBeans architecture.



GUI Behavior

This section describes the event categories.

Event Categories

The general mechanism for receiving events from components has been described in the context of a single type of event. Many of the event classes reside in the `java.awt.event` package, but others exist elsewhere in the API. Figure 13-3 shows a UML class hierarchy of GUI event classes.

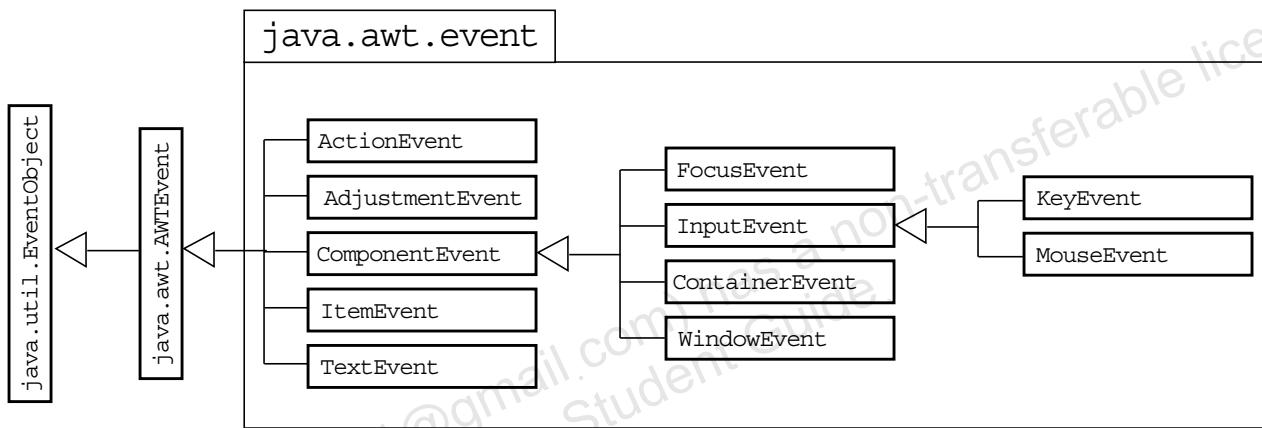


Figure 13-3 Class Hierarchy of GUI Events

For each category of events, an interface has to be implemented by the class of objects that wants to receive the events. That interface demands that one or more methods be defined as well. Those methods are called when particular events arise. Table 13-1 lists these categories and interfaces.

Table 13-1 Method Categories and Interfaces

Category	Interface Name	Methods
Action	ActionListener	actionPerformed(ActionEvent)
Item	ItemListener	itemStateChanged(ItemEvent)

GUI Behavior

Table 13-1 Method Categories and Interfaces (Continued)

Category	Interface Name	Methods
Mouse	MouseListener	mousePressed(MouseEvent) mouseReleased(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mouseClicked(MouseEvent)
Mouse motion	MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
Key	KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
Focus	FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)
Adjustment	AdjustmentListener	adjustmentValueChanged (AdjustmentEvent)
Component	ComponentListener	componentMoved(ComponentEvent) componentHidden(ComponentEvent) componentResized(ComponentEvent) componentShown(ComponentEvent)
Window	WindowListener	windowClosing(WindowEvent) windowOpened(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent)
Container	ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
Window state	WindowStateListener	windowStateChanged(WindowEvent e)
Window focus	WindowFocusListener	windowGainedFocus(WindowEvent e) windowLostFocus(WindowEvent e)
Mouse wheel	MouseWheelListener	mouseWheelMoved(MouseWheelEvent e)

Table 13-1 Method Categories and Interfaces (Continued)

Category	Interface Name	Methods
Input methods	InputMethodListener	caretPositionChanged (InputMethodEvent e) inputMethodTextChnaged (InputMethodEvent e)
Hierarchy	HierarchyListener	hierarchyChanged (HierarchyEvent e)
Hierarchy bounds	HierarchyBoundsListe ner	ancestorMoved (HierarchyEvent e) ancestorResized (HierarchyEvent e)
AWT	AWTEventListener	eventDispatched (AWTEvent e)
Text	TextListener	textValueChanged (TextEvent)

Complex Example

This section examines a more complex Java code software example. It tracks the movement of the mouse when the mouse button is pressed (*mouse dragging*). It also detects mouse movement even when the buttons are not pressed (*mouse moving*).

The events caused by moving the mouse with or without a button pressed can be picked up by objects of a class that implements the `MouseMotionListener` interface. This interface requires two methods, `mouseDragged()` and `mouseMoved()`. Even if you are interested only in the drag movement, you must provide both methods. However, the body of the `mouseMoved()` method can be empty.

To pick up other mouse events, including mouse clicking, you must implement the `MouseListener` interface. This interface includes several events, including `mouseEntered`, `mouseExited`, `mousePressed`, `mouseReleased`, and `mouseClicked`.

When mouse or keyboard events occur, information about the position of the mouse or the key that was pressed is available in the event that it generated. In Code 13-2 on page 13-5 on event handling, there was a separate class named `ButtonHandler` that handled events. In Code 13-3, events are handled within the class named `TwoListener`.

Code 13-3 The TwoListener Example

```
1 import javax.swing.*;
```

GUI Behavior

```
2 import java.awt.event.*;
3 import java.awt.*;
4
5 public class TwoListener
6     implements MouseMotionListener, MouseListener {
7     private JFrame f;
8     private JTextField tf;
9
10    public TwoListener() {
11        f = new JFrame("Two listeners example");
12        tf = new JTextField(30);
13    }
14
15    public void launchFrame() {
16        JLabel label = new JLabel("Click and drag the mouse");
17        // Add components to the frame
18        f.add(label, BorderLayout.NORTH);
19        f.add(tf, BorderLayout.SOUTH);
20        // Add this object as a listener
21        f.addMouseMotionListener(this);
22        f.addMouseListener(this);
23        // Size the frame and make it visible
24        f.setSize(300, 200);
25        f.setVisible(true);
26    }
27
28    // These are MouseMotionListener events
29    public void mouseDragged(MouseEvent e) {
30        String s = "Mouse dragging: X = " + e.getX()
31                    + " Y = " + e.getY();
32        tf.setText(s);
33    }
34
35    public void mouseEntered(MouseEvent e) {
36        String s = "The mouse entered";
37        tf.setText(s);
38    }
39
40    public void mouseExited(MouseEvent e) {
41        String s = "The mouse has left the building";
42        tf.setText(s);
43    }
44
45    // Unused MouseMotionListener method.
46    // All methods of a listener must be present in the
47    // class even if they are not used.
```

```

48     public void mouseMoved(MouseEvent e) { }
49
50     // Unused MouseListener methods.
51     public void mousePressed(MouseEvent e) { }
52     public void mouseClicked(MouseEvent e) { }
53     public void mouseReleased(MouseEvent e) { }
54
55     public static void main(String args[]) {
56         TwoListener two = new TwoListener();
57         two.launchFrame();
58     }
59 }
```

A number of points shown in Code 13-3 on page 13-9 are described in the following paragraphs.

Implementing Multiple Interfaces

The class is declared in Lines 5 and 6 using the following:

```
implements MouseMotionListener, MouseListener
```

You can declare multiple interfaces by using comma separation.

Listening to Multiple Sources

If you issue the following method calls in Lines 20 and 21

```
f.addMouseListener(this);
f.addMouseMotionListener(this);
```

both types of events cause methods to be called in the `TwoListener` class. An object can *listen* to as many event sources as required. The object's class needs to implement only the required interfaces.

Obtaining Details About the Event

The event arguments with which handler methods, such as `mouseDragged()`, are called contain potentially important information about the original event. To determine the details of what information is available for each category of event, check the appropriate class documentation in the `java.awt.event` package.

Multiple Listeners

The AWT event listening framework permits the attachment of multiple listeners to the same component. In general, if you choose to write a program that performs multiple actions based on a single event, include code for that behavior in your handler method. However, sometimes a program's design requires multiple, unrelated parts of the same program to react to the same event. This might happen if, for example, a context-sensitive help system is added to an existing program.

The listener mechanism enables you to call an `addXXXListener()` method as many times as needed, and you can specify as many different listeners as your design requires. All registered listeners have their handler methods called when the event occurs.

Note – The order in which the handler methods are called is undefined. Generally, if the order of invocation matters, then the handlers are not unrelated. In this case, register only the first listener and have that listener call others directly.



Developing Event Listeners

In this section, you will learn about the design and implementation choices for implementing event listeners.

Event Adapters

It is a lot of work to implement all of the methods in each of the listener interfaces, particularly the `MouseListener` interface and `WindowListener` interface.

For example, the `MouseListener` interface declares the following methods:

```
public void mouseClicked(MouseEvent event)
public void mouseEntered(MouseEvent event)
public void mouseExited(MouseEvent event)
public void mousePressed(MouseEvent event)
public void mouseReleased(MouseEvent event)
```

As a convenience, the Java programming language provides adapter classes that implement each interface containing more than one method. The methods in these adapter classes are empty.

You can extend an adapter class and override only those methods that you need, as shown in Code 13-4.

Code 13-4 The MouseClickHandler Example

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;
4
5 public class MouseClickHandler extends MouseAdapter {
6
7     // We just need the mouseClicked handler, so we use
8     // an adapter to avoid having to write all the
9     // event handler methods
10
11    public void mouseClicked(MouseEvent e) {
12        // Do stuff with the mouse click...
13    }
14}
```

Event Handling Using Inner Classes

Lines 26 and 12–18 in Code 13-5 show how to create event handlers as inner classes. Using inner classes for event handlers gives you access to the private data of the outer class (Line 16).

Code 13-5 The TestInner Example

```

1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  public class TestInner {
5      private JFrame f;
6      private JTextField tf;
7
8      public TestInner() {
9          f = new JFrame("Inner classes example");
10         tf = new JTextField(30);
11     }
12
13     class MyMouseMotionListener extends MouseMotionAdapter {
14         public void mouseDragged(MouseEvent e) {
15             String s = "Mouse dragging: X = " + e.getX()
16                     + " Y = " + e.getY();
17             tf.setText(s);
18         }
19     }
20
21     public void launchFrame() {
22         JLabel label = new JLabel("Click and drag the mouse");
23         // Add components to the frame
24         f.add(label, BorderLayout.NORTH);
25         f.add(tf, BorderLayout.SOUTH);
26         // Add a listener that uses an Inner class
27         f.addMouseMotionListener(new MyMouseMotionListener());
28         f.addMouseListener(new MouseClickHandler());
29         // Size the frame and make it visible
30         f.setSize(300, 200);
31         f.setVisible(true);
32     }
33
34     public static void main(String args[]) {
35         TestInner obj = new TestInner();
36         obj.launchFrame();
37     }
38 }
```

Event Handling Using Anonymous Classes

You can include an entire class definition within the scope of an expression. This approach defines what is called an *anonymous* inner class and creates the instance all at one time. Anonymous inner classes are often used in event handling, Code 13-6 shows an example.

Code 13-6 The TestAnonymous Example

```

1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;
4
5 public class TestAnonymous {
6     private JFrame f;
7     private JTextField tf;
8
9     public TestAnonymous() {
10         f = new JFrame("Anonymous classes example");
11         tf = new JTextField(30);
12     }
13
14     public void launchFrame() {
15         JLabel label = new JLabel("Click and drag the mouse");
16         // Add components to the frame
17         f.add(label, BorderLayout.NORTH);
18         f.add(tf, BorderLayout.SOUTH);
19         // Add a listener that uses an anonymous class
20         f.addMouseMotionListener(new MouseMotionAdapter() {
21             public void mouseDragged(MouseEvent e) {
22                 String s = "Mouse dragging: X = " + e.getX()
23                         + " Y = " + e.getY();
24                 tf.setText(s);
25             }
26         }); // <- note the closing parenthesis
27         f.addMouseListener(new MouseClickHandler()); // Not shown
28         // Size the frame and make it visible
29         f.setSize(300, 200);
30         f.setVisible(true);
31     }
32
33     public static void main(String args[]) {
34         TestAnonymous obj = new TestAnonymous();
35         obj.launchFrame();
36     }
37 }
```

Developing Event Listeners



Note – The compilation of an anonymous class generates a file, such as TestAnonymous\$1.class.

Concurrency in Swing

Applications that contain a GUI require several threads for handling the GUI efficiently.

- Threads responsible for executing the application code are called current threads.
- Threads responsible for handling the events generated by various components are called event dispatch threads.
- Threads responsible for executing lengthy tasks are called worker threads. Some examples of these lengthy tasks include waiting for some shared resource, waiting for user input, blocking for network or disk I/O, performing either CPU or memory-intensive calculations. These tasks can be executed in the background without affecting the performance of the GUI.

You can use instances of the `SwingWorker` class to represent these worker threads. The `SwingWorker` class extends the `Object` class and implements the `RunnableFuture` interface.

The `SwingWorker` class provides the following utility methods:

- For communication and coordination between worker thread tasks and tasks on other threads, the `SwingWorker` class provides properties such as `progress` and `state` to support inter-thread communication.
- To execute simple background tasks, the `doInBackground` method can be used for running the tasks background.
- To execute tasks that have intermediate results, the results are published in a GUI using the `publish` and `process` methods.
- To cancel the background threads, they can be canceled using the `cancel` method.

Note – A full description of the use of the `SwingWorker` class is outside the scope of this module. For more information, see the following URL:
<http://java.sun.com/docs/books/tutorial/uiswing/concurrency/index.html>.



Module 14

GUI-Based Applications

Objectives

Upon completion of this module, you should be able to:

- Describe how to construct a menu bar, menu, and menu items in a Java GUI
- Understand how to change the color and font of a component

Relevance



Relevance

Discussion – The following questions are relevant to the material presented in this module:

- You now know how to set up a Java GUI for both graphic output and interactive user input. However, only a few of the components from which GUIs can be built have been described. What other components would be useful in a GUI?

- How can you create a menu for your GUI frame?

How to Create a Menu

A `JMenu` is different from other components because you cannot add a `JMenu` component to ordinary containers and have them laid out by the layout manager. You can add menus only to a *menu container*. This section describes the following procedure to create a menu:

1. Create a `JMenuBar` object, and set it into a menu container, such as a `JFrame`.
2. Create one or more `JMenu` objects, and add them to the menu bar object.
3. Create one or more `JMenuItem` objects, and add them to the menu object.

Note – Pop-up menus are an exception to this rule because they appear as floating windows and, therefore, do not require layout.



Note – With AWT, you could designate one menu to be the Help menu. This was specified by using the `setHelpMenu` method. This method is not implemented in Swing and Help menus are added just like any other menu item.

Creating a JMenuBar

A JMenuBar component is a horizontal menu. You can add it to a JFrame object only, and it forms the root of all menu trees. A JFrame displays one JMenuBar component at a time. However, you can change the JMenuBar based on the state of the program so that different menus appear at various points.

Code 14-1 JMenuBar

```
10  f = new JFrame ("JMenuBar") ;  
11  mb = new JMenuBar () ;  
12  f.setJMenuBar (mb) ;
```

The following is what the code produces in Microsoft Windows.



Figure 14-1 The JMenuBar Component

Note – The appearance of the window varies slightly from operating system to operating system.

The JMenuBar does not support listeners. All the events that might arise in a menu bar are processed by the menus that are added to the menu bar.



Creating a JMenu

The JMenu component provides a basic pull-down menu. You add it either to a JMenuBar or to another JMenu.

Code 14-2 JMenuBar with Top Level JMenu Items

```
13  f = new JFrame("Menu");
14  mb = new JMenuBar();
15  m1 = new JMenu("File");
16  m2 = new JMenu("Edit");
17  m3 = new JMenu("Help");
18  mb.add(m1);
19  mb.add(m2);
20  mb.add(m3);
21  f.setJMenuBar(mb);
```

The code produces the following:

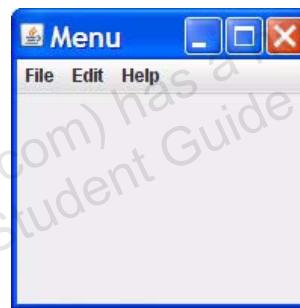


Figure 14-2 The JMenu Component

You *can* add an ActionListener to a JMenu object, but this is unusual. Normally, you use menus to display and control menu items that are described in the following section.

Creating a JMenuItem

The JMenuItem components are the text *leaf* nodes of a menu tree. They are added to a menu to complete it, as shown in the following example.

Code 14-3 JMenuItem Added to a Menu

```
28  mi1 = new JMenuItem("New");
29  mi2 = new JMenuItem("Save");
30  mi3 = new JMenuItem("Load");
```

How to Create a Menu

```
31 mi4 = new JMenuItem("Quit");
32 mi1.addActionListener(this);
33 mi2.addActionListener(this);
34 mi3.addActionListener(this);
35 mi4.addActionListener(this);
36 m1.add(mi1);
37 m1.add(mi2);
38 m1.add(mi3);
39 m1.addSeparator();
40 m1.add(mi4);
```

This code produces the following:



Figure 14-3 The JMenuItem Component

Usually, you add an ActionListener to a JMenuItem object to provide behavior for the menus.

Creating a JCheckboxMenuItem

The JCheckboxMenuItem is a checkable menu item, so you can have selections (*on* or *off* choices) listed in menus, as shown in the following example.

Demonstration - JCheckboxMenuItem

```
19 f = new JFrame ("CheckboxMenuItem");
20 mb = new JMenuBar();
21 m1 = new JMenu ("File");
22 m2 = new JMenu ("Edit");
23 m3 = new JMenu ("Help");
24 mb.add(m1);
25 mb.add(m2);
26 mb.add(m3);
27 f.setJMenuBar(mb);
```

```
.....  
43 mi5 = new JCheckBoxMenuItem("Persistent");  
44 mi5.addItemListener(this);  
45 m1.add(mi5);
```

The code produces the following:

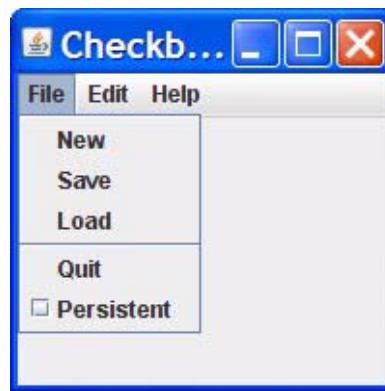


Figure 14-4 The JCheckboxMenuItem Component

You should monitor the JCheckboxMenuItem using the ItemListener interface.

Controlling Visual Aspects

You can control the colors used for the foreground and the background of AWT components.

Colors

You use two methods to set the colors of a component:

```
setForeground()  
setBackground()
```

Both of these methods take an argument that is an instance of the `java.awt.Color` class. You can use constant colors referred to as `Color.red`, `Color.blue`, and so on. The full range of predefined colors is listed in the documentation page for the `Color` class.

You can also construct a specific `Color` object by specifying the color by a combination of three byte-sized integers (0–255), one for each primary color: red, blue, and green. For example:

```
Color purple = new Color(255, 0, 255);  
JButton b = new JButton("Purple");  
b.setBackground(purple);
```

Module 15

Threads

Objectives

Upon completion of this module, you should be able to:

- Define a thread
- Create separate threads in a Java technology program, controlling the code and data that are used by that thread
- Control the execution of a thread and write platform-independent code with threads
- Describe the difficulties that might arise when multiple threads share data
- Use `wait` and `notify` to communicate between threads
- Use `synchronized` to protect data from corruption

This module covers multithreading, which enables a program to perform multiple tasks at the same time.

Relevance

Relevance

Discussion – The following question is relevant to the material presented in this module:

How do you get programs to perform multiple tasks concurrently?



Threads

A simplistic view of a computer is that it has a CPU that performs computations, memory that contains the program that the CPU executes, and memory that holds the data on which the program operates. In this view, there is only one job performed. A more complete view of most modern computer systems provides for the possibility of performing more than one job at the same time.

You do not need to be concerned with how multiple-job performance is achieved, just consider the implications from a programming point of view. Performing more than one job is similar to having more than one computer. In this module, a *thread*, or *execution context*, is considered to be the encapsulation of a *virtual CPU* with its own program code and data. The class `java.lang.Thread` enables you to create and control threads.



Note – This module uses the term *Thread* when referring to the class `java.lang.Thread` and *thread* when referring to an execution context.

A thread, or execution context, is composed of three main parts:

- A virtual CPU
- The code that the CPU executes
- The data on which the code works

A *process* is a program in execution. One or more threads constitute a process. A thread is composed of CPU, code, and data, as illustrated in Figure 15-1.

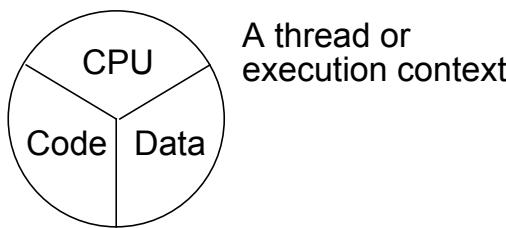


Figure 15-1 A Thread

Code can be shared by multiple threads, independent of data. Two threads share the same code when they execute code from instances of the same class.

Likewise, data can be shared by multiple threads, independent of code. Two threads share the same data when they share access to a common object.

In Java programming, the virtual CPU is encapsulated in an instance of the Thread class. When a thread is constructed, the code and the data that define its context are specified by the object passed to its constructor.

Creating the Thread

This section examines how you create a thread, and how you use constructor arguments to supply the code and data for a thread when it runs.

A Thread constructor takes an argument that is an *instance* of Runnable. An instance of Runnable is made from a class that implements the Runnable interface (that is, it provides a public void run() method).

For example:

```
1  public class ThreadTester {
2      public static void main(String args[]) {
3          HelloRunner r = new HelloRunner();
4          Thread t = new Thread(r);
5          t.start();
6      }
7  }
8
9  class HelloRunner implements Runnable {
10    int i;
11
12    public void run() {
13        i = 0;
14
15        while (true) {
16            System.out.println("Hello " + i++);
17            if ( i == 50 ) {
18                break;
19            }
20        }
21    }
22 }
```

First, the main method constructs an instance `r` of class HelloRunner. Instance `r` has its own data, in this case the integer `i`. Because the instance, `r`, is passed to the Thread class constructor, `r`'s integer `i` is the data with which the thread works when it runs. The thread always begins executing at the `run` method of its loaded Runnable instance (`r` in this example).

A multithreaded programming environment enables you to create multiple threads based on the same Runnable instance. You can do this as follows:

```
Thread t1 = new Thread(r);
Thread t2 = new Thread(r);
```

In this case, both threads share the same data and code.

To summarize, a thread is referred to through an instance of a Thread object. The thread begins execution at the start of a loaded Runnable instance's `run` method. The data that the thread works on is taken from the *specific* instance of Runnable, which is passed to that Thread constructor (Figure 15-2).

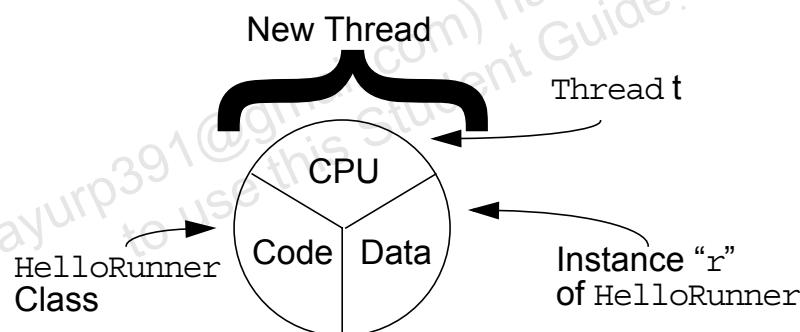


Figure 15-2 Thread Creation

Starting the Thread

A newly created thread does not start running automatically. You must call its `start` method. For example, you can issue the following command as on Line 5 of the previous example:

```
t.start();
```

Calling `start` places the virtual CPU embodied in the thread into a runnable state, meaning that it becomes viable for scheduling for execution by the JVM. This does not necessarily mean that the thread runs immediately.

Thread Scheduling

Usually, in Java technology threads are *pre-emptive*, but not necessarily time-sliced (the process of giving each thread an equal amount of CPU time). It is a common mistake to believe that *pre-emptive* is another word for *does time-slicing*.

The model of a pre-emptive scheduler is that many threads might be runnable, but only one thread is running. This thread continues to run until it ceases to be runnable or until another thread of higher priority becomes runnable. In the latter case, the lower priority thread is *pre-empted* by the thread of higher priority, which gets a chance to run instead.

A thread might cease to be runnable (that is, become *blocked*) for a variety of reasons. The thread's code can execute a `Thread.sleep()` call, asking the thread to pause deliberately for a fixed period of time. The thread might have to wait to access a resource and cannot continue until that resource becomes available.

All threads that are runnable are kept in pools according to priority. When a blocked thread becomes runnable, it is placed back into the appropriate runnable pool. Threads from the highest priority non-empty pool are given CPU time.

A Thread object can exist in several different states throughout its lifetime as shown in Figure 15-3.

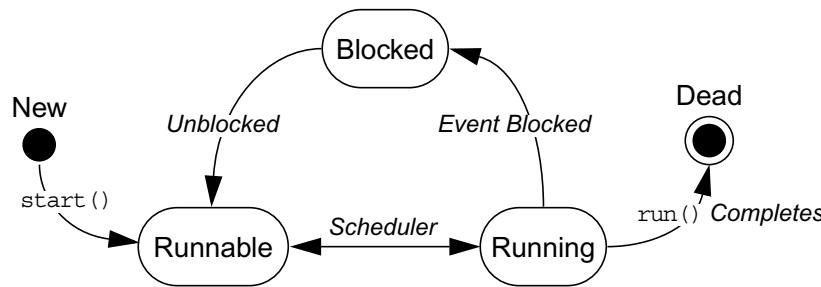


Figure 15-3 Fundamental Thread State Diagram

Although the thread becomes runnable, it does not always start running immediately. Only one action at a time is performed on a machine with one CPU. The following paragraphs describe how the CPU is allocated when more than one thread is runnable.

Given that Java threads are not necessarily time-sliced, you must ensure that the code for your threads gives other threads a chance to execute from time to time. This can be achieved by issuing the `sleep` call at various intervals, as shown in Code 15-1.

Code 15-1 Thread Scheduling Example

```

1  public class Runner implements Runnable {
2      public void run() {
3          while (true) {
4              // do lots of interesting stuff
5              // ...
6              // Give other threads a chance
7              try {
8                  Thread.sleep(10);
9              } catch (InterruptedException e) {
10                  // This thread's sleep was interrupted
11                  // by another thread
12              }
13          }
14      }
15  }

```

Threads

Code 15-1 on page 15-7 shows how the `try` and `catch` block is used. The `Thread.sleep()` and other methods that can pause a thread for periods of time are interruptible. Threads can call another thread's `interrupt` method, which signals the paused thread with an `InterruptedException`.

The `sleep` is a static method in the `Thread` class, because it operates on the current thread and is referred to as `Thread.sleep(x)`. The `sleep` method's argument specifies the minimum number of milliseconds for which the thread must be made inactive. The execution of the thread does not resume until after this period unless it is interrupted, in which case execution is resumed earlier.

Terminating a Thread

When a thread completes execution and terminates, it *cannot* run again.

You can stop a thread by using a flag that indicates that the `run` method should exit.

```
1  public class Runner implements Runnable {  
2      private boolean timeToQuit=false;  
3  
4      public void run() {  
5          while ( ! timeToQuit ) {  
6              // do work until we are told to quit  
7          }  
8          // clean up before run() ends  
9      }  
10  
11     public void stopRunning() {  
12         timeToQuit=true;  
13     }  
14 }
```



```
1  public class ThreadController {  
2      private Runner r = new Runner();  
3      private Thread t = new Thread(r);  
4  
5      public void startThread() {  
6          t.start();  
7      }  
8  
9      public void stopThread() {  
10         // use specific instance of Runner
```

```
11      r.stopRunning();  
12  }  
13 }
```

Within a particular piece of code, you can obtain a reference to the current thread using the static Thread method `currentThread`. For example:

```
1  public class NameRunner implements Runnable {  
2      public void run() {  
3          while (true) {  
4              // lots of interesting stuff  
5          }  
6          // Print name of the current thread  
7          System.out.println("Thread "  
8                  + Thread.currentThread().getName()  
9                  + " completed");  
10     }  
11 }
```

Basic Control of Threads

This section describes how to control threads.

Testing Threads

A thread can be in an unknown state. Use the method `isAlive` to determine if a thread is still viable. The term *Alive* does not imply that the thread is running; it returns `true` for a thread that has been started but has not completed its task.

Accessing Thread Priority

Use the `getPriority` method to determine the current priority of the thread. Use the `setPriority` method to set the priority of the thread. The priority is an integer value. The `Thread` class includes the following constants:

`Thread.MIN_PRIORITY`
`Thread.NORM_PRIORITY`
`Thread.MAX_PRIORITY`

Putting Threads on Hold

Mechanisms exist that can temporarily block the execution of a thread. You can resume execution as if nothing happened. The thread appears to have executed an instruction very slowly.

The Thread.sleep() Method

The `sleep` method is one way to halt a thread for a period of time. Recall that the thread does not necessarily resume its execution at the instant that the sleep period expires. This is because some other thread could be executing at that instant and might not be unscheduled unless one of the following occurs:

- The thread *waking up* is of a higher priority.
- The running thread blocks for some other reason.

The join Method

The `join` method causes the current thread to wait until the thread on which the `join` method is called terminates. For example:

```
1 public static void main(String[] args) {  
2     Thread t = new Thread(new Runner());  
3     t.start();  
4     ...  
5     // Do stuff in parallel with the other thread for a while  
6     ...  
7     // Wait here for the timer thread to finish  
8     try {  
9         t.join();  
10    } catch (InterruptedException e) {  
11        // t came back early  
12    }  
13    ...  
14    // Now continue in this thread  
15    ...  
16 }
```

Basic Control of Threads

You can also call the `join` method with a time-out value in milliseconds.
For example:

```
void join(long timeout);
```

For this example, the `join` method either suspends the current thread for `timeout` milliseconds or until the thread it calls on terminates.

The `Thread.yield()` Method

Use the method `Thread.yield()` to give other runnable threads a chance to execute. If other threads are runnable, `yield` places the calling thread into the runnable pool and allows another runnable thread to run. If no other threads are runnable, `yield` does nothing.

A `sleep` call gives threads of lower priority a chance to execute. The `yield` method gives other runnable threads a chance to execute.

Other Ways to Create Threads

So far, you have seen how you can create thread contexts with a separate class that implements `Runnable`. In fact, this is not the only possible approach. The `Thread` class implements the `Runnable` interface itself, so you can create a thread by creating a class that extends `Thread` rather than implements `Runnable`.

```
1  public class MyThread extends Thread {  
2      public void run() {  
3          while ( true ) {  
4              // do lots of interesting stuff  
5              try {  
6                  Thread.sleep(100);  
7              } catch (InterruptedException e) {  
8                  // sleep interrupted  
9              }  
10         }  
11     }  
12  
13     public static void main(String args[] ) {  
14         Thread t = new MyThread();  
15         t.start();  
16     }  
17 }
```

Selecting a Way to Create Threads

Given a choice of approaches to creating a thread, how can you decide between them? Each approach has its advantages, which are described in this section.

The following describes the advantages of implementing Runnable:

- From an object-oriented design point of view, the Thread class is strictly an encapsulation of a virtual CPU and, as such, it should be extended only when you change or extend the behavior of that CPU model. Because of this and the value of making the distinction between the CPU, code, and data parts of a running thread, this course module has used this approach.
- Because Java technology permits single inheritance only, you cannot extend any other class, such as Applet, if you extended Thread already. In some situations, this forces you to take the approach of implementing Runnable.
- Because there are times when you are obliged to implement Runnable, you might prefer to be consistent and always do it this way.

The advantage of extending Thread is that the code tends to be simpler.

Note – While both techniques are possible, you should consider very carefully why you would extend Thread. Do so only when you change or extend the behavior of a thread, not when you implement a `run` method.



Using the synchronized Keyword

This section describes the use of the `synchronized` keyword. It provides the Java programming language with a mechanism that enables a programmer to control threads that are sharing data.

The Problem

Imagine a class that represents a stack. This class might appear first as:

```
1  public class MyStack {  
2      int idx = 0;  
3      char [] data = new char[6];  
4  
5      public void push(char c) {  
6          data[idx] = c;  
7          idx++;  
8      }  
9  
10     public char pop() {  
11         idx--;  
12         return data[idx];  
13     }  
14 }
```

The class makes no effort to handle the overflow or underflow of the stack, and the stack capacity is limited. However, these aspects are not relevant to this discussion.

The behavior of this model requires that the index value contains the array subscript of the next *empty* cell in the stack. The *predecrement*, *postincrement* approach generates this information.

Imagine now that *two* threads have a reference to a *single* instance of this class. One thread is pushing data onto the stack and the other, more or less independently, is popping data off of the stack. In principle, the data is added and removed successfully. However, there is a potential problem.

Suppose thread *a* is adding characters and thread *b* is removing characters. Thread *a* has just deposited a character, but has not yet incremented the index counter. For some reason, this thread is now pre-empted. At this point, the data model represented in the object is inconsistent.

Using the synchronized Keyword

```
buffer |p|q|r| | | |
idx = 2 ^
```

Specifically, consistency requires either `idx = 3` or that the character has not yet been added.

If thread *a* resumes execution, there might be no damage, but suppose thread *b* was waiting to remove a character. While thread *a* is waiting for another chance to run, thread *b* gets its chance to remove a character.

There is an inconsistent data situation on entry to the `pop` method, yet the `pop` method proceeds to decrement the index value.

```
buffer |p|q|r| | | |
idx = 1 ^
```

This effectively serves to ignore the character `r`. After this, it then returns the character `q`. So far, the behavior has been as if the letter `r` had not been pushed, so it is difficult to say that there is a problem. But look at what happens when the original thread, *a*, continues to run.

Thread *a* picks up where it left off, in the `push` method, and it proceeds to increment the index value. Now you have the following:

```
buffer |p|q|r| | | |
idx = 2 ^
```

This configuration implies the `q` is valid and the cell containing `r` is the next empty cell. In other words, `q` is read as having been placed into the stack twice, and the letter `r` never appears.

This is a simple example of a general problem that arises when *multiple* threads are accessing *shared* data. You need a mechanism to ensure that shared data is in a consistent state before any thread starts to use it for a particular task.

One approach would be to prevent thread *a* from being switched out until it completes the critical section of code. This approach is common in low-level machine programming but is generally inappropriate in multi-user systems.

Another approach, and the one on which Java technology works, is to provide a mechanism to treat the data *delicately*. This approach provides a thread atomic with access to data regardless of whether that thread gets switched out in the middle of performing that access.

The Object Lock Flag

In Java technology, every object has a flag associated with it. You can think of this flag as a *lock flag*. The keyword `synchronized` enables interaction with this flag, and provides exclusive access to code that affects shared data. The following is the modified code fragment:

```
public class MyStack {
    ...
    public void push(char c) {
        synchronized(this) {
            data[idx] = c;
            idx++;
        }
    }
    ...
}
```

When the thread reaches the `synchronized` statement, it examines the object passed as the argument, and tries to obtain the lock flag from that object before continuing (see Figure 15-4).

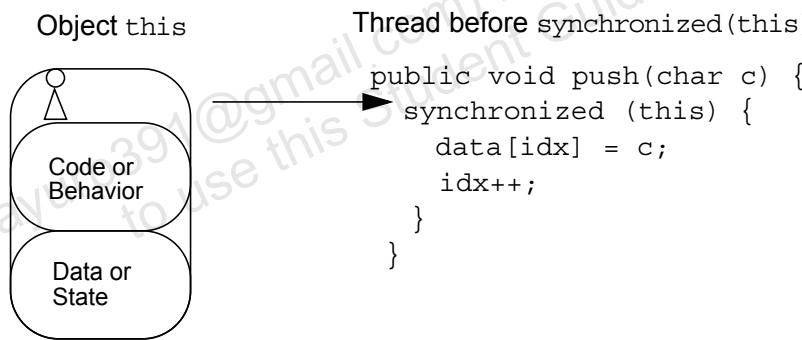


Figure 15-4 Using the `synchronized` Statement Before a Thread

Using the synchronized Keyword

An example of using the synchronized statement after a thread is shown in Figure 15-5.

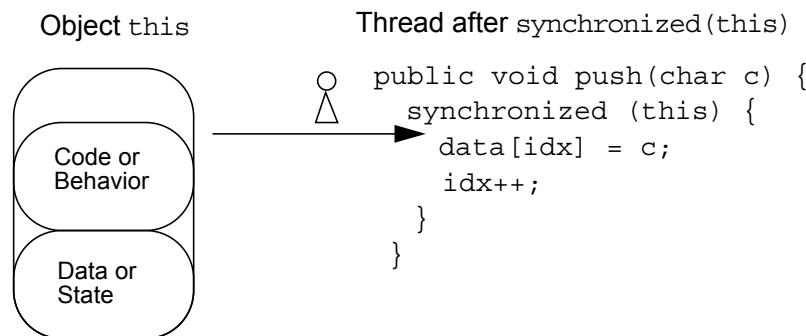


Figure 15-5 Using the synchronized Statement After a Thread

You should realize that this does not protect the data. If the pop method of the shared data object is not protected by synchronized, and pop is invoked by another thread, *there is still a risk of damaging the consistency of the data*. All methods accessing shared data must synchronize on the same lock if the lock is to be effective.

Figure 15-6 illustrates what happens if pop is protected by synchronized and another thread tries to execute an object's pop method while the original thread holds the synchronized object's lock flag.

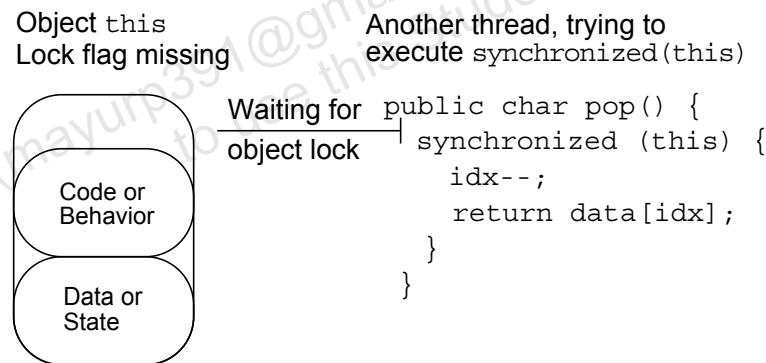


Figure 15-6 Thread Trying to Execute synchronized

When the thread tries to execute the synchronized(this) statement, it tries to take the lock flag from the object this. Because the flag is not present, the thread cannot continue execution. The thread then joins a pool of waiting threads that are associated with *that* object's lock flag. When the flag is returned to the object, a thread that was waiting for the flag is given it, and the thread continues to run.

Releasing the Lock Flag

A thread waiting for the lock flag of an object cannot resume running until the flag is available. Therefore, it is important for the holding thread to return the flag when it is no longer needed.

The lock flag is given back to its object automatically. When the thread that holds the lock passes the end of the synchronized code block for which the lock was obtained, the lock is released. Java technology ensures that the lock is always returned automatically, even if an encountered exception, a break statement, or a return statement transfers code execution out of a synchronized block. Also, if a thread executes nested blocks of code that are synchronized on the same object, that object's flag is released correctly on exit from the outermost block and the innermost block is ignored.

These rules make using synchronized blocks much simpler to manage than equivalent facilities in some other systems.

Using the synchronized Keyword

Using synchronized – Putting It Together

The synchronized mechanism works only if *all* access to delicate data occurs within the synchronized blocks.

You should mark delicate data protected by synchronized blocks as *private*. If you do not do this the delicate data can be accessed from code outside the class definition; such a situation would enable other programmers to bypass your protection and cause data corruption at runtime.

A method consisting entirely of code belonging in a block synchronized to this instance might put the synchronized keyword in its header. The following two code fragments are equivalent:

```
public void push(char c) {  
    synchronized(this) {  
        // The push method code  
    }  
}  
  
public synchronized void push(char c) {  
    // The push method code  
}
```

Why use one technique instead of the other?

If you use synchronized as a method modifier, the whole method becomes a synchronized block. That can result in the lock flag being held longer than necessary.

However, marking the method in this way permits users of the method to know, from javadoc utility-generated documentation, that synchronization is taking place. This can be important when designing against deadlock (which is described in the following section). The javadoc documentation generator propagates the synchronized modifier into documentation files, but it cannot do the same for synchronized(this), which is found *inside* the method's block.

Thread States

Synchronization is a special thread state. Figure 15-7 illustrates the new state transition diagram for a thread.

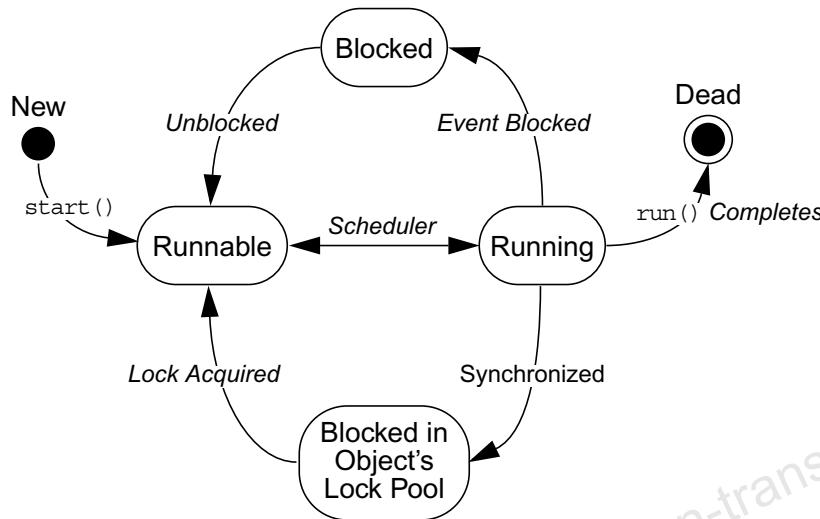


Figure 15-7 Thread State Diagram With Synchronization

Deadlock

In programs where multiple threads are competing for access to multiple resources, a condition known as *deadlock* can occur. This occurs when one thread is waiting for a lock held by another thread, but the other thread is waiting for a lock already held by the first thread. In this condition, neither can proceed until after the other has passed the end of its synchronized block. Because neither is able to proceed, neither can pass the end of its block.

Java technology neither detects nor attempts to avoid this condition. It is the responsibility of the programmer to ensure that a deadlock cannot arise. A general rule of thumb for avoiding a deadlock is: If you have multiple objects that you want to have synchronized access to, make a global decision about the order in which you will obtain those locks, and adhere to that order throughout the program. Release the locks in the reverse order that you obtained them.

Thread Interaction – wait and notify

Different threads are created specifically to perform unrelated tasks. However, sometimes the jobs they perform are related in some way and it might be necessary to program some interactions between them.

Scenario

Consider yourself and a cab driver as two threads. You need a cab to take you to a destination and the cab driver wants to take on a passenger to make a fare. So, each of you has a task.

The Problem

You expect to get into a cab and rest comfortably until the cab driver notifies you that you have arrived at your destination. It would be annoying, for both you and the cab driver, to ask every 2 seconds, “Are we there yet?” Between fares, the cab driver wants to sleep in the cab until a passenger needs to be driven somewhere. The cab driver does not want to have to wake up from this nap every 5 minutes to see if a passenger has arrived at the cab stand. So, both threads would prefer to get their jobs done in as relaxed a manner as possible.

The Solution

You and the cab driver require some way of communicating your needs to each other. While you are busy walking down the street toward the cab stand, the cab driver is sleeping peacefully in the cab. When you notify the cab driver that you want a ride, the driver wakes up and begins driving, and you get to relax. After you arrive at your destination, the cab driver notifies you to get out of the cab and go to work. The cab driver now gets to wait and nap again until the next fare comes along.

Thread Interaction

This section describes how threads interact.

The wait and notify Methods

The `java.lang.Object` class provides two methods, `wait` and `notify`, for thread communication. If a thread issues a `wait` call on a rendezvous object `x`, that thread pauses its execution until another thread issues a `notify` call on the same rendezvous object `x`.

In the previous scenario, the cab driver waiting in the cab translates to the `cab driver` thread executing a `cab.wait` call, and your need to use the cab translates to the `you` thread executing a `cab.notify()` call.

For a thread to call either `wait` or `notify` on an object, the thread must have the lock for that particular object. In other words, `wait` and `notify` are called only from within a synchronized block on the instance on which they are being called. For this example, you require a block starting with `synchronized(cab)` to permit either the `cab.wait` or the `cab.notify()` call.

The Pool Story

When a thread executes synchronized code that contains a `wait` call on a particular object, that thread is placed in the wait pool for that object. Additionally, the thread that calls `wait` releases that object's lock flag automatically. You can invoke different `wait` methods.

```
wait()  
wait(long timeout)
```

When a `notify` call is executed on a particular object, an *arbitrary* thread is moved from that object's wait pool to a lock pool, where threads stay until the object's lock flag becomes available. The `notifyAll` method moves all threads waiting on that object out of the wait pool and into the lock pool. Only from the lock pool can a thread obtain that object's lock flag, which enables the thread to continue running where it left off when it called `wait`.

Thread Interaction

In many systems that implement the wait-notify mechanism, the thread that wakes up is the one that has waited the longest. However, Java technology does not guarantee this.

You can issue a `notify` call without regard to whether any threads are waiting. If the `notify` method is called on an object when no threads are blocked in the wait pool for that object's lock flag, the call has no effect. Calls to `notify` are not stored.

Thread States

The `wait pool` is also a special thread state. Figure 15-8 illustrates the final state transition diagram for a thread.

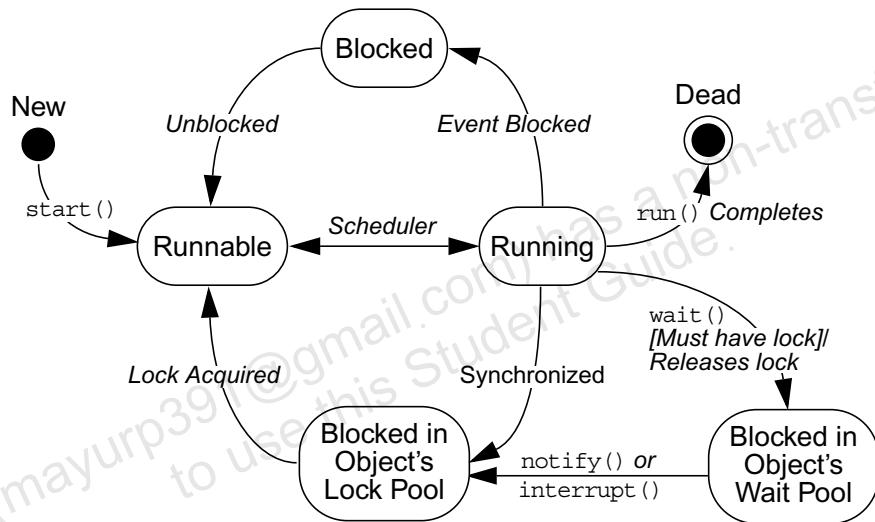


Figure 15-8 Thread States Diagram With `wait` and `notify`

Monitor Model for Synchronization

Coordination between two threads needing access to *common* data can get complex. You must ensure that no thread leaves shared data in an inconsistent state when there is the possibility that any other thread can access that data. You also must ensure that your program does not deadlock, because threads cannot release the appropriate lock when other threads are waiting for that lock.

In the cab example, the code relied on one rendezvous object, the cab, on which wait and notify were executed. If someone was expecting a bus, you would need a separate bus object on which to apply notify.

Remember that all threads in the same wait pool must be satisfied by notification from *that* wait pool's controlling object. Never design code that puts threads expecting to be notified for *different* conditions in the *same* wait pool.

Putting It Together

The code in this section is an example of thread interaction that demonstrates the use of `wait` and `notify` methods to solve a classic producer-consumer problem.

Start by looking at the outline of the stack object and the details of the threads that access it. Then look at the details of the stack and the mechanisms used to protect the stack's data and to implement the thread communication based on the stack's state.

The example stack class, called `SyncStack` to distinguish it from the core class `java.util.Stack`, offers the following public API:

```
public synchronized void push(char c);  
public synchronized char pop();
```

The Producer Thread

The producer thread generates new characters to be placed on the stack. Code 15-2 shows the Producer class.

Code 15-2 The Producer Class

```
1  package mod13;
2
3  public class Producer implements Runnable {
4      private SyncStack theStack;
5      private int num;
6      private static int counter = 1;
7
8      public Producer (SyncStack s) {
9          theStack = s;
10         num = counter++;
11     }
12
13     public void run() {
14         char c;
15
16         for (int i = 0; i < 200; i++) {
17             c = (char)(Math.random() * 26 + 'A');
18             theStack.push(c);
19             System.out.println("Producer" + num + ":" + c);
20             try {
21                 Thread.sleep((int)(Math.random() * 300));
22             } catch (InterruptedException e) {
23                 // ignore it
24             }
25         }
26     } // END run method
27
28 } // END Producer class
```

This example generates 200 random peerages characters and pushes them onto the stack with a random delay of 0–300 milliseconds between each push. Each pushed character is reported on the console, along with an identifier for which producer thread is executing.

The Consumer Thread

The consumer thread removes characters from the stack. Code 15-3 shows the Consumer class.

Code 15-3 The Consumer Class

```
1 package mod13;
2
3 public class Consumer implements Runnable {
4     private SyncStack theStack;
5     private int num;
6     private static int counter = 1;
7
8     public Consumer (SyncStack s) {
9         theStack = s;
10        num = counter++;
11    }
12
13    public void run() {
14        char c;
15        for (int i = 0; i < 200; i++) {
16            c = theStack.pop();
17            System.out.println("Consumer" + num + ":" + c);
18
19            try {
20                Thread.sleep((int)(Math.random() * 300));
21            } catch (InterruptedException e) {
22                // ignore it
23            }
24        }
25    } // END run method
26
27 } // END Consumer class
```

This example collects 200 characters from the stack, with a random delay of 0–300 milliseconds between each attempt. Each uppercase character is reported on the console, along with an identifier to identify the consumer thread that is executing.

Now consider construction of the stack class. You are going to create a stack that has a seemingly limitless size, using the `ArrayList` class. With this design, your threads have only to communicate based on whether the stack is empty.

The SyncStack Class

A newly constructed SyncStack object's buffer should be empty. You can use the following code to build your class:

```
public class SyncStack {  
  
    private List<Character> buffer  
        = new ArrayList<Character>(400);  
  
    public synchronized char pop() {  
        // pop code here  
    }  
  
    public synchronized void push(char c) {  
        // push code here  
    }  
}
```

There are no constructors. It is considered good style to include a constructor, but it has been omitted here for brevity.

The pop Method

Now consider the push and pop methods. They must be synchronized to protect the shared buffer. In addition, if the stack is empty in the pop method, the executing thread must wait. When the stack in the push method is no longer empty, waiting threads are notified. Code 15-4 shows the pop method.

Code 15-4 The pop Method

```

1  public synchronized char pop() {
2      char c;
3      while (buffer.size() == 0) {
4          try {
5              this.wait();
6          } catch (InterruptedException e) {
7              // ignore it...
8          }
9      }
10     c = buffer.remove(buffer.size() - 1);
11     return c;
12 }
```

The `wait` call is made with respect to the stack object that shows how the rendezvous is being made with a *particular object*. Nothing can be popped from the stack when it is empty, so a thread trying to pop data from the stack must wait until the stack is no longer empty.

The `wait` call is placed in a try-catch block because an interrupt call can terminate the thread's waiting period. The `wait` must also be within a loop for this example. If for some reason (such as an interrupt) the thread wakes up and discovers that the stack is still empty, then the thread must re-enter the waiting condition.

The pop method for the stack is synchronized for two reasons. First, popping a character off of the stack affects the shared data buffer. Second, the call to `this.wait()` must be within a block that is synchronized on the stack object, which is represented by `this`.

The push method uses `this.notify()` to release a thread from the stack object's wait pool. After a thread is released, it can obtain the lock on the stack and continue executing the pop method, which removes a character from the stack's buffer.



Note – In `pop`, the `wait` method is called *before* any character is removed from the stack. This is because the removal cannot proceed until some character is available.

You should also consider error checking. You might notice that there is no explicit code to prevent a stack underflow. This is not necessary because the only way to remove characters from the stack is through the `pop` method, and this method causes the executing thread to enter the `wait` state if no character is available. Therefore, error checking is unnecessary.

The push Method

The `push` method is similar to the `pop` method. It affects the shared buffer and must also be synchronized. In addition, because the `push` method adds a character to the buffer, it is responsible for notifying threads that are waiting for a non-empty stack. This notification is done with respect to the stack object.

Code 15-4 shows the `push` method.

Code 15-5 The push Method

```
1  public synchronized void push(char c) {  
2      this.notify();  
3      buffer.add(c);  
4  }  
5
```

The call to `this.notify()` serves to release a *single* thread that called `wait` because the stack is empty. Calling `notify` before the shared data is changed is of no consequence. The stack object's lock is released only upon exit from the synchronized block, so threads waiting for that lock can obtain it while the stack data are being changed by the `pop` method.

Putting It Together

Putting all of the pieces together, Code 15-6 shows the complete SyncStack class.

Code 15-6 The SyncStack Class

```

1  package mod13;
2
3  import java.util.*;
4
5  public class SyncStack {
6      private List<Character> buffer
7          = new ArrayList<Character>(400);
8
9      public synchronized char pop() {
10         char c;
11         while (buffer.size() == 0) {
12             try {
13                 this.wait();
14             } catch (InterruptedException e) {
15                 // ignore it...
16             }
17         }
18         c = buffer.remove(buffer.size() - 1);
19         return c;
20     }
21
22     public synchronized void push(char c) {
23         this.notify();
24         buffer.add(c);
25     }
26 }
```

The SyncTest Example

You must assemble the producer, consumer, and stack code into complete classes. A test harness is required to bring these pieces together. Pay particular attention to how SyncTest creates only one stack object that is *shared by all threads*. Code 15-7 shows the SyncTest class.

Code 15-7 The SyncTest Class

```

1  package mod13;
2
3  public class SyncTest {
4
```

```
5     public static void main(String[] args) {  
6  
7         SyncStack stack = new SyncStack();  
8  
9         Producer p1 = new Producer(stack);  
10        Thread prodT1 = new Thread (p1);  
11        prodT1.start();  
12  
13        Producer p2 = new Producer(stack);  
14        Thread prodT2 = new Thread (p2);  
15        prodT2.start();  
16  
17        Consumer c1 = new Consumer(stack);  
18        Thread consT1 = new Thread (c1);  
19        consT1.start();  
20  
21        Consumer c2 = new Consumer(stack);  
22        Thread consT2 = new Thread (c2);  
23        consT2.start();  
24    }  
25 }
```

The following is an example of the output from `java mod13.SyncTest`. Every time this thread code is run, the results vary.

```
Producer2: F  
Consumer1: F  
Producer2: K  
Consumer2: K  
Producer2: T  
Producer1: N  
Producer1: V  
Consumer2: V  
Consumer1: N  
Producer2: V  
Producer2: U  
Consumer2: U  
Consumer2: V  
Producer1: F  
Consumer1: F  
Producer2: M  
Consumer2: M  
Consumer2: T
```


Module 16

Networking

Objectives

At the end of this module, you should be able to:

- Develop code to set up the network connection
- Understand the TCP/IP Protocol
- Use ServerSocket and Socket classes for implementation of TCP/IP clients and servers

This module discusses Java 2 SDK support for sockets and socket programming. Socket programming communicates with other programs running on computers on the same network.

Relevance

Relevance

Discussion – The following question is relevant to the material presented in this module:

How can a communication link between a client machine and a server on the network be established?

Networking

The following section describes the concept of networking by using sockets.

Sockets

Socket is the name given, in one particular programming model, to the endpoints of a communication link between processes. Because of the popularity of that particular programming model, the term socket has been reused in other programming models, including Java technology.

When processes communicate over a network, Java technology uses the streams model. A socket can hold two streams: one input stream and one output stream. A process sends data to another process through the network by writing to the output stream associated with the socket. A process reads data written by another process by reading from the input stream associated with the socket.

After the network connection is set up, using the streams associated with that connection is similar to using any other stream.

Setting Up the Connection

To set up the connection, one machine must run a program that is waiting for a connection, and a second machine must try to reach the first. This is similar to a telephone system, in which one party must make the call, while the other party is waiting by the telephone when that call is made.

Networking

A description of the TCP/IP network connections is presented in this module. An example network connection is shown in Figure 16-1.

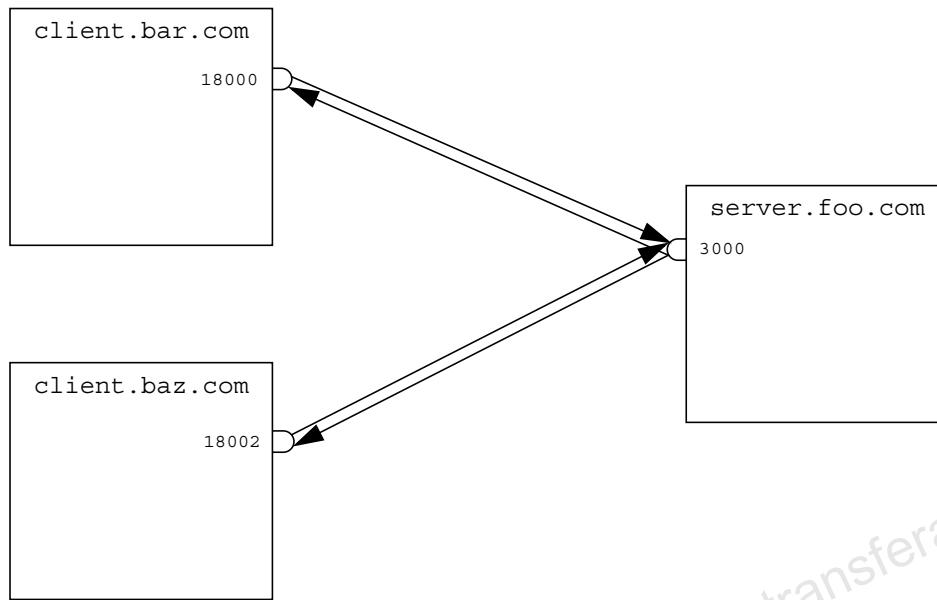


Figure 16-1 Diagram of Example Network Connections

Networking With Java Technology

This section describes the concept of networking by using Java technology.

Addressing the Connection

When you make a telephone call, you need to know the telephone number to dial. When you make a network connection, you need to know the address or the name of the remote machine. In addition, a network connection requires a port number, which you can think of as a telephone extension number. After you connect to the proper computer, you must identify a particular purpose for the connection. So, in the same way that you can use a particular telephone extension number to talk to the accounts department, you can use a particular port number to communicate with the accounting program.

Port Numbers

Port numbers in TCP/IP systems are 16-bit numbers and the values range from 0–65535. In practice, port numbers below 1024 are reserved for predefined services, and you should not use them unless communicating with one of those services (such as telnet, Simple Mail Transport Protocol [SMTP] mail, ftp, and so on). Client port numbers are allocated by the host OS to something not in use, while server port numbers are specified by the programmer, and are used to identify a particular service.

Both client and server must agree in advance on which port to use. If the port numbers used by the two parts of the system do not agree, communication does not occur.

Java Networking Model

In the Java programming language, TCP/IP socket connections are implemented with classes in the `java.net` package. Figure 16-2 illustrates what occurs on the server side and the client side.

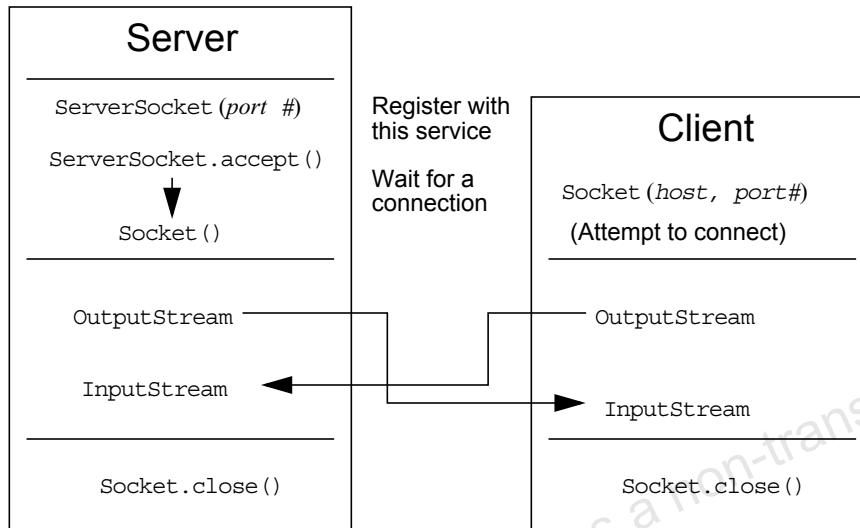


Figure 16-2 TCP/IP Socket Connections

In Figure 16-2:

- The server assigns a port number. When the client requests a connection, the server opens the socket connection with the `accept()` method.
- The client establishes a connection with `host` on port `port#`.
- Both the client and server communicate by using an `InputStream` and an `OutputStream`.

Minimal TCP/IP Server

TCP/IP server applications rely on the `ServerSocket` and `Socket` networking classes provided by the Java programming language. The `ServerSocket` class takes most of the work out of establishing a server connection.

```
1  import java.net.*;
2  import java.io.*;
3
4  public class SimpleServer {
5      public static void main(String args[]) {
6          ServerSocket s = null;
7
8          // Register your service on port 5432
9          try {
10              s = new ServerSocket(5432);
11          } catch (IOException e) {
12              e.printStackTrace();
13          }
14
15          // Run the listen/accept loop forever
16          while (true) {
17              try {
18                  // Wait here and listen for a connection
19                  Socket s1 = s.accept();
20
21                  // Get output stream associated with the socket
22                  OutputStream s1out = s1.getOutputStream();
23                  BufferedWriter bw = new BufferedWriter(
24                      new OutputStreamWriter(s1out));
25
26                  // Send your string!
27                  bw.write("Hello Net World!\n");
28
29                  // Close the connection, but not the server
30                  // socket
31                  bw.close();
32                  s1.close();
33              } catch (IOException e) {
34                  e.printStackTrace();
35              }
36          }
37      }
}
```

Minimal TCP/IP Client

The client side of a TCP/IP application relies on the `Socket` class. Again, much of the work involved in establishing connections is done by the `Socket` class. The client attaches to the server presented in “Minimal TCP/IP Server” on page 16-7, and then prints everything sent by the server to the console.

```
1 import java.net.*;
2 import java.io.*;
3
4 public class SimpleClient {
5     public static void main(String args[]) {
6         try {
7             // Open your connection to a server, at port 5432
8             // localhost used here
9             Socket s1 = new Socket("127.0.0.1", 5432);
10
11            // Get an input stream from the socket
12            InputStream is = s1.getInputStream();
13            // Decorate it with a "data" input stream
14            DataInputStream dis = new DataInputStream(is);
15
16            // Read the input and print it to the screen
17            System.out.println(dis.readUTF());
18
19            // When done, just close the steam and connection
20            tbr.close();
21            s1.close();
22        } catch (ConnectException connExc) {
23            System.err.println("Could not connect.");
24        } catch (IOException e) {
25            // ignore
26        }
27    }
28}
```

Appendix A

Elements of Advanced Java Programming

Objectives

At the end of this appendix, you should be able to:

- Understand two-tier and three-tier architectures for distributed computing
- Understand the role of the Java programming language as a front-end for database applications
- Use the JDBC API
- Understand data interchange methodologies using object brokers
- Explain the JavaBeans Component Model
- Describe and use the javadoc and jar tools

Introduction to Two-Tier and Three-Tier Architectures

Client-server computing involves two or more computers sharing tasks related to a complete application. Ideally, each computer is performing logic appropriate to its design and stated function.

The most widely used form of client-server implementation is a two-tier client-server. This involves a front-end client application communicating with a back-end database engine running on a separate computer. Client programs send structured query language (SQL) statements to the database server. The server returns the appropriate results, and the client is responsible for handling the data.

The basic two-tier client-server model is used for applications that can run with many popular databases including ORACLE®, Sybase, and Informix.

A major performance penalty is paid in two-tier client-server. The client software ends up larger and more complex because most of the logic is handled there. The use of server-side logic is limited to database operations. The client here is referred to as a *thick client*.

Thick clients tend to produce frequent network traffic for remote database access. This works well for intranet and local area networks (LAN)-based network topologies but produces a large footprint on the desktop in terms of disk and memory requirements. Also, not all back-end database servers are the same in terms of server logic offered, and all of them have their own API sets that programmers must use to optimize and scale performance. Three-tier client-server, which is described next, takes care of scalability, performance, and logic partitioning in a more efficient manner.

Three-Tier Architecture

Three-tier is the most advanced type of client-server software architecture. A three-tier client-server demands a much steeper development curve initially, especially when you have to support a number of different platforms and network environments. The payback comes in the form of reduced network traffic, excellent Internet and intranet performance, and more control over system expansion and growth.

Three-Tier Client-Server Definition

Figure A-1 shows a diagram of a generic three-tier architecture.

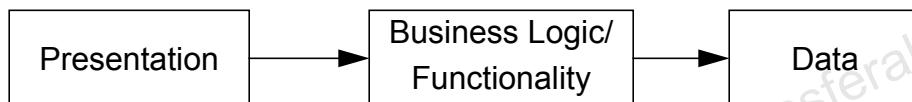


Figure A-1 Generic Three-Tier Architecture

The three components or tiers of a three-tier client-server environment are *presentation*, *business logic* or *functionality*, and *data*. They are separated such that the software for any one of the tiers can be replaced by a different implementation without affecting the other tiers. For example, if you wanted to replace a character-oriented screen (or screens) with a GUI (the presentation tier), you would write the GUI using an established API or interface to access the same functionality programs in the character-oriented screens. The business logic offers functionality in terms of defining all of the business rules through which the data can be manipulated. Changes to business policies can affect this layer without having any impact on the actual databases. The third tier or data tier, includes existing systems, applications, and data that has been encapsulated to take advantage of this architecture with minimal transitional programming effort.

A Database Front End

The Java programming language offers wide benefits to software engineers creating front-end applications for database-oriented systems. With its “Write Once, Run Anywhere™” language feature, the Java programming language offers immediate advantages in terms of its deployment on a wide range of hardware and operating systems. Programmers do not have to write platform-specific code for front-end applications, even in a multi-platform environment.

With Java technology’s rich set of supported front-end development classes, you can interact with databases through the JDBC API. The JDBC API provides a connectivity to back-end databases that can be queried with results being handled by the front end.

In a two-tier model, the database resides on a database server. The client executes a front-end application that opens a socket for communication over the network. The socket provides a communication path between the client application and the back-end server. In the following illustration, client programs send SQL database query requests to the database server. The server returns the results to the client, which formats the results for presentation. This architecture is shown in Figure A-2.

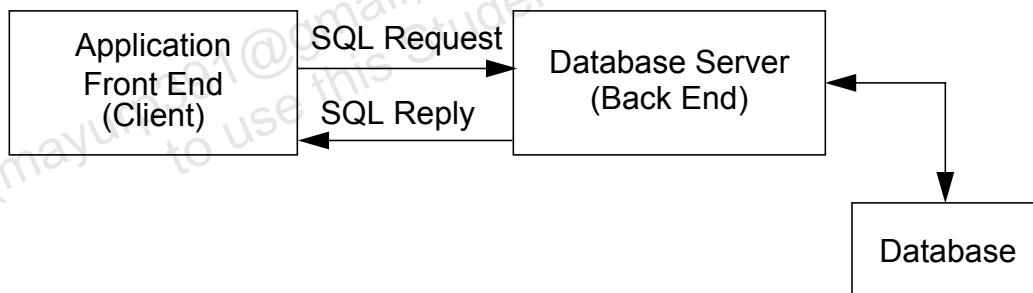


Figure A-2 Database-Centric Architecture

Frequently used mechanisms for data manipulations are often embedded as *stored procedures*. Triggers automatically execute stored procedures when certain conditions are activated during the course of manipulations on the database. The primary drawback of this model is that all business rules are implemented in the client application, creating large client-side runtimes and increased rewriting of the client’s code.

In a three-tier model, the presentation and control logic is embedded in the client (front-end) tier. It communicates with an intermediate server that provides a layer of abstraction from the back-end applications. This middle tier manages the business rules that manipulate the data per the governing conditions of the applications. It can also accept connections from several clients to one or more database servers on a variety of communications protocols. The middle tier provides a database-independent interface for applications and makes the front-end robust. Figure A-3 shows this architecture.

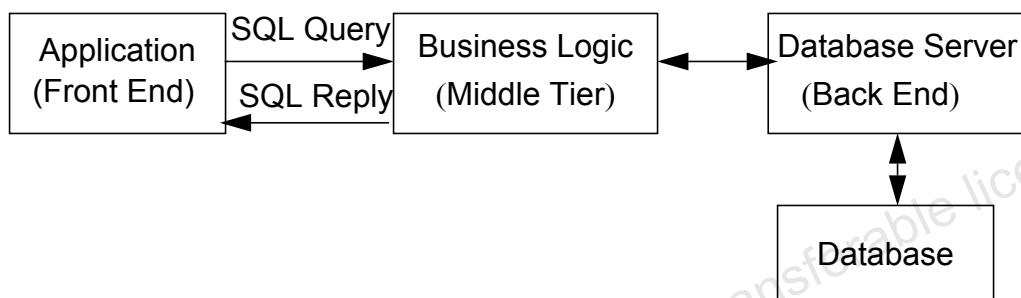


Figure A-3 Application-Centric Architecture

Introduction to the JDBC™ API

The ability to create robust, platform-independent applications and web-based applets prompted developers to create front-end connectivity solutions. JavaSoft™ technology worked with database and database-tool vendors to create a database management system-independent (DBMS-independent) mechanism that would enable developers to write client-side applications that worked with all databases. This effort resulted in the *Java Database Connectivity Application Programming Interface* (JDBC API).

JDBC, An Overview

The JDBC provides a standard interface for accessing a relational database. Modeled after the *open database connectivity* (ODBC) specification, the JDBC package contains a set of classes and methods for issuing SQL statements, table updates, and calls to stored procedures.

Figure A-4 shows a Java programming language front-end application uses the JDBC API to interact with the JDBC Driver Manager. The JDBC Driver Manager uses the JDBC Driver API to load the appropriate JDBC driver. JDBC drivers, which are available from different database vendors, communicate with the underlying DBMS.

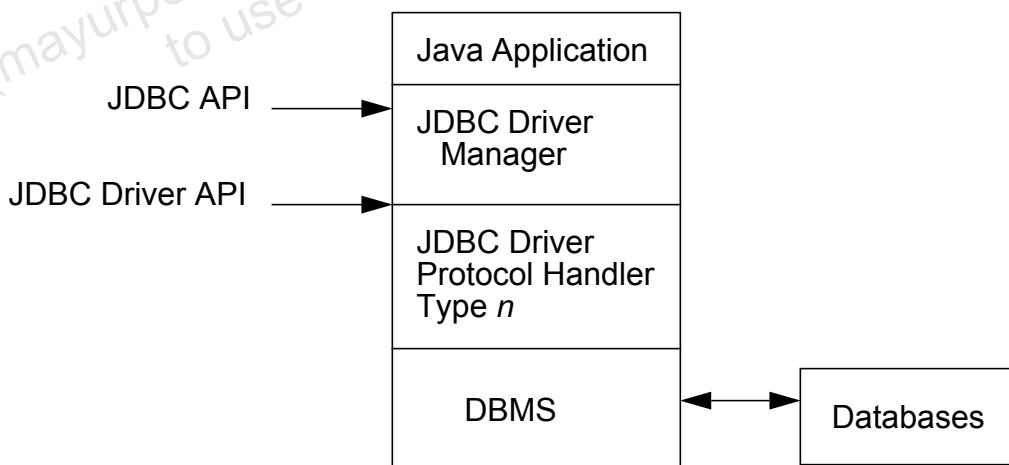


Figure A-4 Layers in a JDBC Application

JDBC Drivers

Java applications use the JDBC API to connect with a database through a database driver. Most database engines have different types of JDBC drivers associated with them. JavaSoft has defined four types of drivers. For more information, refer to
<http://java.sun.com/products/jdbc/jdbc.drivers.html>.

The JDBC-ODBC Bridge

The JDBC-ODBC bridge is a JDBC driver that translates JDBC calls to ODBC operations. This bridge enables all DBMS that support ODBC to interact with Java applications. Figure A-5 shows the layers in a JDBC application that uses the ODBC bridge.

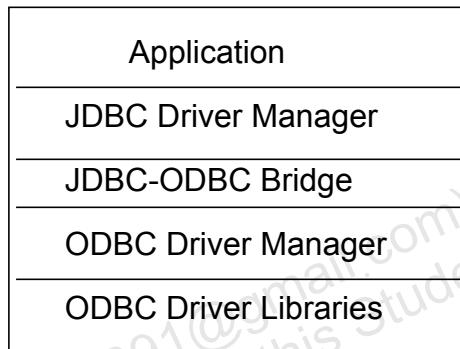


Figure A-5 A JDBC Application Using the ODBC Bridge

The JDBC-ODBC bridge interface is provided as a set of the C-shared dynamic libraries. ODBC provides a client side set of libraries and a driver specific to the client's operating system. These ODBC calls are made as C calls, and the client must have a local copy of the ODBC driver and associated client-side libraries. This places a restriction on its usage in web-based applications.

Distributed Computing

There are Java technologies available for creating distributed computing environments. Two popular technologies are the *remote method invocation* (RMI) and the *common object request broker architecture* (CORBA). RMI is analogous to the *remote procedure call* (RPC) and is preferred by programmers of the Java programming language. CORBA provides flexibility in heterogeneous development environments.

The RMI feature enables a program running on a client computer to make method calls on an object located on a remote server machine. It gives a programmer the ability to distribute computing across a networked environment. Object-oriented design requires that every task be executed by the object most appropriate to that task. RMI takes this concept one step further by allowing a task to be performed on the machine most appropriate to the task. RMI defines a set of remote interfaces that you can use to create remote objects. A client can invoke methods of a remote object with the same syntax that it uses to invoke methods on a local object. The RMI API provides classes that handle all of the underlying communication and parameter referencing requirements of accessing remote methods.

With all of the distributed computing architectures, an application process or *object server (daemon)* advertises itself to the world by registering with a naming service on the local machine (node). In the case of RMI, a naming service daemon called the RMI registry runs over an RMI port that by default listens over IP Port 1099 on that host. The RMI registry contains an internal table of remote object references. For each remote object, the table contains a registry name and a reference to that object. You can store multiple instances for the same object by instantiating and binding it multiple times to the registry, using different names.

RMI

When an RMI client binds a remote object through the registry, it receives a local reference to the remote instantiated object through its interface and communicates with the object through that reference. Local references to the same remote object can exist on multiple clients; any variables and methods contained within the remote object are shared.

The applet begins by importing the appropriate RMI packages and creating a reference to the remote object. After the applet establishes this link, it can call the remote object's methods as if they were locally available to the applet.

RMI Architecture

The RMI architecture provides three layers: Transport layers, Remote Reference layer, and Stubs/Skeleton layer. Figure A-6 shows these layers.

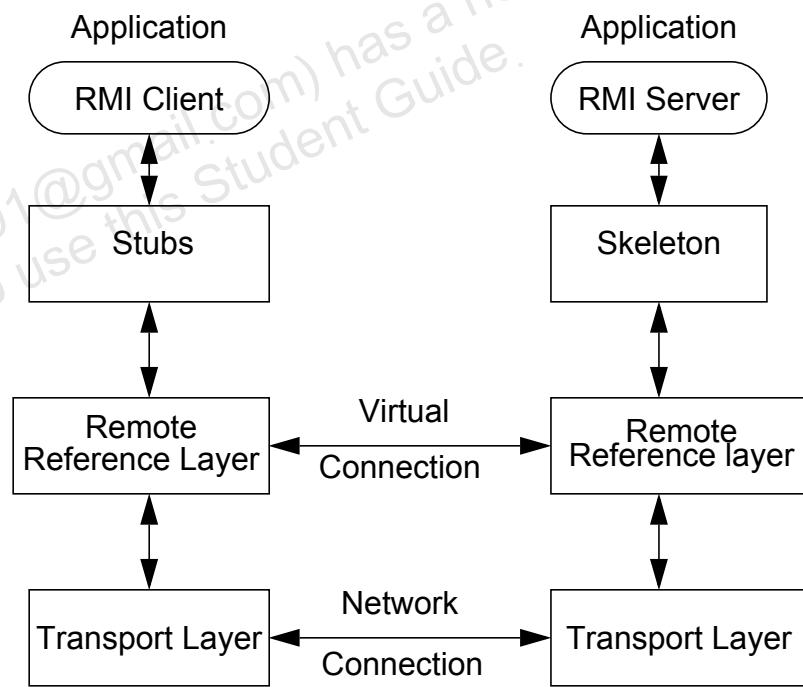


Figure A-6 Layers of an RMI Architecture

The Transport layer creates and maintains physical connections between the client and server. It handles the data stream passing through the Remote/Reference layers (RRLs) on the client and server side.

The Remote Reference layer provides an independent reference protocol for establishing a virtual network between the client and server. It establishes interfaces to the lower Transport layer and the upper Stub/Skeleton layer.

A Stub is a client-side proxy representing the remote object. The client interacts with the Stub through interfaces. The Stub appears as a local object to the client. The Skeleton on the server side acts as an interface between the RRL and the object implemented on the server side.

Creating an RMI Application

This section guides you through the steps for creating, compiling, and running an RMI application. The following steps illustrate the process:

1. Define interfaces for remote classes.
2. Create and compile implementation classes for the remote classes.
3. Create stub and skeleton classes using the `rmiic` command.
4. Create and compile the server application.
5. Start the RMI Registry and the server application.
6. Create and compile a client program to access the remote objects.
7. Test the client.

CORBA

CORBA is a *specification* that defines how distributed objects can interoperate. The CORBA specification is controlled by the Object Management Group (OMG), an open consortium of more than 700 companies that work together to define open standards for distributed computing. For more information, refer to the following URL:

<http://www.omg.org>

You can write CORBA objects in almost any programming language, including C and C++. These objects can also exist on almost any platform, including the Solaris OS, Microsoft Windows, openVMS, Digital UNIX, HP-UX, and many others. This means a Java application running on a Microsoft Windows platform can interact over a network with C++ objects on a UNIX system.

Language independence is possible using the construction of interfaces to objects using the *Interface Definition Language* (IDL). IDL allows all CORBA objects to be described in the same manner; the only requirement is a *bridge* between the native language (C/C++, COBOL, Java) and IDL.

At the core of CORBA is the *object request broker* (ORB). The ORB is the principal component for the transmission of information between the client and the server of the CORBA application. The ORB manages marshalling requests, establishes a connection to the server, sends the data, and executes the requests on the server side. The same process occurs when the server wants to return the results of the operation. ORBs from different vendors communicate over a network (often, but not necessarily using TCP/IP) using the *Internet Inter ORB Protocol (IIOP)*, which is a part of the CORBA 2.0 standard.

The Java IDL

Java IDL adds CORBA capability to the Java programming language, providing standards-based interoperability and connectivity. Java IDL enables distributed Java applications to transparently invoke operations on remote network services, using the industry standard IDL and IIOP.

Java IDL is not an *implementation* of OMG's IDL. It is, in fact, a CORBA ORB that uses IDL to define interfaces. The `idltojava` compiler generates portable client stubs and server skeletons. The CORBA client interacts with another object running on a remote server by accessing a reference object through its *naming service*. Like the RMI Registry, the naming service is an application that runs as a background process on a remote server. It holds a table of named services and remote object references used to resolve client requests.

The steps involved in setting up a CORBA object can be summarized as follows:

1. Create the object's interface using the IDL.
2. Convert the interface into stub and skeleton objects using the `javatoidl` compiler.
3. Implement the skeleton object, creating the CORBA server object.
4. Compile and execute the server object, binding it to the naming service.
5. Create and compile a client object, which invokes the methods within the server object.
6. Execute the client object, accessing the server object through the CORBA naming service.

RMI Compared With CORBA

RMI's biggest advantage stems from the fact that it is fully object oriented. By contrast, CORBA provides a largely-procedural mechanism for connecting distributed objects. Consider the command pattern. This pattern provides excellent flexibility and maintainability, but cannot be implemented properly between two CORBA systems because it requires that objects (both state and *behavior*) be moved from the client to the server. RMI can do this as a direct consequence of the platform independent bytecode.

One of the key benefits often cited for CORBA is its language independence. In fact, RMI can provide this too by using the Java Native Interface.

CORBA services are available for a significant variety of both vertical and horizontal problems, these services are often well-understood and mature. Examples of potentially important features of services are transactions and security.

CORBA's language independence adds significant complexity to the development cycle and precludes garbage collection features that RMI supports.

In many cases, the choice between RMI and CORBA is most likely to be made largely on political and environmental grounds, rather than on purely technical ones. A company that already has CORBA would need a compelling reason to introduce a new technology, especially if the change would render a substantial investment in services redundant. New systems, however, are likely to benefit from the use of RMI, provided that higher management do not perceive this as an immature technology.

The JavaBeans™ Component Model

The JavaBeans architecture is an integration technology, a component framework that allows reusable component objects (called *beans*) to communicate with one another and with the framework.

A Java bean is an independent and reusable software component that you can manipulate visually in a builder tool. Beans can be visible objects, such as AWT components, or invisible objects, such as queues and stacks. A builder or integration tool manipulates beans to create applets and applications. The component model specified by the JavaBeans 1.00-A specification defines five major services:

- Introspection – This process exposes the properties, methods, and events that a JavaBean component supports. It is used at runtime while the bean is being created with a visual development tool.
- Communication – This event-handling mechanism creates an event that serves as a message to other components.
- Persistence – Persistence is a means of storing the state of a component. The simplest way to support persistence is to take advantage of Java object serialization, but it is up to the individual browsers or the applications that use the bean to actually save the state of the bean.
- Properties – Properties are attributes of a bean that are referenced by name. These properties are usually read and written by calling methods on the bean created specifically for that purpose. Some property types affect neighboring beans as well as the one in which the property originates.
- Customization – One of the primary characteristics of a bean is its reusability. The beans framework provides several ways of customizing existing beans into new ones.

Bean Architecture

A bean is represented by an interface that is seen by the users. The environment must connect to this interface, if it wants to interact with this bean. Beans consist of three general-purpose interfaces: events, properties, and methods. Because beans rely on their state, they must be persistent over time.

Events

Bean events are the mechanism for sending asynchronous messages between beans, and between beans and containers. A bean uses an event to notify another bean to take an action or to inform the bean that a state change has occurred. An event allows your beans to communicate when something interesting happens; to do this, they make use of the event model introduced in JDK version 1.1. The event model used in Java 2 SDK is the same event model that was introduced in JDK version 1.1. There are three parts to this communication: event object, event listener, and event source.

The JavaBeans architecture communicates primarily using event listener interfaces that extend `EventListener`.

Bean developers can design their own event types and event listener interfaces and make their beans act as a source by implementing the `addXXXListener(EventObject e)` and `removeXXXListener(EventObject e)` methods, where `XXX` is the name of the event type. Then, the developers can make other beans act as event targets by implementing the `XXXListener` interface. The `sourceBean` and the `targetBean` are brought together by calling `sourceBean.addXXXListener(targetBean)`.

Properties

Properties define the characteristics of the bean. They can be changed at runtime through their `get` and `set` methods.

You can use properties to send two-way synchronous communications between beans. Beans also support asynchronous property changes between beans using special event communication.

Methods

Methods are operations through which you can interact with a bean. Beans receive notification of events by having the appropriate event source method call them. Some methods are special and deal with properties and events. These methods must follow special naming conventions outlined in the beans specification. Other methods might be unrelated to an event or property. All public methods of a bean are accessible to the beans framework and can be used to connect a bean to other beans.

Bean Introspection

The JavaBean introspection process exposes the properties, methods, and events of a bean. Bean classes are assumed to have properties if there are methods that either set or get a property type.

The `BeanInfo` interface, provided by the JavaBeans API, enables bean designers to expose properties, events, methods, and any global information about a bean. The `BeanInfo` interface provides a series of methods to access bean information, but a bean developer can also include private description files that the `BeanInfo` class uses to define bean information. By default, a `BeanInfo` object is created when introspection is run on the bean (Figure A-7).

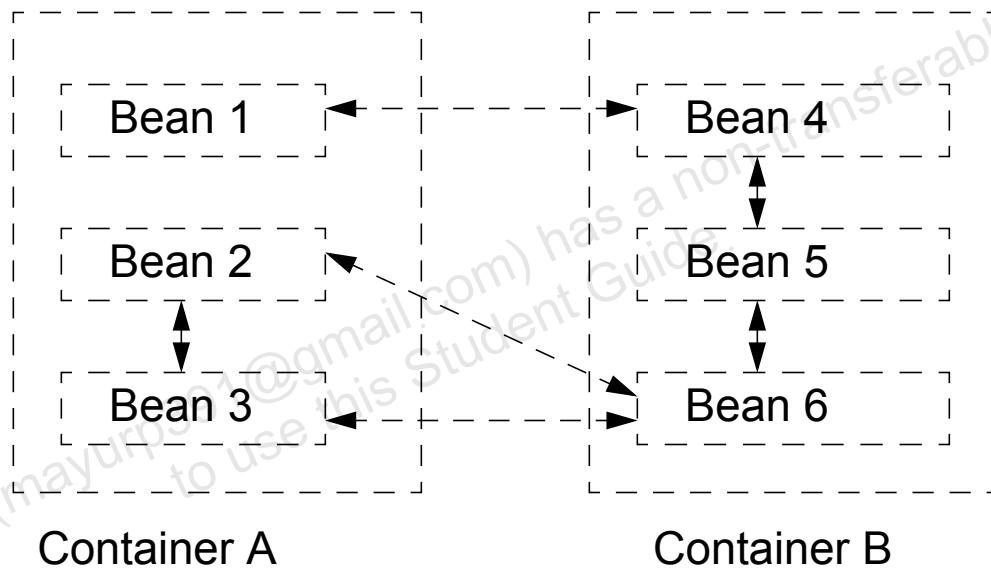


Figure A-7 A Sample Bean Interaction

A Sample Bean Interaction

In Figure A-7 on page A-16, Container A and Container B contain six beans. A bean can interact with other beans that are present in the same container and with beans that are in a separate container. In this example, Bean 1 interacts with Bean 4. It does not communicate with Bean 2 and Bean 3, which reside in the same container. This illustrates the point that a bean can communicate with any other bean and is not restricted to communicating with a bean in the same container. However, Bean 4 communicates with Bean 5, which resides in the same container. Source Bean 1 sends an event to the target Bean 4, which causes it to listen for messages on its event listener. All other inter-container and intra-container bean interactions can be explained in a similar manner.

The Beans Development Kit (BDK)

The BDK is a Java application developed by JavaSoft that enables Java technology developers to create reusable components that use the bean model. It is a complete system that contains source code for all examples and documentation. The BDK comes with a sample bean builder and customizer application called BeanBox. The BeanBox is a test container that you can use to do the following:

- Resize and move beans
- Alter beans with property sheets
- Customize beans with a customizer application
- Connect beans together
- Drop beans onto a composition window
- Save beans through serialization
- Restore beans

The BDK comes with a set of 12 sample beans, which cover all of the aspects of the JavaBeans API.

JAR Files

JAR (Java archive) is a platform-independent file format that aggregates many files into one. You can bundle multiple Java applets and their requisite components (.class files, images, and sounds) in a JAR file and subsequently download to a browser in a single Hypertext Transfer Protocol (HTTP) transaction, greatly improving the download speed. The JAR format also supports compression, which reduces the file size, further improving the download time. In addition, the applet author can digitally sign individual entries in a JAR file to authenticate their origin. It is fully backward-compatible with existing applet code and can be extended.

Changing the `applet` tag in your HTML page to accommodate a JAR file is easy. The JAR file on the server is identified by the `archive` parameter, which contains the directory location of the JAR file relative to the location of the HTML page; for example:

```
<applet code="Animator.class"
        archive="jars/animator.jar"
        width="460" height="160" >
    <param name="foo" value="bar">
</applet>
```

Using the javadoc Tool

Documentation of your code is important to the success of future maintenance efforts for standalone applications and is critical to the use of APIs. In this section, the javadoc tool, comment tags, and how to use the tool are briefly described.

Hopefully you already have some experience in using the Java 2 SDK documentation. Look at how you can generate documentation HTML pages for your projects.

```
> javadoc -private -d .../doc/api banking domain banking.reports
```

Typically if you are generating documentation for an API, you would use the `-public` option (or leave it out because it is the default). However, it is common to use the `-private` option to generate documentation that is shared among an application project team.

Note – Read the online documentation for more details.



Documentation Tags

The javadoc tool parses the specified source files for comment lines that start with `/**` and end with `*/`. The tool uses these to document the declaration that the comment immediately precedes.

The first sentence of the comment is called the *summary sentence*, and it should be a complete, concise description of the declaration. You can use the text following the summary sentence to give details about the declaration, including usage information. HTML tags can be included in any portion of the text, such as using the `<P>` tag to separate paragraphs, `` to generate lists, `` (and so on) to format the text.

Also within the comment block, javadoc uses tags to identify special elements of the declaration, such as the return value of a method. The table in the preceding overhead shows a set of the most common javadoc tags, their meaning, and with which declarations they may be used.

Using the javadoc Tool

Example

The following example is used in the tests of javadoc:

```
1  /*
2   * This is an example using javadoc tags.
3   */
4
5  package mypack;
6
7  import java.util.List;
8
9 /**
10  * This class contains a bunch of documentation tags.
11  * @author Bryan Basham
12  * @version 0.5(beta)
13  */
14 public class DocExample {
15
16     /** A simple attribute tag. */
17     private int x;
18
19 /**
20  * This variable a list of stuff.
21  * @see #getStuff()
22  */
23     private List stuff;
24
25 /**
26  * This constructor initializes the x attribute.
27  * @param x_value the value of x
28  */
29     public DocExample(int x_value) {
30         this.x = x_value;
31     }
32
33 /**
34  * This method return some stuff.
35  * @throws IllegalStateException if no stuff is found
36  * @return List the list of stuff
37  */
38     public List getStuff()
39             throws IllegalStateException {
40         if ( stuff == null ) {
41             throw new java.lang.IllegalStateException("ugh, no stuff");
42     }
```

```

43     return stuff;
44   }
45 }
```

The following command results in the display shown in Figure A-8:

```
> javadoc -d doc/api/public DocExample.java
```

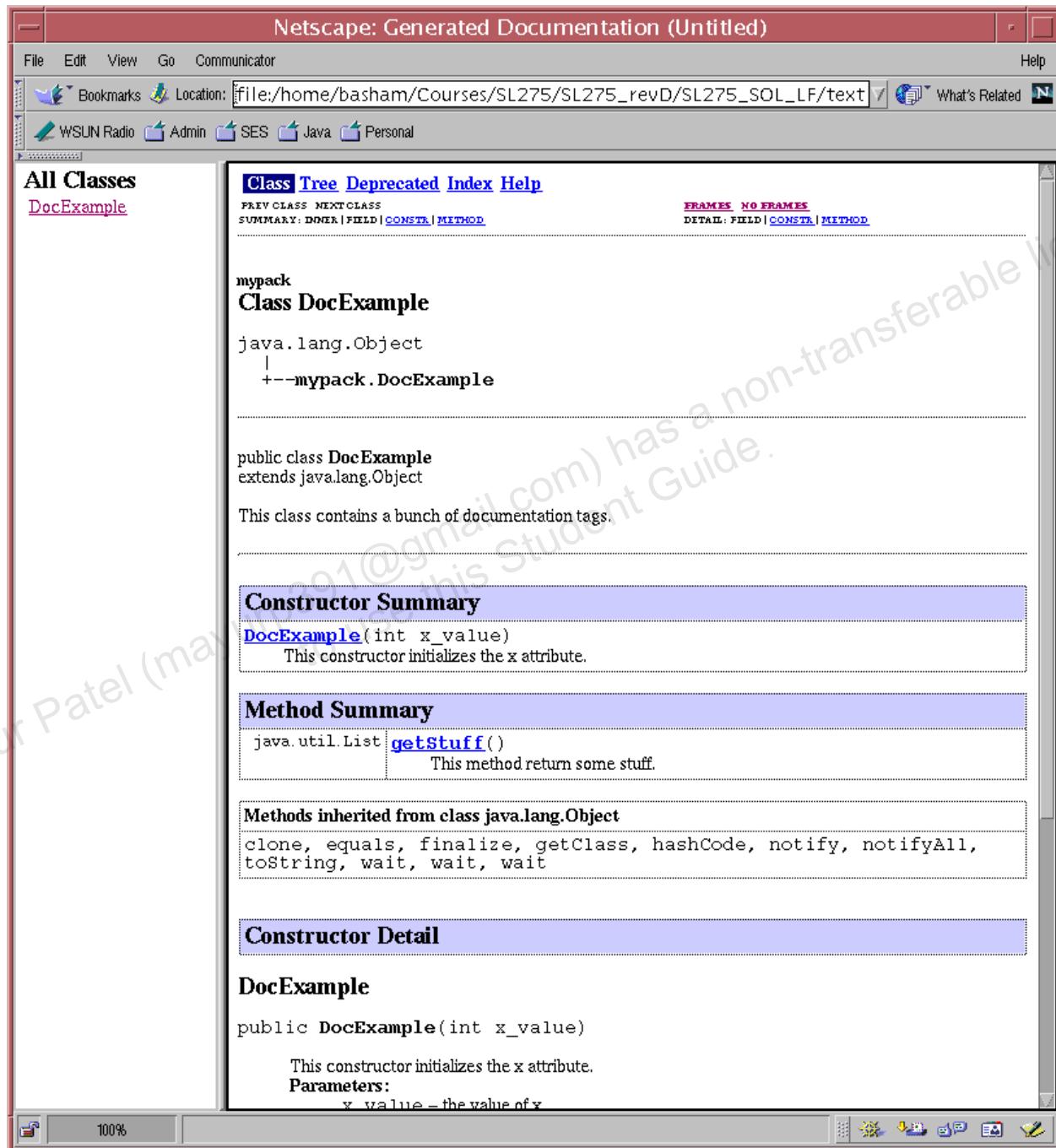


Figure A-8 Public Example

Using the javadoc Tool

The following command results in the display shown in Figure A-9:

```
> javadoc -private -d doc/api/private DocExample.java
```

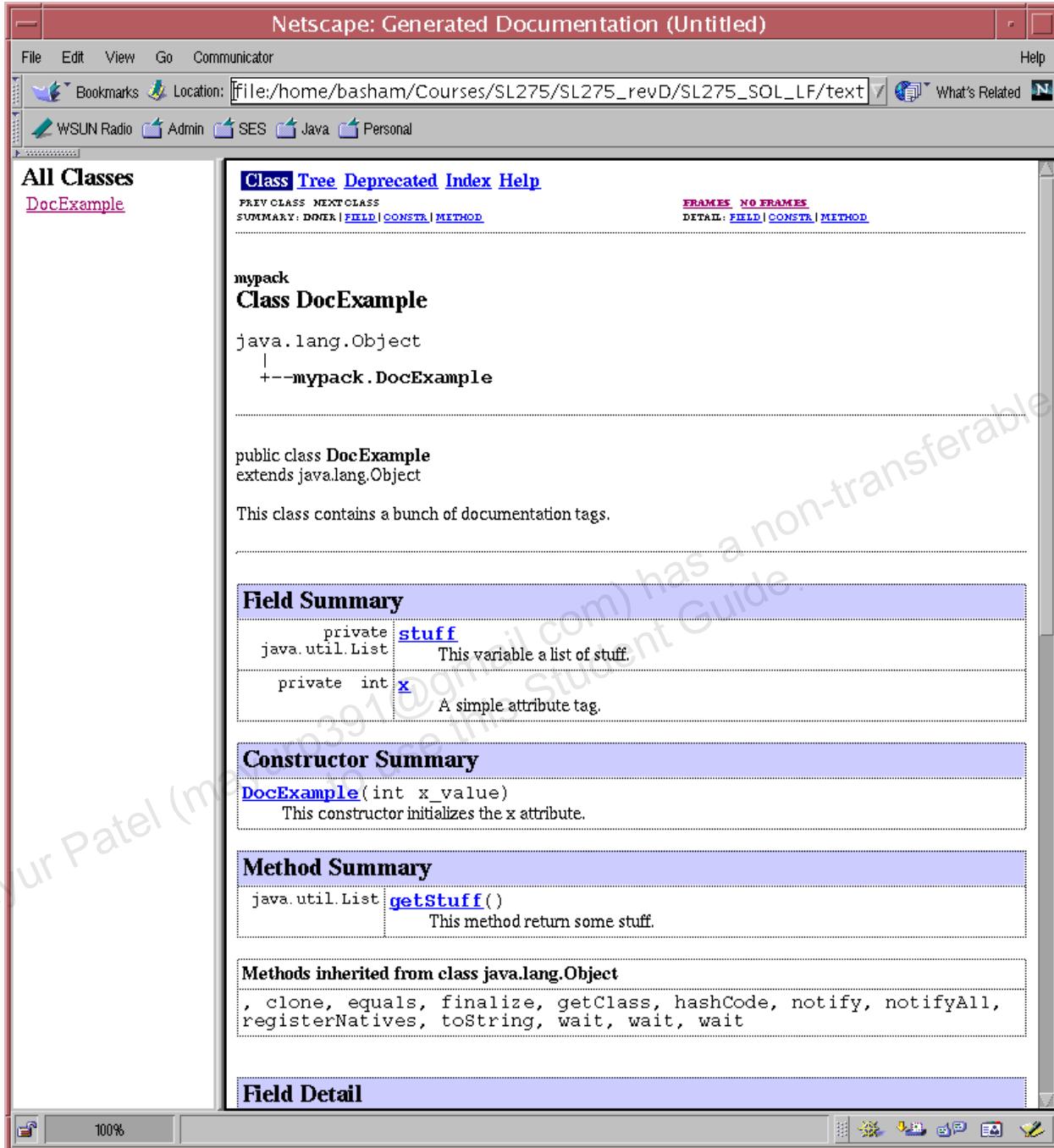


Figure A-9 Private Example

Appendix B

Quick Reference for UML

Objectives

Upon completion of this appendix, you should be able to:

- Understand the fundamentals of UML
- Describe general UML elements
- Describe use case diagrams
- Describe class diagrams
- Describe object diagrams
- Describe collaboration diagrams
- Describe sequence diagrams
- Describe statechart diagrams
- Describe activity diagrams
- Describe component diagrams
- Describe deployment diagrams

Additional Resources



Additional resources – The following references provide additional details on the topics described in this appendix:

- Booch, Grady, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Reading: Addison Wesley Longman, Inc., 1999.
- Folwer, Martin, with Kendall Scott. *UML Distilled (2nd ed)*. Reading: Addison Wesley Longman, Inc., 2000.
- The Object Management Group. “OMG Unified Modeling Language Specification,” version 1.4, September 2001, [<http://www.omg.org/technology/documents/formal/uml.htm>], accessed 22 April 2004.

Note – Additional UML resources are available online at:
<http://www.omg.org/uml/>.



UML Fundamentals

Unified Modeling Language (UML) is a graphical language for modeling software systems. UML is not a programming language, it is a set of diagrams that can be used to specify, construct, visualize, and document software designs. Software engineers use UML diagrams to construct and explain their software designs, similar to how a building architect uses blueprints to construct and explain their building designs. UML has diagrams to assist in every part of the application development process from requirements gathering through design, and into coding, testing, and deployment.

UML was developed in the early 1990s by three leaders in the object modeling world: Grady Booch, James Rumbaugh, and Ivar Jacobson. Their goal was to unify the major methods that they had previously developed to create a new standard for software modelling. UML is now the most commonly used modeling language. UML is currently maintained by the OMG. The UML specification is available on the OMG web site at <http://www.omg.org/uml/>.

The UML is not a process for how to do analysis and design. UML is only a set of tools to use in a process. UML is frequently used with a process such as the Unified Software Development Process. Sun Microsystems OO-226: *Object Oriented Analysis and Design Using UML*, is a five day course that teaches effective methods of analysis and design using UML language, the Unified Software Development Process (USDP) method, and software patterns.

UML Fundamentals

UML defines nine standard types of diagrams. Table B-1 provides a list of these diagrams, an informal description, and use.

Table B-1 Types of UML Diagrams

Diagram Name	Description	Use
Use Case	A Use Case diagram is a simple diagram that shows who is using your system and what processes they will perform in the system.	Use the Use Case diagram to document the Requirements Gathering and Analysis workflows. Throughout the entire development, all work should be traceable back to the Use Case diagram.
Class	A Class diagram shows a set of classes in the system, and the associations and inheritance relationships between the classes. Class nodes might also contain a listing of attributes and operations.	Use the Class diagram for showing the structure of the system and what needs to be programmed. Most UML case tools can generate code based on the Class diagram.
Object	An Object diagram shows specific object instances and the links between them. An Object diagram represents a snapshot of the system objects at a specific point in time.	Use the Object diagram to clarify or validate the Class diagram.
Activity	An Activity diagram is essentially a flow chart with new symbols. This diagram represents the flow of activities in a process or algorithm.	Use the activity diagram to model real world business systems during the Requirements Gathering workflow.
Collaboration	The Collaboration diagram and the Sequence diagram both show processes from an object oriented perspective. The main difference is that the layout of the Collaboration diagram puts more focus on the objects, rather than the sequence.	Use the collaboration diagram to focus on the objects in a sequence. Collaboration diagrams are typically more difficult to read than Sequence diagrams.

Table B-1 Types of UML Diagrams (Continued)

Diagram Name	Description	Use
Sequence	A Sequence diagram shows a process from an object oriented perspective by showing how a process is executed by a set of objects or actors.	Use the Sequence diagram for assigning responsibilities to classes by considering how they can work together to implement the processes in the system. This is essential.
Statechart	A Statechart diagram shows how a particular object changes behavioral state as various events happen to it.	Use the Statechart diagram to understand objects that change behavioral states in significant ways.
Component	A Component diagram shows the major software components of a system and how they can be integrated. Component diagrams can contain non-OO software components, such as legacy procedural code and web documents.	Use Component diagrams to show how all of the OO and non-OO components fit together in your system. It is also a good way to look at the high-level software structure of your system.
Deployment	A Deployment diagram shows the hardware nodes in the system.	Use the Deployment diagram to show how a distributed system will be configured. Software components might be displayed inside the hardware nodes to show how they will be deployed.

General Elements

General Elements

In general, UML diagrams represent concepts, depicted as symbols (also called nodes), and relationships among concepts, depicted as paths (also called links), connecting the symbols. These nodes and links are specialized for the particular diagram. For example, in Class diagrams, the nodes represent object classes, and the links represent associations between classes and generalization (inheritance) relationships.

There are other elements that augment these diagrams, including: packages, stereotypes, annotations, constraints, and tagged values.

Packages

In UML, packages enable you to arrange your modeling elements into groups. UML packages are a generic grouping mechanism and should not be directly associated with Java technology packages. However, you can use UML packages to model Java technology packages. Figure B-1 shows a package diagram that contains a group of classes in a Class diagram.

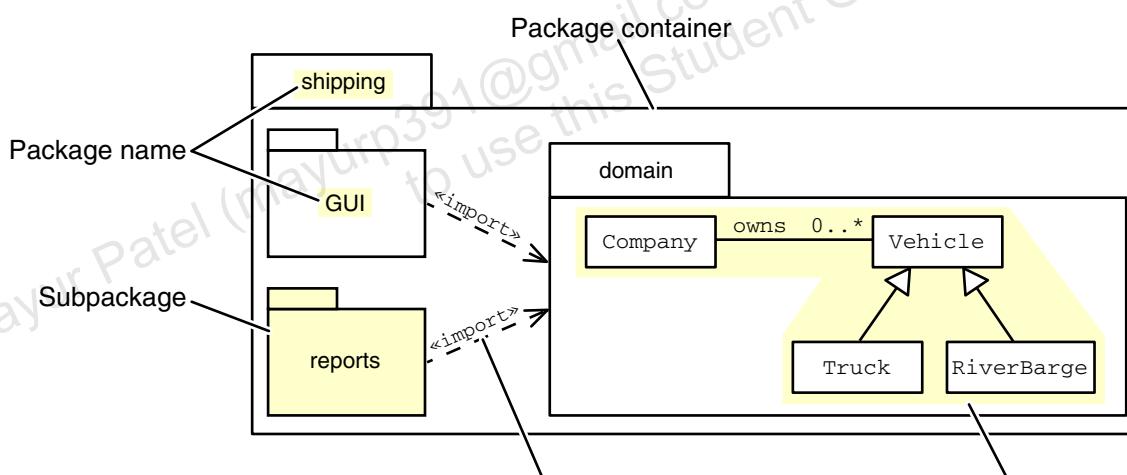


Figure B-1 Example Package

Mapping to Java Technology Packages

The mapping of UML packages to Java technology packages implies that the classes would contain the package declaration of package `shipping.domain`. For example, in the file `Vehicle.java`:

```
package shipping.domain;

public class Vehicle {
    // declarations
}
```

Figure B-1 on page B-6 also shows a simple hierarchy of packages. The `shipping` package contains the `GUI`, `reports`, and `domain` subpackages. The dashed arrow from one package to another indicates that the package at the tail of the arrow uses (`imports`) elements in the package at the head of the arrow. For example, `reports` uses elements in the `domain` package as follows:

```
package shipping.reports;

import shipping.domain.*;

public class VehicleCapacityReport {
    // declarations
}
```

Note – In Figure B-1 on page B-6, the `shipping.GUI` and `shipping.reports` packages have their names in the body of the package box, rather than in the head of the package box. This is done only when the diagram does not expose any of the elements in that package.



General Elements

Stereotypes

The designers of UML understood that they could not build a modeling language that would satisfy every programming language and every modeling need. They built several mechanisms into UML to enable modelers to create their own semantics for model elements (nodes and links). Figure B-2 shows the use of a stereotype tag «interface» to declare that the class node Set is a Java technology interface declaration. Stereotype tags can adorn relationships as well as nodes. There are over a hundred standard stereotypes. You can create your own stereotypes to model your own semantics.

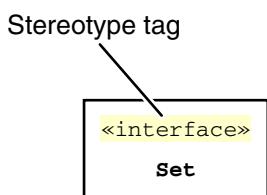


Figure B-2 Example Stereotype

Annotation

The designers of UML also built a method for annotating the diagrams into the UML language. Figure B-3 shows a simple annotation.

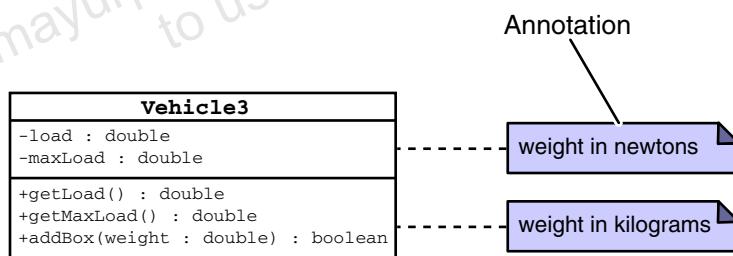


Figure B-3 Example Annotation

Annotations can contain notes about the diagram as a whole, notes about a particular node or link, or even notes about an element within a node. The dotted link from the annotation points to the element being annotated. If there is no link from the annotation, then the note is about the whole diagram.

Constraints

Constraints enable you to model certain conditions that apply to a node or link. Figure B-4 shows several constraints. The topmost constraint specifies that the Player objects must be stored in a persistent database. The middle constraint specifies that the captain and co-captain must also be members of the team's roster. The constraint on the bottom specifies the minimum number of players by gender.

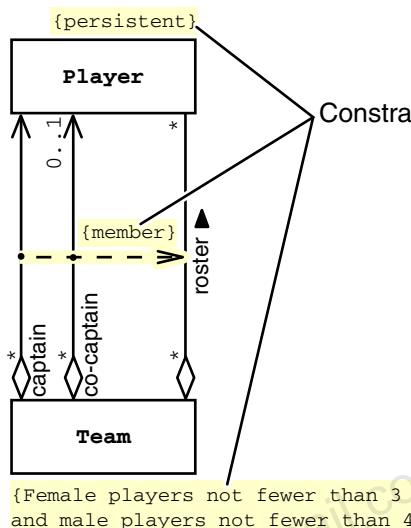


Figure B-4 Example Constraints

Tagged Values

Figure B-5 shows several examples of how tagged values enable you to add new properties to nodes in a diagram.

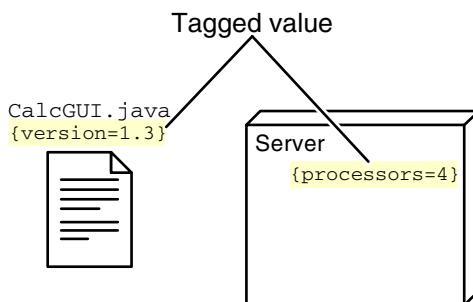


Figure B-5 Example Tagged Values

Use Case Diagrams

Use Case Diagrams

A Use Case diagram represents the functionality provided by the system to external users. The Use Case diagram is composed of actors, use case nodes, and their relationships. Actors can be humans or other systems.

Figure B-6 shows a simple banking Use Case diagram. An actor node can be denoted as a *stick figure* (as in the three Customer actors) or as a class node (see “Class Nodes” on page B-11) with the stereotype of «actor». There can be a hierarchy of actors.

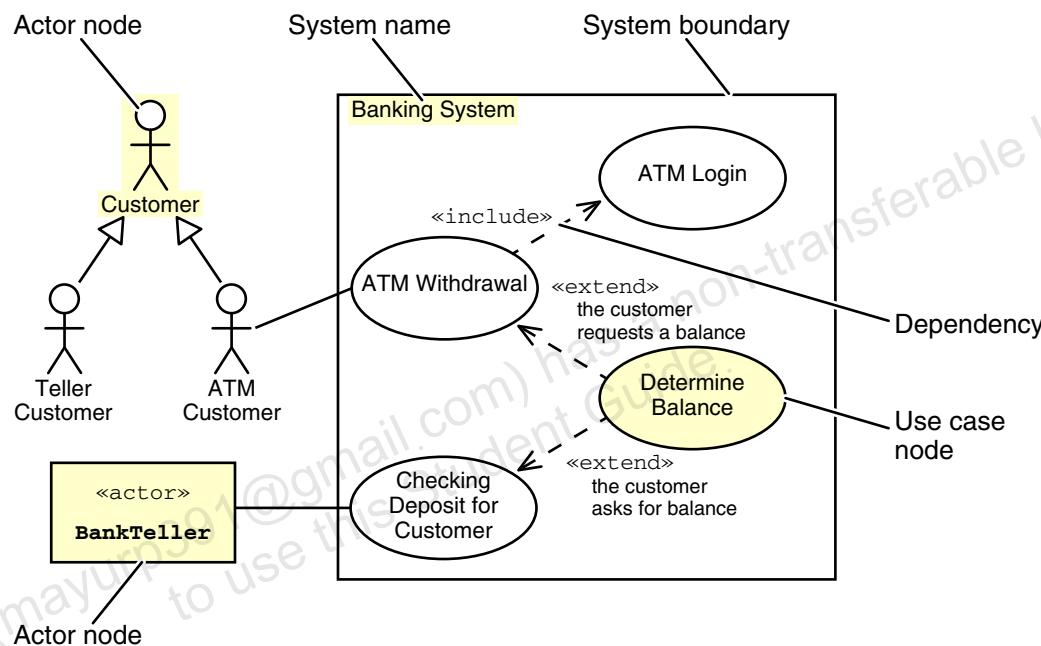


Figure B-6 Use Case Diagram Example

A use case node is denoted by a labeled oval. The label indicates the activity summary that the system performs for the actor. Use case nodes are grouped into a system box, which is usually labeled in the top left corner. The relationship *actor uses the system to* is represented by the solid line from the actor to the use case node.

Use case nodes can depend on other use cases. For example, the ATM Withdrawal use case uses the ATM Login use case.

Use case nodes can also extend other use cases to provide optional functionality. For example, you can use the Determine Balance use case to extend the Checking Deposit for Customer use case.

Class Diagrams

A Class diagram represents the static structure of a system. These diagrams are composed of classes and their relationships.

Class Nodes

Figure B-7 shows several *class nodes*. You do not have to model every aspect of a class every time that class is used in a diagram.

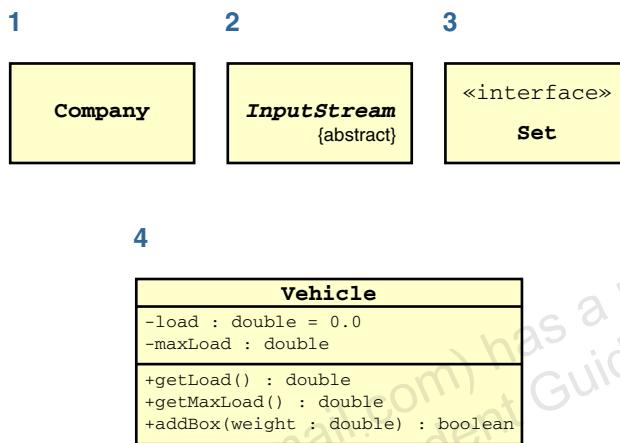


Figure B-7 Several Class Nodes

A class node can be only the name of the class, as in Examples 1, 2, and 3 of Figure B-7. Example 1 is a concrete class, where no members are modeled. Example 2 is an abstract class (name is in italics). Example 3 is an interface. Example 4 is a concrete class, where members are modeled.

Class Diagrams

Figure B-8 shows the element of a class node.

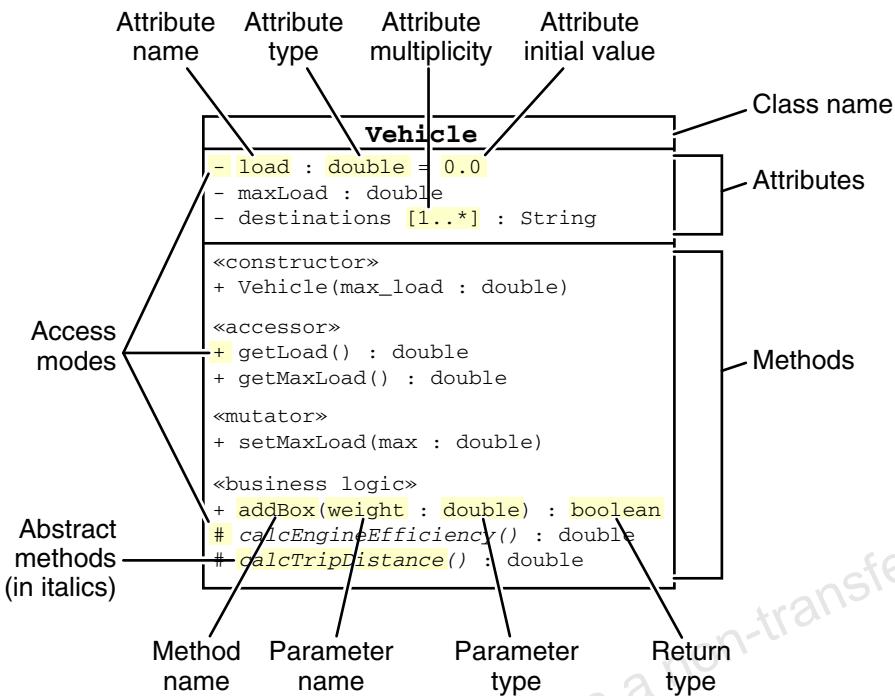


Figure B-8 Class Node Elements

A fully specified class node has three fundamental compartments:

- The name of the class in the top is one compartment.
- The set of attributes under the first bar is a second compartment.

An attribute is specified by five elements, including: access mode, name, multiplicity, data type, and initial value. All of the attribute elements are optional, except for the name element.

- The set of methods under the second bar is another compartment.

A method is specified by four elements, including: access mode, name, parameter list (a comma-delimited list of parameter name and type), and the return type. All elements of a method are optional, except for the name element. If the return value is not specified, then no value is returned (`void`). The name of an abstract method is marked in italics.

You can use stereotypes to group attributes or methods together. For example, you can separate accessor, mutator, and business logic methods from each other for clarity. And because there is no UML-specific notation for constructors, you can use the `<constructor>` stereotype to mark the constructors in your method compartment.

Table B-2 shows the valid UML access mode symbols.

Table B-2 UML Defined Access Modes and Their Symbols

Access Mode	Symbol
Private	-
Package private	~
Protected	#
Public	+

Figure B-9 shows an example class node with elements that have class (or static) scope. This is denoted by the underline under the element. For example, counter is a static data attribute and getTotalCount is a static method.

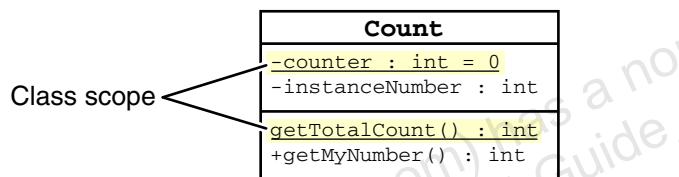


Figure B-9 Class Node With Static Elements Example

You could write the Count class in the Java programming language as:

```

public class Count {
    private static int counter = 0;
    private int instanceNumber;

    public static int getTotalCount() {
        return counter;
    }
    public int getMyNumber() {
        return instanceNumber;
    }
}

```

Class Diagrams

Inheritance

Figure B-10 shows class inheritance through the generalization relationship arrow.

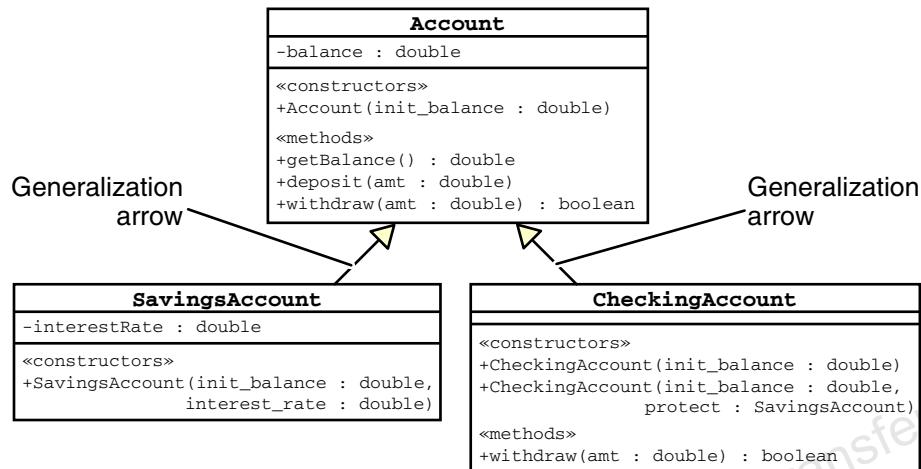


Figure B-10 Class Inheritance Relationship

Class inheritance is implemented in the Java programming language with the `extends` keyword. For example:

```

public class Account {
    // members
}

public class SavingsAccount extends Account {
    // members
}

public class CheckingAccount extends Account {
    // members
}
  
```

Interface Implementation

Figure B-11 shows how to use the Realization arrow to model a class that is implementing an interface.

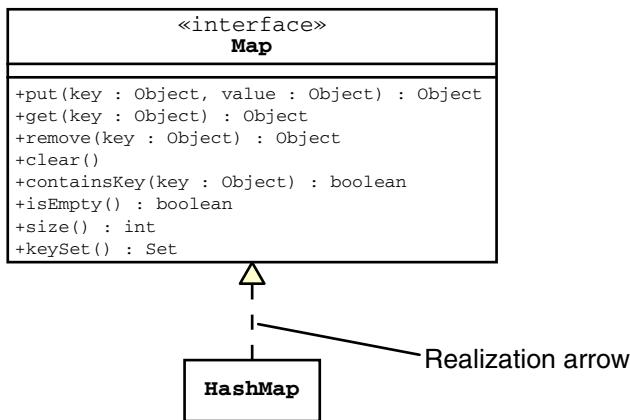


Figure B-11 Class Implementing an Interface Example

An interface is implemented in the Java programming language with the `implements` keyword. For example:

```
public interface Map {  
    // declaration here  
}  
  
public class HashMap implements Map {  
    // definitions here  
}
```

Association, Roles, and Multiplicity

Figure B-12 shows an example association. An **association** is a link between two types of objects, and implies the ability for the code to navigate from one object to another.

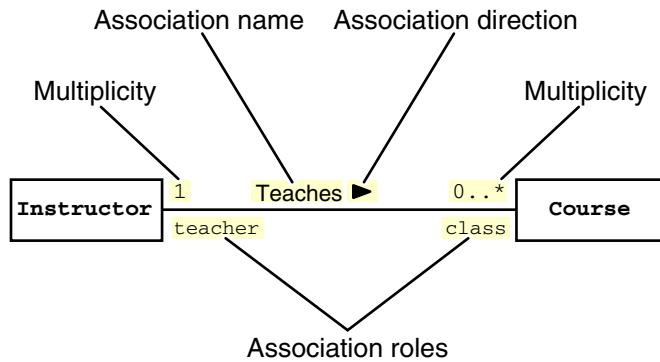


Figure B-12 Class Associations and Roles

In this diagram, **Teaches** is the name of the association with a directional arrow pointing to the right. This association can be read as *an instructor teaches a course*. You can also attach roles to each end of the association. In Figure B-12, the **teacher** role indicates that the instructor is the teacher for a given course. All of these elements are optional if the association is obvious.

This example also demonstrates how many objects are involved in each role of the association. This is called **multiplicity**. In this example, there is only one teacher for every class, which is denoted by the **1** on the **Instructor** side of the association. Also, any given teacher might teach zero or more courses, which is denoted by the **0..*** on the **Course** side. You can leave out the multiplicity for a given role if it is always one. You can also abbreviate zero or more (**0..***) using only an asterisk (*).

You can express multiplicity values as follows:

- A range of values – For example, **2..10** means at least 2 and up to 10
- A disjoint set of values – For example, **2,4,6,8,10**
- A disjoint set of values or ranges – For example, **1..3,6,8..11**

However, the most common values for multiplicity are exactly one (1 or left blank), zero or one (**0..1**), zero or more (*), or one or more (**1..***).

Associations are typically represented in the Java programming language as an attribute in the class at the tail of the relationship (specified by the direction indicator). If the multiplicity is greater than one, then some sort of collection or array is necessary to hold the elements.

Also, in Figure B-12 on page B-16 the association between an instructor and a course might be represented in the Instructor class as:

```
public class Instructor {  
    private Course[] classes = new Course[MAXIMUM];  
}
```

or as:

```
public class Instructor {  
    private List classes = new ArrayList();  
}
```

The latter representation is preferable if you do not know the maximum number of courses any given instructor might teach.

Aggregation and Composition

An *aggregation* is an association in which one object contains a group of parts that make up the *whole* object (see Figure B-13). For example, a car is an aggregation of an engine, wheels, body, and frame. Composition is a specialization of aggregation in which the parts cannot exist independently of the whole object. For example, a human is a composition of a head, two arms, two legs, and a torso. If you remove any of these parts without surgical intervention, the whole person is going to die.

In the example in Figure B-13, a sports league, defined as a sports event that occurs seasonally every year, is a composition of divisions and schedules. A division is an aggregation of teams. A team might exist independently of a particular season. In other words, a team might exist for several seasons (leagues) in a row. Therefore, a team might still exist, even if a division is eliminated. Moreover, a game can only exist in the context of a particular schedule of a particular league.

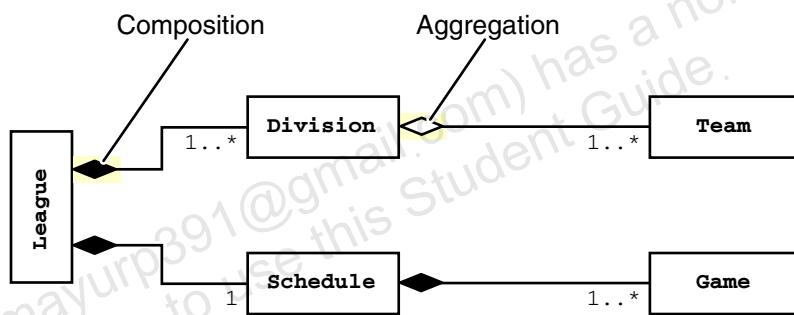


Figure B-13 Aggregation and Composition Example

Association Classes

An association between two classes might have properties and behavior of its own. Association classes are used to model this characteristic. For example, Figure B-14 shows that players might be required to register for a particular division within a sports league. The association class is attached to the association link by a dashed line.

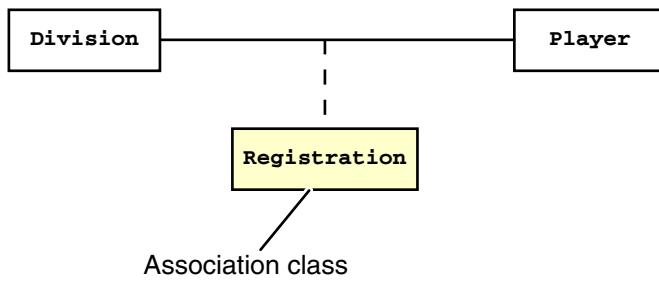


Figure B-14 Simple Association Class

Figure B-15 shows an association class that is used by two associations and that has two private attributes. This example indicates that a Game object is associated with two opposing teams, and that each team will have a score and a flag specifying whether that team forfeited the game.

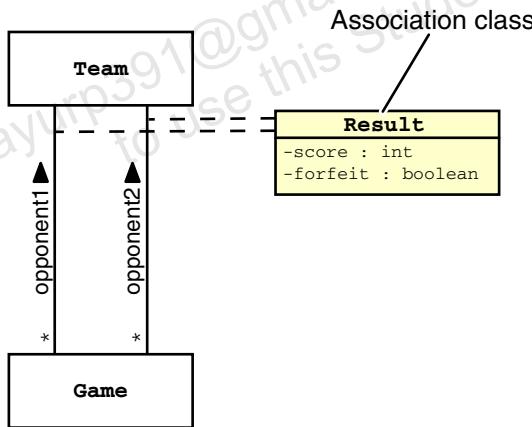


Figure B-15 More Complex Association Class

Class Diagrams

You can use the Java programming language to represent an association class in several different ways. One way is to code the association class as a standard Java technology class. For example, Registration could be coded as follows:

```
public class Registration {  
    private Division division;  
    private Player player;  
    // registration data  
    // methods...  
}  
public class Division {  
    // data  
    public Registration retrieveRegistration(Player p) {  
        // lookup registration info for player  
        return new Registration(this, p, ...);  
    }  
    // methods...  
}
```

Another technique is to code the association class attributes directly into one of the associated classes. For example, the Game class might include the score information as follows:

```
public class Game {  
    // first opponent and score details  
    private Team opponent1;  
    private int opponent1_score;  
    private boolean opponent1_forfeit;  
    // second opponent and score details  
    private Team opponent2;  
    private int opponent2_score;  
    private boolean opponent2_forfeit;  
    // methods...  
}
```

Other Association Elements

There are several other parts of associations. This section presents the constraints and qualifiers elements.

An association constraint enables you to augment the semantics of two or more associations by attaching a dependency arrow between them and tagging that dependency with a constraint. For example, in Figure B-16, the captain and co-captain of a team are also members of the team's roster.

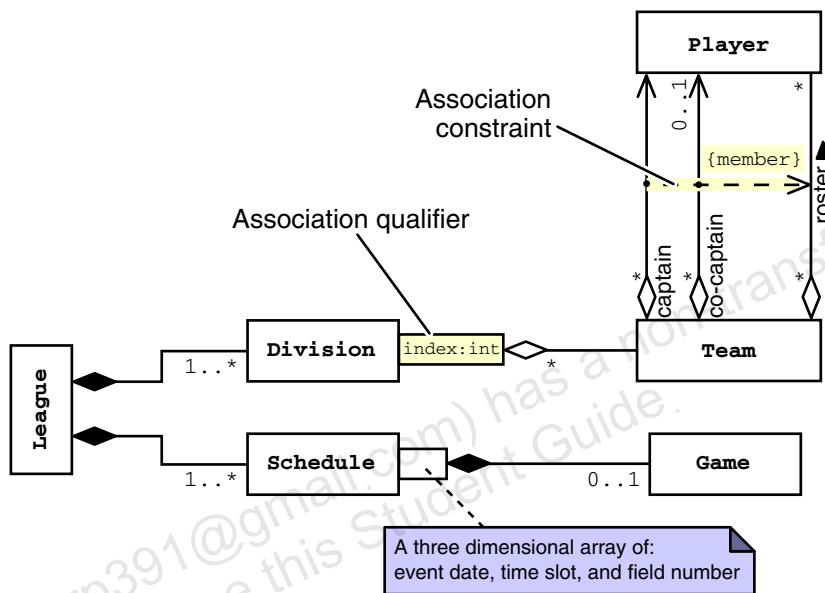


Figure B-16 Other Associations Elements

An association qualifier provides a modeling mechanism to state that an object at one end of the association can look up another object at the other end. For example, a particular game in a schedule is uniquely identified by the following:

- An event date – Such as Saturday, August 12, 2000
- A time-slot – Such as 11:00 a.m. to 12:30 p.m.
- A particular field number – Such as field number 2

One particular implementation might be a three dimension array (for example, Game [] [] []), in which each qualifier element (date, time-slot, and field number) is mapped to an integer index.

Object Diagrams

An Object diagram represents the static structure of a system at a particular instance in time. These diagrams are composed of object nodes, associations, and sometimes class nodes.

Figure B-17 shows a hierarchy of objects that represent a set of teams in a single division in a soccer sports league. This diagram shows one configuration of objects at a specific point in the system. Object nodes only show instance attributes because methods are elements of the class definition. Also, an Object diagram does need not to show every associated object, it just needs to be representative.

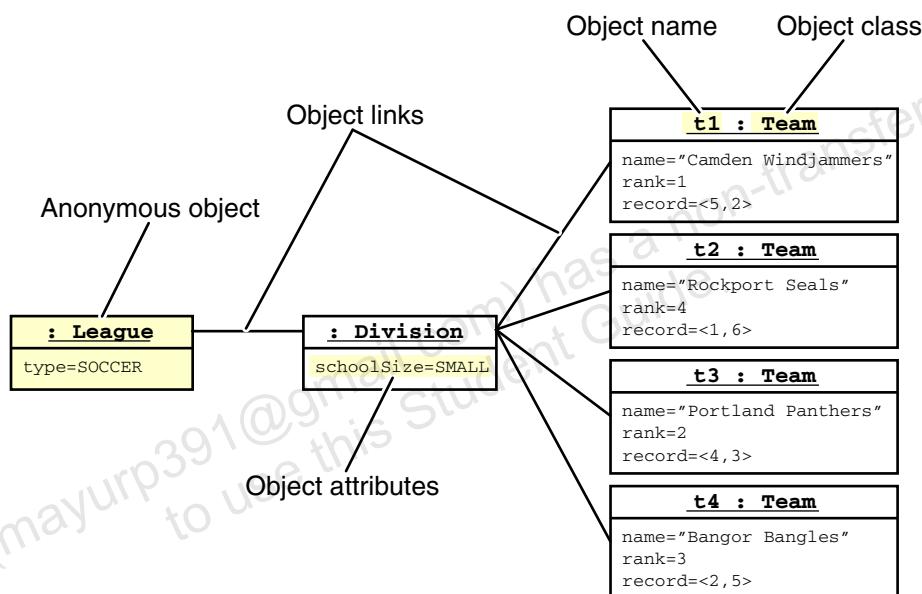


Figure B-17 Object Diagram Example

Figure B-18 shows two objects, `c1` and `c2`, with their instance data. They refer to the class node for `Count`. The dependency arrow indicates that the object is an instance of the class `Count`. The objects do not include the `counter` attribute because it has class scope.

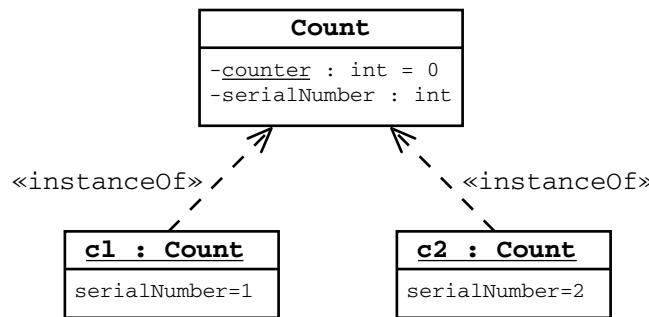


Figure B-18 Object Diagram Example

Collaboration Diagrams

Collaboration Diagrams

A Collaboration diagram represents a particular behavior shared by several objects. These diagrams are composed of objects, their links, and the message exchanges that accomplish the behavior.

Figure B-19 shows a Collaboration diagram in which an actor initiates a login sequence within a web application using a servlet.

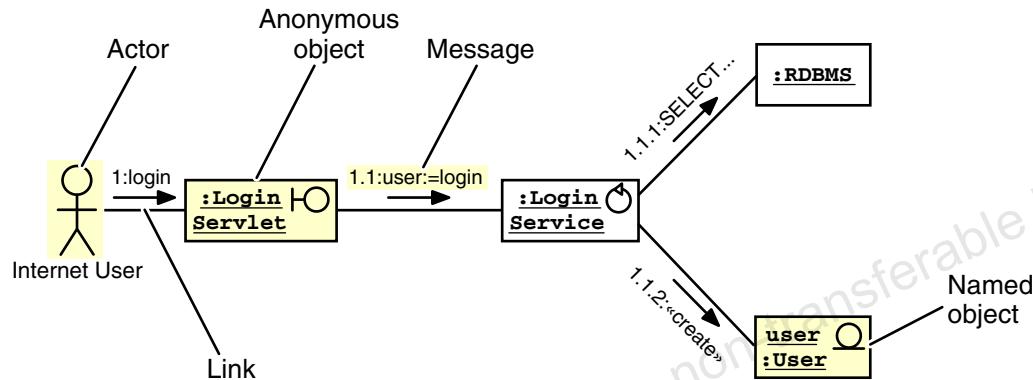


Figure B-19 User-Driven Collaboration Diagram

The servlet uses an object of the `LoginService` class to perform the lookup of the username, verify the password, and create the `User` object. The links between objects show the dependencies and collaborations between these objects. Messages between objects are shown by the messages on the links. Messages are indicated by an arrow in the direction of the message and a text string declares the type of message. The text of this message string is unrestricted. Messages are also labeled with a sequence number so you can see the order of the message calls.

Figure B-20 shows a more elaborate Collaboration diagram. In this diagram, some client object initiates an action on a session bean. This session bean then performs two database modifications within a single transaction context.

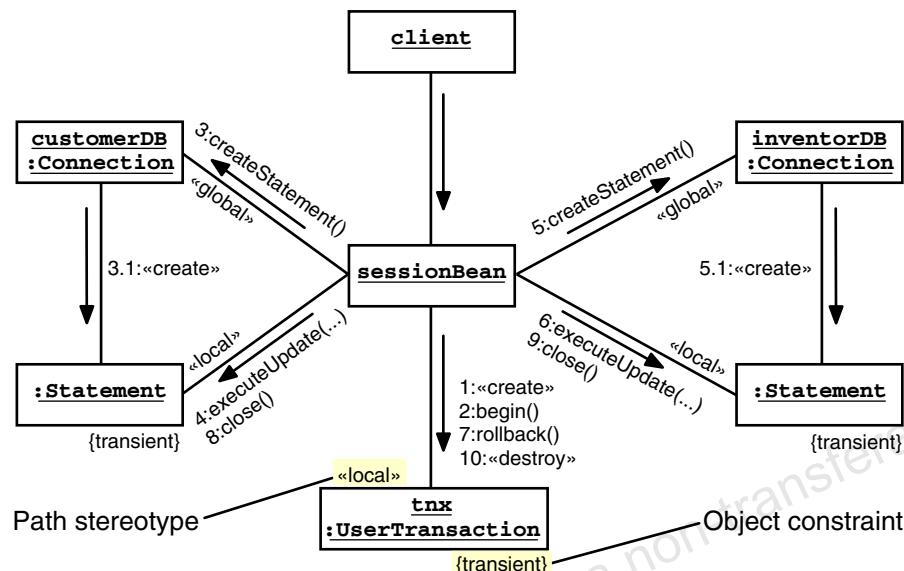


Figure B-20 Collaboration Diagram

You can label the links with a stereotype to indicate if the object is global or local to the call sequence. In this example, the connection objects are global, and the statement and transaction objects are local.

You can also label objects with a constraint to indicate if the object is transient.

Sequence Diagrams

A Sequence diagram represents a time sequence of messages exchanged between several objects to achieve a particular behavior. A Sequence diagram enables you to understand the flow of messages and events that occur in a certain process or collaboration. In fact, a Sequence diagram is just a time-ordered view of a Collaboration diagram, as shown in Figure B-19 on page B-24.

Figure B-21 shows a Sequence diagram in which an actor initiates a login sequence within a web application, which uses a servlet. This diagram is equivalent to the Collaboration diagram in Figure B-19 on page B-24. An important aspect of this type of diagram is that time is moving from top to bottom. The Sequence diagram shows the time-based interactions between a set of objects or roles. Roles can be named or anonymous, and usually have a class associated with them. Roles can also be actors.

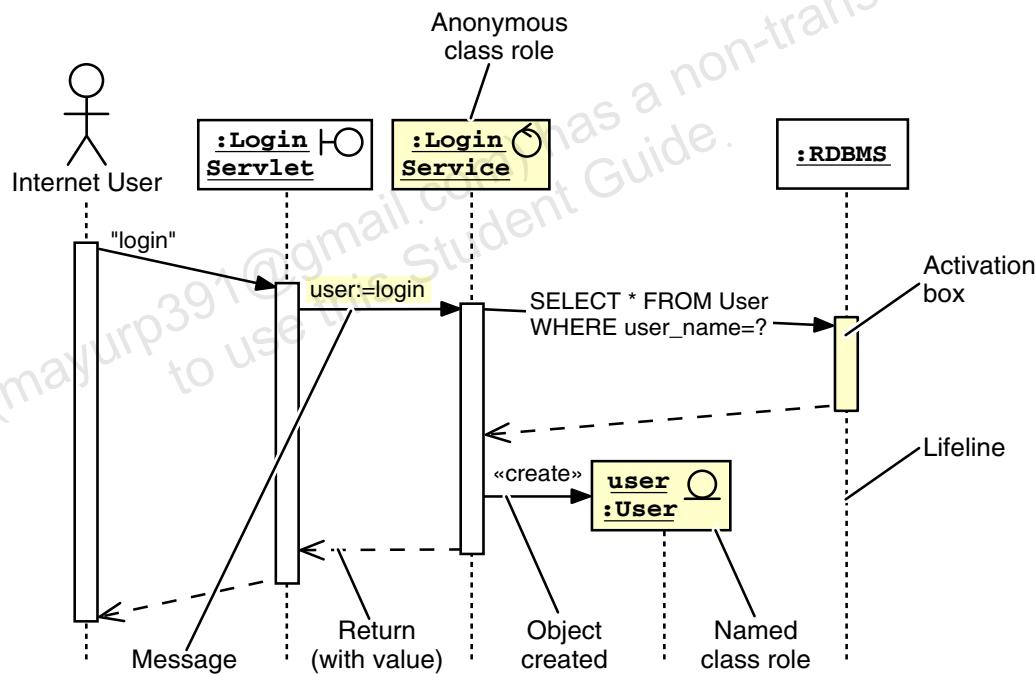
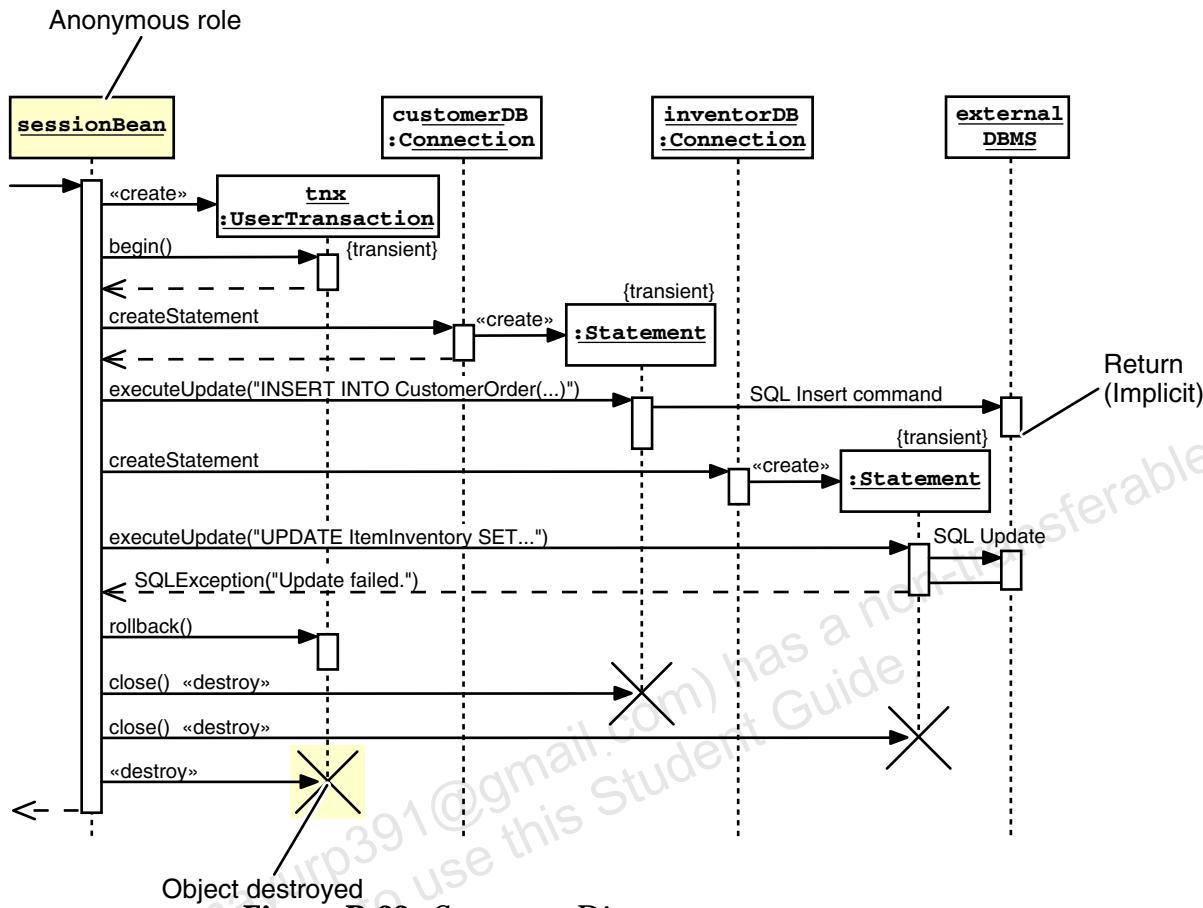


Figure B-21 User-Driven Sequence Diagram Example

Notice the message arrows between the servlet and the service object. The arrow is perfectly horizontal, which indicates that the message is probably implemented by the local method call. Notice that the message arrow between the actor and the servlet is angled, which indicates that the message is sent between components on different machines. For example, an HTTP message from the user's web browser to the web container that handles the login servlet.

Figure B-22 shows a more elaborate Sequence diagram. This diagram is equivalent to the Collaboration diagram in Figure B-20 on page B-25.



This example shows a few more details about Sequence diagrams. First, the return arrow is not always important. A return arrow is implicit at the end of the activation bar. Also, Sequence diagrams can show the creation and destruction of objects explicitly. Every role has a lifeline that extends from the base of the object node vertically. Roles at the top of the diagram existed before the entry message (into the left-most role). Roles that have a message arrow pointing to the head of the role node with the «create» message are created during the execution of the sequence. The destruction of an object is shown with a large cross that terminates the role's lifeline.



Note – Sequence diagrams can also show asynchronous messages. This type of message uses a solid line with stick arrow head: .

Statechart Diagrams

Statechart Diagrams

A Statechart diagram represents the states and responses of a class to external and internal triggers. You can also use a Statechart diagram to represent the life cycle of an object. The definition of an object state is dependent on the type of object and the level of depth you want to model.



Note – A Statechart diagram is recognized by several other names, including State diagram and State Transition diagram.

Figure B-23 shows an example Statechart diagram. Every Statechart diagram should have an initial state (the state of the object at its creation) and a final state. By definition, no state can transition into the initial state and the final state cannot transition to any other state.

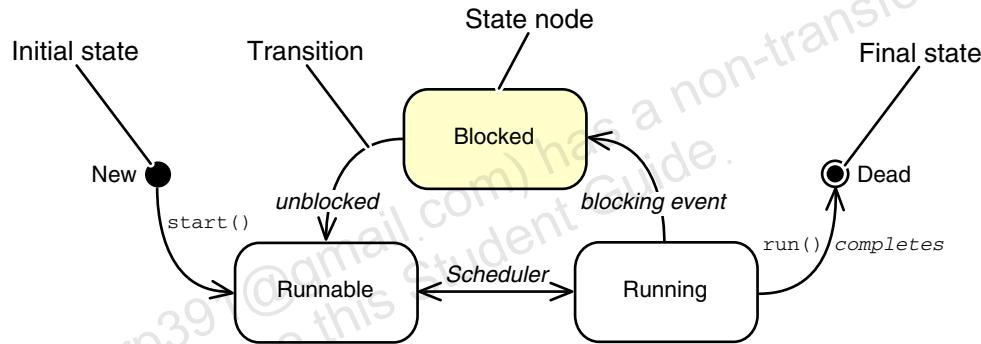


Figure B-23 Statechart Diagram Example

There is no pre-defined way to implement a Statechart diagram. For complex behavior, you might consider using the State design pattern.

Transitions

A transition has five elements:

- Source state – The state affected by the transition
- Event trigger – The event whose reception by the object in the source state makes the transition eligible to fire, providing its guard condition is satisfied
- Guard condition – A Boolean expression used to determine if the state transition should be made when the event trigger occurs
- Action – A computation or operation performed on the object that makes the state transition
- Target state – The state that is active after the completion of the transition

Figure B-24 shows a detailed transition.

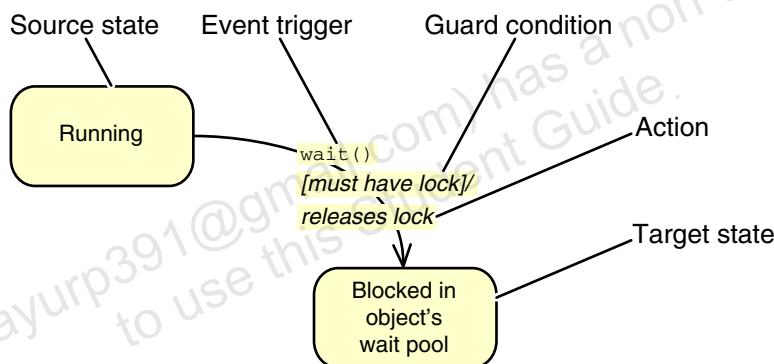


Figure B-24 State Transition Example

Activity Diagrams

An Activity diagram represents the activities or actions of a process without regard to the objects that perform these activities.

Figure B-25 shows the elements of an Activity diagram. An Activity diagram is similar to a flowchart. There are activities and transitions between them. Every Activity diagram starts with a single start state and ends in a single stop state.

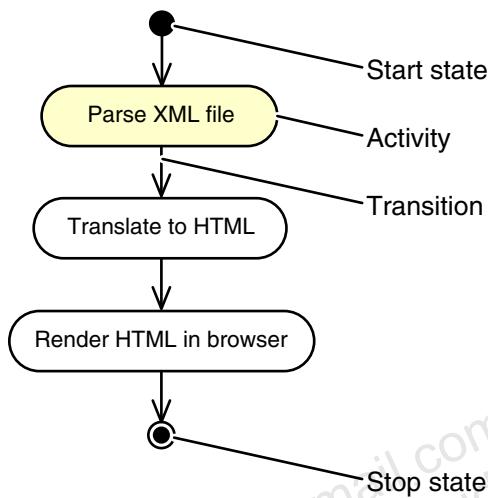


Figure B-25 Activities and Transitions

Figure B-26 demonstrates branching and looping in Activity diagrams. The diagram models the higher level activity of Verify availability of products in a purchase order. The top-level branch node forms a simple while loop. The guard on the transition below the branch is true if there are more products in the order to be processed. The else transition halts this activity.

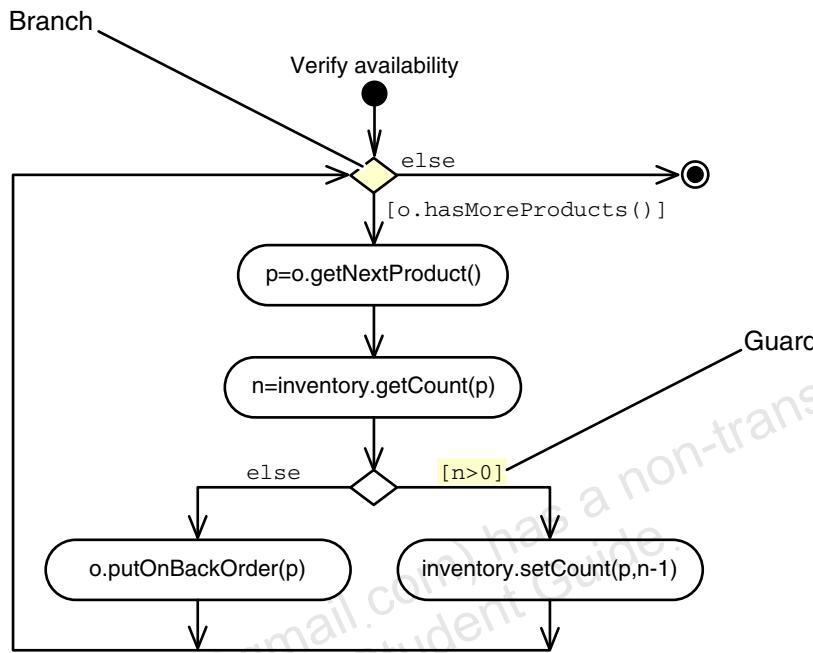


Figure B-26 Branching and Looping

Activity Diagrams

Figure B-27 shows a richer Activity diagram.

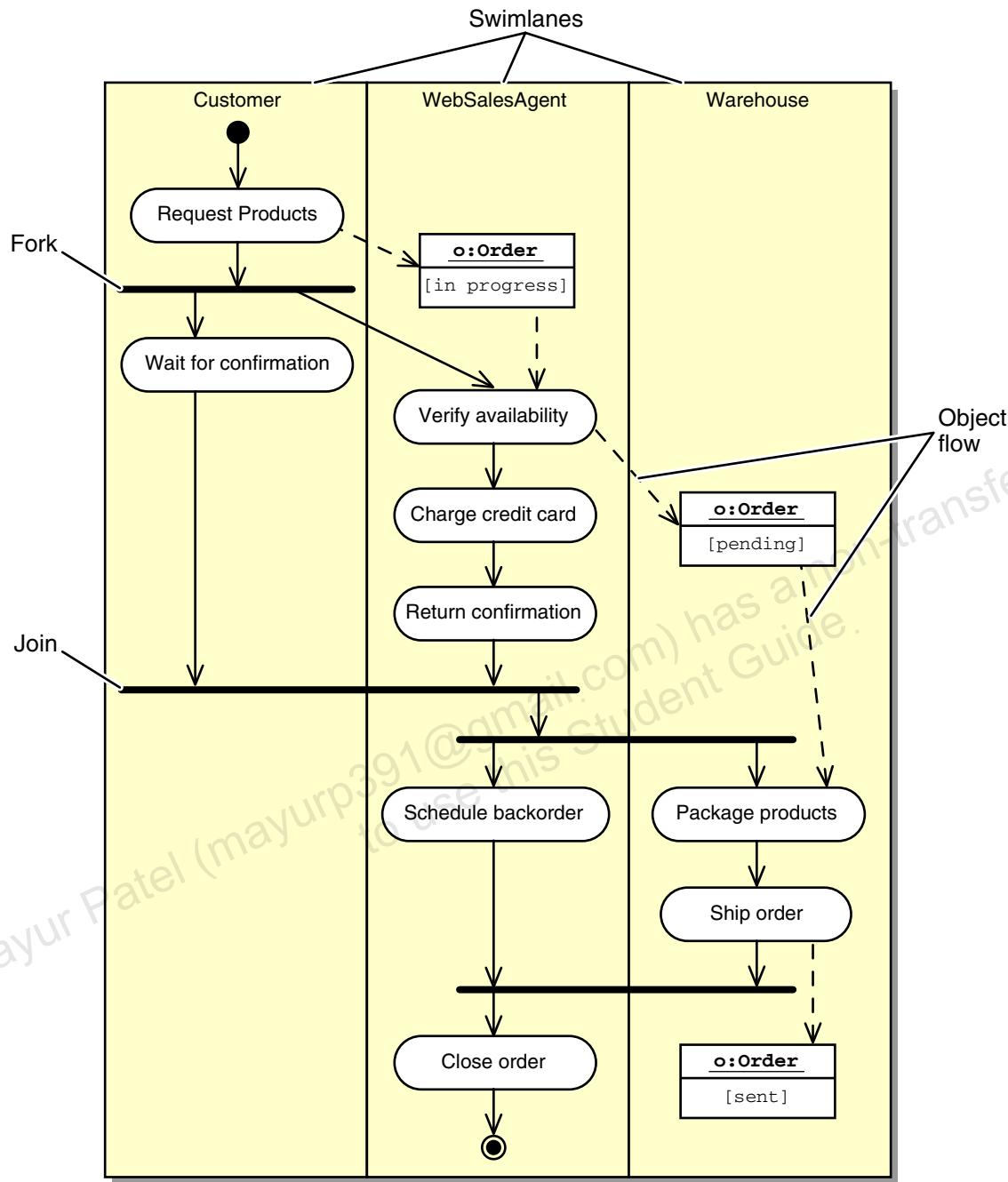


Figure B-27 Activity Diagram Example

In the example in Figure B-27 on page B-32, swim lanes are used to isolate the actor of a given set of activities. These actors might include humans, systems, or organization entities. This diagram also demonstrates the ability to model concurrent activities. For example, the Customer initiates the purchase of one or more products on the company's web site. The Customer then waits as the WebSalesAgent software begins to process the purchase order. The fork bar splits a single transition into two or more transitions. The corresponding join bar must contain the same number of inbound transitions.

Component Diagrams

Component Diagrams

A Component diagram represents the organization and dependencies among software implementation components.

Figure B-28 shows three types of icons that can represent software components. Example 1 is a generic icon. Example 2 is an icon that represents a source file. Example 3 is an icon that represents a file that contains executable (or object) code.

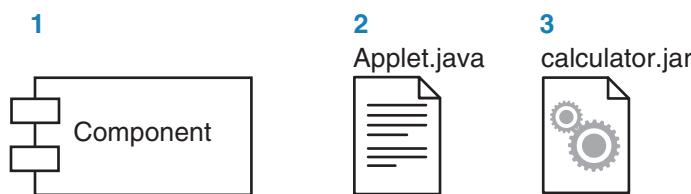


Figure B-28 Component Nodes Example

Figure B-29 shows the dependencies of packaging an HTML page that contains an applet. The HTML page depends on the JAR file, which is constructed from a set of class files. The Class files are compiled from the corresponding source files. You can use a tagged value to indicate the source control version numbers on the source files.

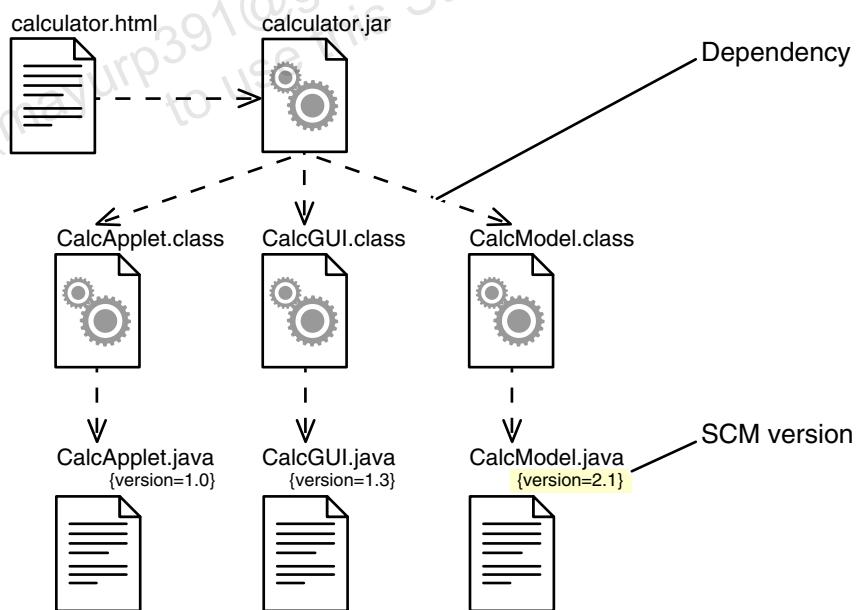


Figure B-29 Component Diagram Example



Note – SCM is Source Control Management.

Figure B-30 shows another Component diagram. In this diagram, several components have an interface connector. The component attached to the connector implements the named interface. The component that has an arrow pointing to the connector depends on the fact that component realizes that interface.

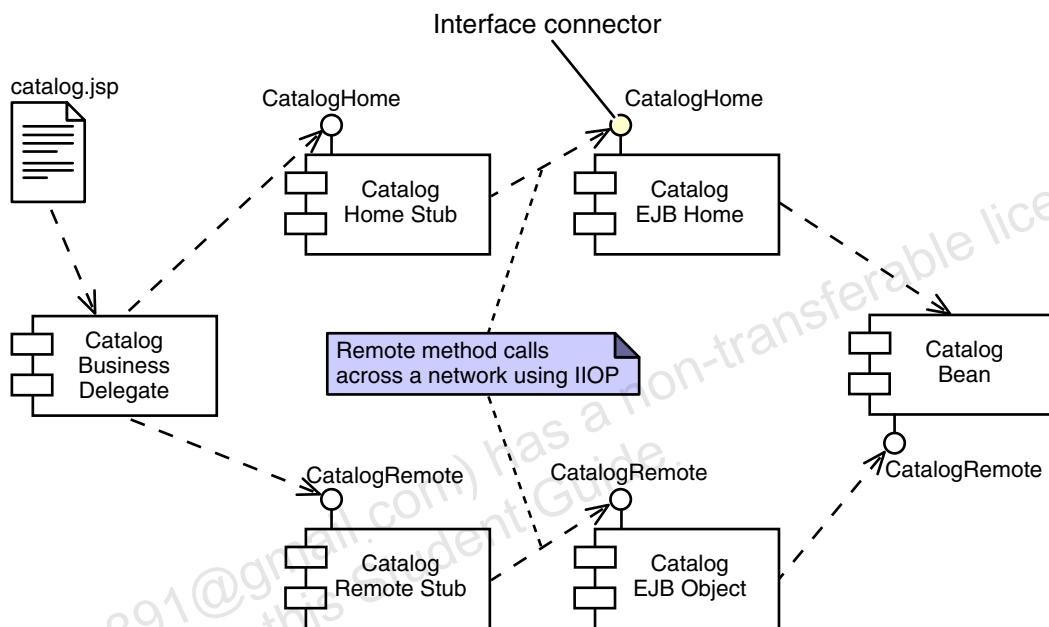


Figure B-30 Component Diagram With Interfaces

In this J2EE technology example, the web tier includes a catalog JSP, which uses a catalog Business Delegate JavaBeans component to communicate to the Enterprise JavaBeans™ (EJB™) technology tier. Every enterprise bean must include two interfaces. The home interface enables the client to create new enterprise beans on the EJB server. The remote interface enables the client to call the business logic methods on the (remote) enterprise bean. The business delegate communicates with the catalog bean through local stub objects that implement the proper home and remote interfaces. These objects communicate over a network using the Internet Inter-ORB protocol with remote *skeletons*. In EJB technology terms, these objects are called `EJBHome` and `EJBObject`. These objects communicate directly with the catalog bean that implements the true business logic.

Deployment Diagrams

Deployment Diagrams

A Deployment diagram represents the network of processing resource elements and the configuration of software components on each physical element.

A Deployment diagram is composed of hardware nodes, software components, software dependencies, and communication relationships. Figure B-31 shows an example in which the client machine communicates with a web server using HTTP over TCP/IP.

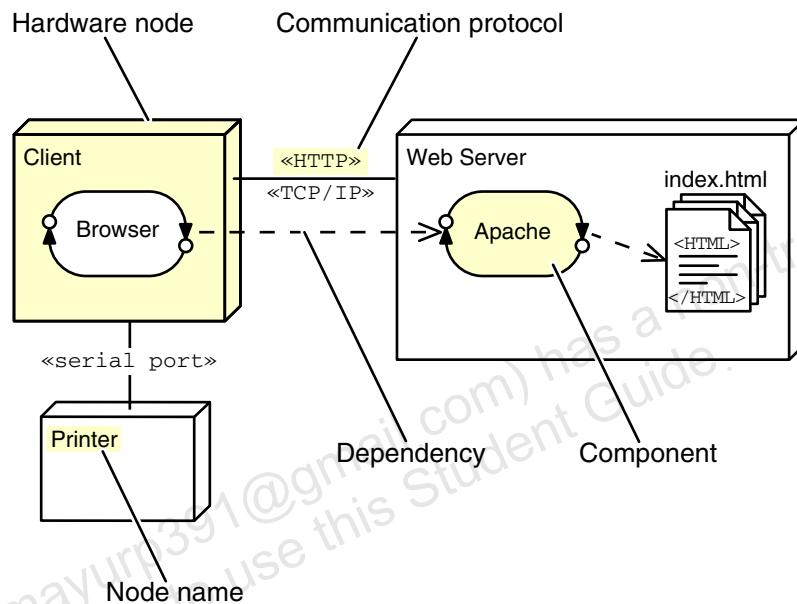


Figure B-31 Deployment Diagram Example

The client is running a web browser, which is communicating with an Apache web server. Therefore, the browser component depends on the Apache component. Similarly, the Apache application depends on the HTML files that it serves. The client machine is also connected to local printer using a parallel port.

You can use a Deployment diagram to show how the logical tiers of an application architecture are configured into a physical network.

Appendix C

Swing Components

Objectives

At the end of this appendix, you should be able to:

- Describe the appearance of each Swing component.

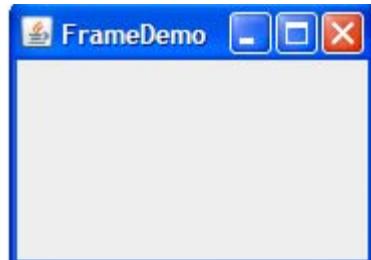
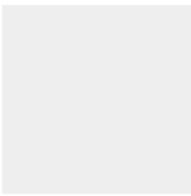
Swing Component Examples

This appendix shows examples of many of the Swing components along with a brief description of that component.

Top Level Containers

Table C-1 contains a brief description of each top-level container.

Table C-1 Top-level Containers

Container	User Interface
<p>A Frame is the basic window used in most GUI applications. It has a border and a title. Other components can be added to the frame. You can also draw in the window. There is a provision to add menus to the frame. The <code>javax.swing.JFrame</code> class is used to create frames.</p>	
<p>JDialog is used to create a dialog window. The API provides several different versions of constructors for defining dialogs. Dialogs are dependent on the frames. Dialogs can be used to take input from the user and confirm any critical actions. Dialogs can be used to display warnings, errors, questions, and information to the user.</p>	
<p>The JWindow container is similar to JFrame but it does not have a border or title bar. There are no window management services.</p>	
<p>The JApplet container is used to create a UI that is run in a web browser. Usually, JApplets are embedded in a web page and can be used to run animations. Other components and menus can be added to this container. You can also draw in an applet.</p>	

General-purpose Containers

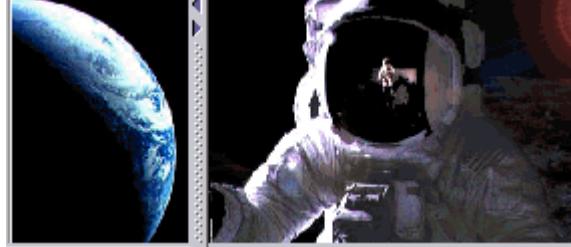
General-purpose Containers

General-purpose containers are intermediate containers, for use in several circumstances, for example: JPanel, JScrollPane, JToolBar, JSplitPane, and JTabbedPane. All of these components extend JComponent. Table C-2 contains a brief description of each general-purpose container.

Table C-2 General-purpose Containers

Container	User interface
Panels are containers into which several components can be added. They also provide a surface to draw on. Unlike JFrame, they are not top-level containers. Panels should be contained in top-level containers. The javax.swing.JPanel class is used to create panels. It extends JComponent and not java.awt.Panel.	
Scroll panes are very handy when the amount of space is limited. They are used to display large components or images. Scroll panes have two scroll bars, a row header, and a column header. The javax.swing.JScrollPane class is used to create scroll panes.	
Toolbars are a group of buttons with icons for easily accessing the frequently used functions. They can be considered as shortcuts to the actions in menus. The javax.swing.JToolBar class is used to create toolbars.	

Table C-2 General-purpose Containers (Continued)

Container	User interface
<p>Split panes display two or more components separated by a divider. Components can be displayed side by side or one over the other. By dragging the divider the amount of space for each component can be adjusted. The <code>javax.swing.JSplitPane</code> class is used to create split panes.</p>	
<p>Tabbed panes are also useful when the space is limited. Several tabs share the same space. At any one time only one tab is visible. To be displayed, a tab needs to be selected by the user. The <code>javax.swing.JTabbedPane</code> class is used to create tabbed panes.</p>	

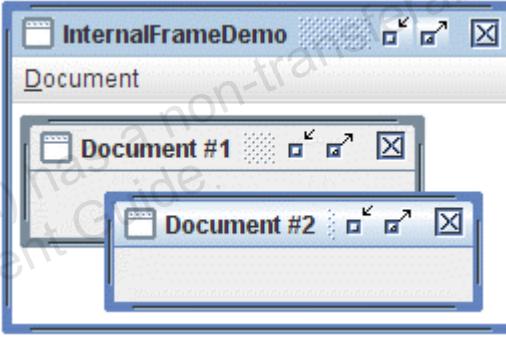
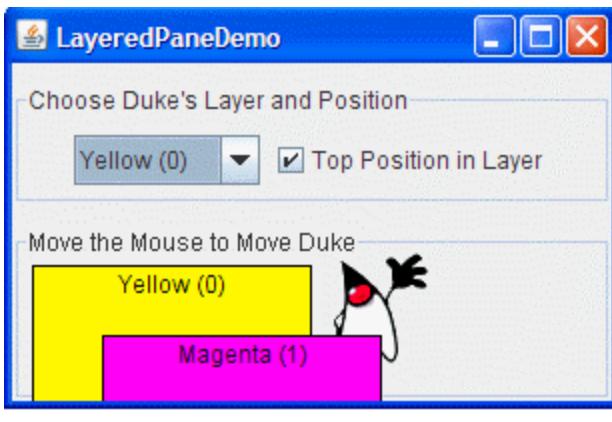
Special-Purpose Containers

Special-Purpose Containers

`JInternalFrame` and `JLayeredPane` are examples of special-purpose containers. They play specific roles in a user interface. Internal frames are designed to work within desktop panes. They are not top-level, such as `JFrame`. The `JLayeredPane` container helps to specify the depth of the component, which is helpful for rendering the GUI when components overlap.

Table C-3 contains a brief description of the special-purpose containers.

Table C-3 Special-purpose Containers

Containers	User Interface
Internal frames are non top-level containers. They contain features similar to <code>JFrames</code> such as dragging, resizing, iconifying, and maximizing. Internal frames are created using the <code>javax.swing.JInternalFrame</code> class, which is added to <code>JDesktopPane</code> and is, in turn, added to a <code>JFrame</code> . As with regular frames, components can be added to a <code>JInternalFrame</code> .	
Layered panes allow the addition of components at required depths. The depth is specified as an integer value. The <code>javax.swing.JLayeredPane</code> class is used to create layered panes. For convenience, you can use the standard layers defined by this class: <code>DEFAULT_LAYER</code> , the bottommost layer, <code>PALETTE_LAYER</code> , <code>MODAL_LAYER</code> , <code>POPUP_LAYER</code> , <code>DRAG_LAYER</code> the topmost layer.	

When using container classes, you should keep the following rules in mind:

- GUI components are displayed only when they are in a containment hierarchy. A containment hierarchy is a tree of components that has a top-level container as its root.
- A GUI component object instance can appear only once in a containment tree. If a component in one container is added to another container, the component is moved to the latter container and removed from the former.
- Each top-level container has a content pane that, generally speaking, contains (directly or indirectly) the visible components in that top-level container's GUI.
- You can optionally add a menu bar to a top-level container. The menu bar is, by convention, positioned within the top-level container, but outside the content pane. Some look-and-feels, such as the Mac look-and-feel, give you the option of placing the menu bar in another place more appropriate for the look-and-feel, such as at the top of the screen.

Each of these four containers (including `JApplet`) implements a special interface called `RootPaneContainer`. An in-depth examination of the `RootPaneContainer` is outside the scope of this module.

JFrame Container Essentials

The `JFrame` container is the most commonly used top-level Swing container. The `JFrame` container permits you to set one of four reactions for the Close Window menu button. These reactions are:

- `DO NOTHING ON CLOSE`
- `HIDE ON CLOSE`
- `DISPOSE ON CLOSE`
- `EXIT ON CLOSE`

You set the option by invoking the `setDefaultCloseOperation` method on the `JFrame` instance.

Buttons

Regular buttons, check boxes, and radio buttons are all considered to be in the button category. Creating graphical buttons is straight forward if you use an `Icon` object that defines the graphic you want displayed. The `JCheckBox` class provides support for check box buttons. The `JRadioButton` class behaves in such a way that turning *on* a radio button in a radio button group causes all the other radio buttons in that group to be turned *off*. Table C-4 describes each of these components and shows their user interface.

Table C-4 Buttons

Component	User Interface
<p><code>JButton</code> objects can be created with a simple <code>String</code> argument to the constructor, in which case they display that text as their label. Clicking on a <code>JButton</code> generates an <code>ActionEvent</code>. You might want to set the action command property of your button so that the <code>ActionEvent</code> carries a particular command string. If you create a <code>JButton</code> with text, the label text is used by default as the action command string. However, if you create a graphics-only button or if the default action command string (which is the text label of the button) is not what you need, you can define the action command explicitly using the <code>setActionCommand</code> method.</p>	
<p>Check boxes are similar to <code>JButtons</code> in that they can be initialized with a simple <code>String</code> argument to the constructor, in which case they display that text as their label. But their selection model is different, by convention. A check box has a boolean state value that can be in either <i>on</i> (true) or <i>off</i> (false). Clicking the check box toggles its state from <i>on</i> to <i>off</i>, or from <i>off</i> to <i>on</i>. The <code>javax.swing.JCheckBox</code> class is used to create check boxes.</p>	

Table C-4 Buttons (Continued)

Component	User Interface
<p>Individually, a <code>JRadioButton</code> simply toggles on and off each time it is selected, just like a <code>JCheckBox</code>. To obtain the mutual exclusion behavior of a radio button, add the buttons to a <code>ButtonGroup</code> instance. A button group is a manager that ensures that only one button is selected at one time. Use the <code>ButtonGroup</code> class to create a button group.</p>	

Text Components

Swing text components can be broadly divided into three categories.

- Text controls – JTextField, JPasswordField (for user input)
- Plain text areas – JTextArea (displays text in plain text, also for multi-line user input)
- Styled text areas – JEditorPane, JTextPane (displays formatted text)

Table C-5 describes each of these components and shows their user interface.

Table C-5 Text Components

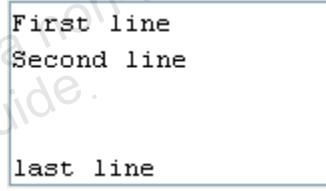
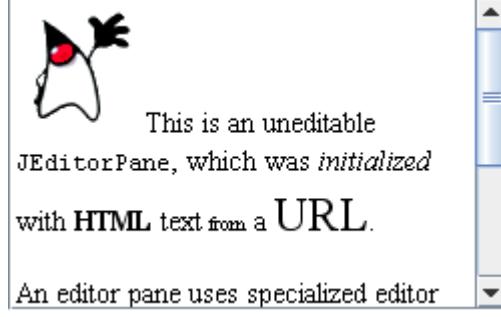
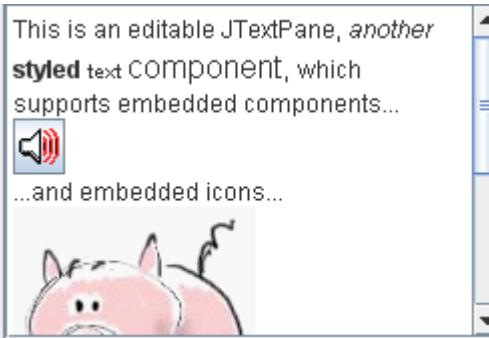
Component	User Interface
Text areas are commonly used to collect more than one line of information from the user. The javax.swing.JTextArea class is used to create text areas. When creating a text area, you can specify the number of rows, columns, and initial content. Text areas display plain text only.	
Text fields are also used to collect input from the user. These are similar to text areas but they are restricted to accept only one line of text. The javax.swing.JTextField class is used to create text fields.	
Editor panes are styled text components. In addition to plain text, editor panes can display and edit text in RTF and HTML format. Editor panes are commonly used to display help in HTML format. The javax.swing.JEditorPane class is used to create editor panes.	 <p>This is an uneditable JEditorPane, which was <i>initialized</i> with HTML text from a URL.</p> <p>An editor pane uses specialized editor</p>

Table C-5 Text Components (Continued)

Component	User Interface
<p>The <code>javax.swing.JTextPane</code> class inherits from <code>javax.swing.JEditorPane</code>. In addition to providing all the features of the <code>JEditorPane</code>, the <code>JTextPane</code> class also allows for embedding components.</p>	<p>This is an editable <code>JTextPane</code>, another styled text component, which supports embedded components...</p>  <p>...and embedded icons...</p>
<p><code>javax.swing.JPasswordField</code> is a text input field specialized for password entry. For security, a password field displays a character such as an asterisk '*'. A password field's value is stored as an array of characters, instead of a string. Like any other text field object, a password field sends an action event when you press the Enter key.</p>	

Uneditable Information Display Components

Uneditable information display components are used to display more information about the components. These components can be used only as display components. Table C-6 describes some of these components.

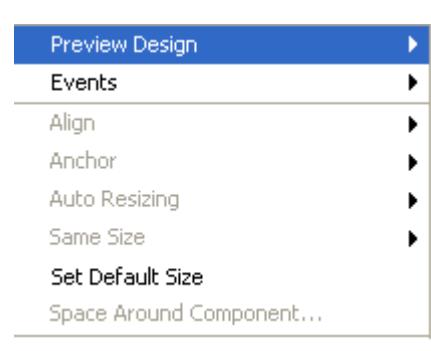
Table C-6 Uneditable Display Components

Component	User Interface
Labels are used to display text on the screen. They are uneditable components. The <code>javax.swing.JLabel</code> class is used to create labels. Labels can also be used to display images.	
The <code>javax.swing.JToolTip</code> class is helpful to display information about the components. <code>JComponent</code> provides a method called the <code>setToolTipText</code> , which can be used by all the components to set the string that should be displayed.	
The <code>javax.swing.JProgressBar</code> class is very helpful in indicating the progress of any long-running tasks.	

Menus

Menus behave similar to lists except, by convention, a menu typically is displayed either in a menu bar or as a popup menu. A menu bar can contain one or more menus (called pull-down menus) and has an operating system-dependent location, typically, at the top of each window. A popup menu is displayed when the user triggers a platform-specific mouse button or keyboard sequence, for example, pressing the right mouse button, or rolling the mouse cursor over a popup-enabled component. Table C-7 describes each of these components and shows their user interface.

Table C-7 Menu Components

Component	User interface
<p>A menu bar contains the names of one or more pull down menus. Clicking these names opens menu items and submenus. The <code>javax.swing.JMenu</code>, <code>javax.swing.JMenuItem</code> classes are used to create menus. Menu items can be selected by using keyboard mnemonics and accelerators. Menu items could be of type check boxes and radio buttons.</p>	
<p>A popup menu is a menu that is not attached to the menu bar. These menus are sometimes referred to as context menus. The <code>javax.swing.JPopupMenu</code> class is used to create popup menus.</p>	

Formatted Display Components

Formatted Display Components

Formatted display components are among the most complex components found in Swing. Tables, trees, color chooser, and file chooser are a few examples in this category. Table C-8 describes these components and shows their user interface.

Table C-8 Formatted Display Component

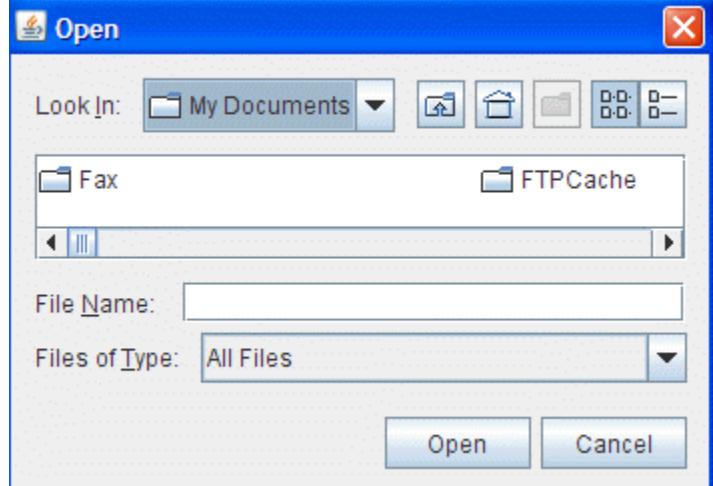
Component	User Interface
Tables are used to display and edit information in the form of a grid. The javax.swing.JTable class is used to create tables. JTable does not store the data, but only displays the data from the table model.	
JTree is used to display hierarchical information. The data for constructing the display is provided through a TreeModel instance. The data can also be provided as a collection of elements such as a Hashtable or a Vector. A JTree does not actually store the data, but rather presents the data in the TreeModel instance.	
The JFileChooser class allows users to navigate to the file system and choose a file. The file chooser contains filter methods you can use to filter the types of files displayed and methods you can use to customize the view presented by the file chooser.	

Table C-8 Formatted Display Component (Continued)

Component	User Interface
<p>JColorChooser allows the user to manipulate and select a color. A color can be selected in three different ways. Swatches, Hue, Saturation, and Brightness (HSB) or Red, Green, and Blue (RGB). The JColorChooser class provides several constructors for creating the color-chooser pane. The default constructor creates a pane with initial color as white. Another constructor takes the initial color as a parameter. You can also specify the color selection model in the constructor.</p>	

Other Basic Controls

This section describes other Swing components, such as combo boxes, lists, sliders, and spinners that are often used in GUIs. A `JComboBox` lets the user choose one of several choices. The `JComboBox` class has a convenient constructor that takes an array of objects to use as the initial choices. You can add and remove choices using the `addItem` and `removeItem` methods, respectively. A `JList` presents items in one or more columns. You choose one or more items from the display by clicking or navigating with keyboard commands. With `JSlider`, you use a mouse click and drag to enter a numeric value. A spinner is a possible alternative to a slider when screen space is limited.

Table C-9 describes each of these components and shows their user interface.

Table C-9 Other Basic Controls

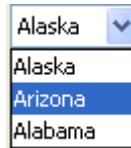
Component	User Interface
JComboBox has two forms: Editable and UnEditable. The default form, uneditable, shows a button and a drop-down list of values. The editable form presents a text edit field with a selection button. You can type a value in the text field or use the button to display a drop-down list of choices.	
Lists can feature scroll bars if the number of items is too large to display in the allotted screen area. Lists can also be made resizable. The <code>JList</code> class is simple to use in simple circumstances, and can create its own <code>ListModel</code> if needed. To do this, you put the data items into either a <code>Vector</code> or an array of <code>Objects</code> and invoke the <code>JList</code> constructor using the data as an argument.	

Table C-9 Other Basic Controls (Continued)

Component	User Interface
Spinners also allow you to type in a value. JSpinner has three subcomponents: an up arrow, a down arrow, and an editor. The editor can be any JComponent, but, by default, it is implemented as a panel with formatted text field	
Slider values have a finite range, that is, a minimum and maximum value. If the ability to specify precise numbers is important, a slider can be coupled with a formatted text field.	 <p>Frames Per Second</p> <p>0 10 20 30</p>

